CAPI User Guide and Reference Manual

Version 8.0

Copyright and Trademarks

CAPI User Guide and Reference Manual (Macintosh version)

Version 8.0

December 2021

Address	Telephone	Fax
LispWorks Ltd St. John's Innovation Centre Cowley Road Cambridge CB4 0WS England	From North America: 877 759 8839 (toll-free) From elsewhere: +44 1223 421860	From North America: 617 812 8283 From elsewhere: +44 870 2206189

www.lispworks.com

Contents

Index

Preface

This preface contains information you need when using the rest of the CAPI documentation. It discusses the purpose of this manual, the typographical conventions used, and gives a brief description of the rest of the contents.

About this manual

This manual contains a user guide section (previously published separately as the *CAPI User Guide*) and a reference section (previously the *LispWorks CAPI Reference Manual*).

Assumptions

The CAPI documentation assumes that you are familiar with:

- LispWorks.
- Common Lisp and CLOS, the Common Lisp Object System.
- macOS.

Illustrations in this manual show the CAPI running on macOS 10.5, so if you use a different version you should expect some variation from the figures depicted here.

Unless otherwise stated, examples given in this document assume that the current package has **`CAPI`** on its package-use-list.

Conventions used in the manual

Throughout this manual, certain typographical conventions have been adopted to aid readability.

1. Whenever an instruction is given, it is numbered and printed like this.

Text which you should enter explicitly is printed **`like this`**.

Exported symbols and example code are printed **`like-this`**. The package qualifier is often omitted, as if the current package is **`capi`** (or **`graphics-ports`** or **`color`**.)

Variable arguments, slots and return values are italicised. They look *like-this* in the main text.

User Guide section

The user guide section of this manual forms an introductory course in developing applications using the CAPI. Please note that, like the rest of the LispWorks documentation, it does assume knowledge of Common Lisp.

1 Introduction to the CAPI, introduces the principles behind the CAPI, some of its fundamental concepts, and what it sets out to achieve.

2 Getting Started, presents a series of simple examples to familiarize you with some of the most important elements and functions.

3 General Properties of CAPI Panes, introduces more of the fundamental CAPI elements and common themes. These elements are explained in greater detail in the remainder of the manual.

4 General Considerations, covers some general issues that you should be aware of when using CAPI, including information about multiple displays.

5 Choices - panes with items, explains the key CAPI concept of the *choice*. A choice groups CLOS objects together and provides the notion of there being a selected object amongst that group of objects. Button panels and list panels are examples of choices.

6 Laying Out CAPI Panes introduces the idea of *layouts*. These let you combine different CAPI elements inside a single window.

7 Programming with CAPI Windows, outlines basic techniques for modifying existing windows.

8 Creating Menus, shows you how to implement menus.

9 Adding Toolbars, shows you how to add toolbars to a window.

11 Defining Interface Classes - top level windows, introduces the macro **`define-interface`**. This macro can be used to define interface classes composed of CAPI elements, including the predefined elements described in this manual and also elements which you define.

10 Dialogs: Prompting for Input, discusses the ways in which dialogs may be used to prompt the user for input.

12 Creating Panes with Your Own Drawing and Input, shows you how you can define your own classes when the elements provided by the CAPI are not sufficient for your needs.

13 Drawing - Graphics Ports, describes the Graphics Ports API which provides a selection of drawing and image transformation functions. Although not part of the CAPI package, and therefore not strictly part of the CAPI, the Graphics Ports functions are used in conjunction with CAPI panes, and are therefore documented in this manual. See also **22 GRAPHICS-PORTS Reference Entries**.

14 Graphic Tools drawing objects, describes the Graphic Tools API which provides a way to create more complex drawings, including graphs and bar charts. Graphic Tools are built with Graphics Ports and CAPI pinboards, and are therefore documented in this manual. See also **23 LW-GT Reference Entries**.

15 The Color System, allows applications to use keyword symbols as aliases for colors in Graphics Ports drawing functions. They can also be used for backgrounds and foregrounds of windows and CAPI objects. See also **24 COLOR Reference Entries**.

16 Printing from the CAPI—the Hardcopy API, describes the programmatic printing of Graphics Ports.

17 Drag and Drop, describes how you can implement drag and drop in your CAPI application.

19 Host Window System-specific issues, describes how to configure the appearance of CAPI windows on the various supported host window systems.

20 Self-contained examples, enumerates the CAPI example files available in the LispWorks library.

Reference section

The reference section contains reference entries for the symbols in the **`capi`**, **`graphics-ports`**, **`lw-gt`** and **`color`** packages.

Within each chapter, the symbols are organized alphabetically (ignoring non-alphanumeric characters that are common in Lisp symbols, such as *). The typographical conventions used are similar to those used in *Common Lisp: the Language (2nd Edition)*. Further details on the conventions used are given below. The chapters are:

21 CAPI Reference Entries, describes the external symbols of the **`capi`** package.

22 GRAPHICS-PORTS Reference Entries, describes the external symbols of the **`graphics-ports`** package.

23 LW-GT Reference Entries, describes the external symbols of the **lw-gt** package.

24 COLOR Reference Entries, describes the external symbols of the **color** package.

Note: Although the **graphics-ports** and **color** packages are not strictly part of the CAPI, they are included in this manual because the functionality is usually called from CAPI elements such as output panes. **lw-gt** is also included here since it is built on top of **graphics-ports** and **capi**. **13 Drawing - Graphics Ports** and **15 The Color System** shows you how to use the **graphics-ports** and **color** packages respectively; the remainder of the User Guide section shows you how to use the **capi** package.

Conventions used for reference entries

Each entry is headed by the symbol name and type, followed by a number of fields providing further details. These fields consist of a subset of the following: "Summary", "Package", "Signature", "Method signatures", "Arguments", "Values", "Initial value", "Superclasses", "Subclasses", "Initargs", "Accessors", "Readers", "Description", "Notes", "Compatibility notes", "Examples" and "See also".

Some symbols with closely-related functionality are coalesced into a single reference entry.

Entries with a long "Description" section usually have as their first field a short "Summary" providing a quick overview of the symbol's purpose.

The "Package" section shows the package from which the symbol is exported.

The "Signature" section shows the arguments and return values of functions and macros, and the parameters of types.

In a Generic Function entry there may be a "Method signatures" section showing system-defined method signatures.

The "Arguments" and "Values" sections show types of the arguments and return values.

In a Variable entry, the "Initial value" section shows the initial value.

In a Class entry the "Subclasses" section of lists the external subclasses, though not subclasses of those, and the "Superclasses" section lists the external superclasses, though not superclasses of those. The "Initargs" section describes the initialization arguments of the class, though note that initargs of superclasses are also valid. There may be an "Accessors" section listing accessor functions which are both readers and writers, and/or a "Readers" section listing accessor functions which are only readers. Accessor functions access the slot with matching name.

The "Description" section contains the detail of what the symbol does, how each argument is interpreted (and its default value if applicable), and how each return value is derived. More incidental information may be shown in a "Notes" section.

A few entries have a "Compatibility notes" section describing changes in the symbol's functionality relative to other LispWorks versions.

Examples are given under the "Examples" heading. Short examples are shown directly. Longer examples are supplied as source files in your LispWorks installation directory under **examples/capi/**. The convenience function **lw:example-edit-file** allows you to open these files in the LispWorks editor.

Note that the example code is written with explicit package qualifiers such as **capi:interface**, so that it can be run as-is, regardless of the current package.

Finally, the "See also" section provides links to other related symbols and user guide sections.

Viewing example files

This manual often refers to example files in the LispWorks library via a Lisp form like this:

```
(example-edit-file "capi/choice/drag-and-drop")
```

These examples are Lisp source files in your LispWorks installation under `lib/8-0-0-0/examples/`. You can simply evaluate the given form to view the example source file.

Example files contain instructions about how to use them at the start of the file.

The examples files are in a read-only directory and therefore you should compile them inside the IDE (by the Editor command **Compile Buffer** or the toolbar button or by choosing **Buffer > Compile** from the context menu), so it does not try to write a fasl file.

If you want to manipulate an example file or compile it on the disk rather than in the IDE, then you need first to copy the file elsewhere (most easily by using the Editor command **Write File** or by choosing **File > Save As** from the context menu).

The LispWorks manuals

The LispWorks manual set also includes the following books:

- The *LispWorks® User Guide and Reference Manual* describes the main language-level features and tools available in LispWorks, along with reference pages.
- The *LispWorks IDE User Guide* describes the LispWorks IDE, the user interface for LispWorks. This is a set of windowing tools that help you to develop and test Common Lisp programs.
- The *Editor User Guide* describes the keyboard commands and programming interface to the LispWorks IDE editor tool.
- The *Foreign Language Interface User Guide and Reference Manual* explains how you can use C source code in applications developed using LispWorks.
- The *Delivery User Guide* describes how you can deliver working, standalone versions of your LispWorks applications for distribution to your customers.
- *Developing Component Software with CORBA®* describes how LispWorks can interoperate with other CORBA-compliant systems.
- The *COM/Automation User Guide and Reference Manual* describes a toolkit for using Microsoft COM and Automation in LispWorks for Windows.
- The *LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual* describes APIs for interfacing to Objective-C and Cocoa in LispWorks for Macintosh.
- The *KnowledgeWorks and Prolog User Guide* describes the LispWorks toolkit for building knowledge-based systems. Prolog is a logic programming system within Common Lisp.
- The *Common Lisp Interface Manager 2.0 User's Guide* describes the portable Lisp-based GUI toolkit.
- The *Release Notes and Installation Guide* which contains notes explaining how to install LispWorks and get it running. It also contains a set of release notes which lists new features and any last minute issues that could not be included in the main manual set.

These books are provided in both HTML and PDF formats, and may also be found at **www.lispworks.com/documentation**.

Commands in the **Help** menu of any of the LispWorks IDE tools give you direct access to your local copy of the HTML format manuals. Details of how to use these commands can be found in the *LispWorks IDE User Guide*.

You can use Adobe® Reader® to browse the PDF documentation. Adobe Reader is available from Adobe's web site, **http://www.adobe.com/**.

Please let us know at **lisp-support@lispworks.com** if you find any mistakes in the LispWorks documentation, or if you have any suggestions for improvements.

1 Introduction to the CAPI

1.1 What is the CAPI?

The CAPI (Common Application Programmer's Interface) is a library for implementing portable window-based application interfaces. It is a conceptually simple, CLOS-based model of interface elements and their interaction. It provides a standard set of these elements and their behaviors, as well as giving you the opportunity to define elements of your own.

The CAPI's model of window-based user interfaces is an abstraction of the concepts that are shared between all contemporary window systems, such that you do not need to consider the details of a particular system. These hidden details are taken care of by a back end library written for that system alone.

An advantage of making this abstraction is that each of the system-specific libraries can be highly specialized, concentrating on getting things right for that particular window system. Furthermore, because the implementation libraries and the CAPI model are completely separate, libraries can be written for new window systems without affecting either the CAPI model or the applications you have written with it.

The CAPI currently runs under X Window System with either GTK+ or Motif, Microsoft Windows and macOS. Using CAPI with Motif is deprecated.

1.2 The CAPI model

The CAPI provides an abstract hierarchy of classes which represent different sorts of window interface elements, along with functions for interacting with them. Instances of these classes represent window objects in an application, with their slots representing different aspects of the object, such as the text on a button, or the items on a menu. These instances are not actual window objects but provide a convenient representation of them for you. When you ask the CAPI to display your object, it creates a real window system object to represent it. This means that if you display a CAPI button, a real Windows button is created for it when running on Microsoft Windows, a real GTK+ button when running on GTK+, and a real Cocoa button when running on Cocoa.

The CAPI's approach makes the production of the screen objects the responsibility of the native window system, so it always produces the correct look and feel. Furthermore, the CAPI's use of the real interface to the window system means that it does not need to be upgraded to account for look and feel changes, and anything written with it is upwardly compatible, just like any well-written application.

1.2.1 CAPI elements

There are five types of elements in the CAPI model: *interface*, *menu*, *pane*, *layout* and *pinboard-object*.

Everything that the CAPI displays is contained within an interface (an instance of the class **`interface`**). When an interface is displayed a window appears containing all the menus and panes you have specified for it. Top level windows in an application are normally defined as an **`interface`** subclass, by using **`define-interface`**.

An interface can contain a number of menus collected together on a menu bar, and context menus can also appear elsewhere. Each menu can contain menu items or other menus (that is, submenus). Items can be grouped together visually and functionally inside *menu components*. Menus, menu items, and menu components are, respectively, instances of the classes **`menu`**, **`menu-item`**, and **`menu-component`**.

Panes are window objects such as buttons and lists. They can be positioned anywhere in an interface. The CAPI provides many different kinds of pane class, among them **push-button**, **list-panel**, **text-input-pane**, **editor-pane**, **tree-view** and **graph-pane**.

The positions of panes are controlled by a layout, which allows objects to be collected together and positioned either regularly (with instances of the classes **column-layout**, **row-layout** or **grid-layout**) or arbitrarily using a **pinboard-layout**. Layouts themselves can be laid out by other layouts — for example, a row of buttons can be laid out above a list by placing both the **row-layout** and the list in a **column-layout**.

pinboard-objects are lightweight elements that you can use to create complex display and user interaction. They must be used inside a **pinboard-layout**.

Note that layouts and interfaces are actually panes too (**interface** and **layout** are subclasses of **simple-pane**), and in most of the cases can be used where panes are used. They are listed separately because of their special role in the layout of windows.

1.3 The history of the CAPI

Window-based applications written with LispWorks 3 and previous used CLX[2], CLUE, and the LispWorks Toolkit. Such applications are restricted to running under X Windows. Because we and our customers wanted a way to write portable window code, we developed a new system for this purpose: the CAPI.

Part of this portability exercise was undertaken before the development of the CAPI, for graphics ports, the generic graphics library. This includes the portable color, font, and image systems in LispWorks. The CAPI is built on top of this technology, and has been implemented for Motif, Microsoft Windows, Cocoa and GTK+.

All Lisp-based environment and application development in LispWorks Ltd now uses the CAPI. We recommend that you use the CAPI for window-based application development in preference to the systems mentioned earlier.

2 Getting Started

This chapter introduces some of the most basic CAPI elements and functions. The intention is simply that you should become familiar with the most useful elements available, before learning how you can use them constructively.

You should work through the examples in this chapter. For extended example code, see:

```
(example-edit-file "capi/elements/")
```

A CAPI application consists of a hierarchy of CAPI objects. CAPI objects are created using **make-instance**, and although they are standard CLOS objects, CAPI slots should generally be accessed using the documented accessors, and not using the CLOS **slot-value** function. You should not rely on **slot-value** because the implementation of the CAPI classes may evolve.

Once an instance of a CAPI object has been created in an interface, it can be displayed on your screen using the function **display**.

2.1 Using the CAPI package

All symbols in this manual are exported from either the CAPI or COMMON-LISP packages unless explicitly stated otherwise. To access CAPI symbols, you could qualify them all explicitly in your code, for example **capi:output-pane**.

However it is more convenient to create a package which has CAPI on its package-use-list:

```
(defpackage "MY-PACKAGE"
  (:add-use-defaults t)
  (:use "CAPI"))
```

This creates a package in which all the CAPI symbols are accessible. To run the examples in this guide, first evaluate:

```
(in-package "MY-PACKAGE")
```

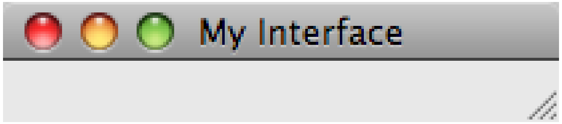
2.2 Creating a window

This section shows how easy it is to create a simple window, and how to include CAPI elements, such as panes, in your window.

1. Enter the following in a listener:

```
(setq interface
      (make-instance 'interface
                     :visible-min-width 200
                     :title "My Interface"))

(display interface)
```

Creating a simple window



A small window appears on your screen, called "My Interface". This is the most simple type of window that can be created with the CAPI.

Note: By default this window has a menu bar with the **Works** menu. The **Works** menu gives you access to a variety of LispWorks tools, just like the **Works** menu of any window in the LispWorks IDE. It is automatically provided by default for any interface you create. You can omit it by passing `:auto-menus nil`.

The usual way to display an instance of a CAPI window is **`display`**. However, another function, **`contain`**, is provided to help you during the course of development.

Notice that the "My Interface" window cannot be made smaller than the minimum width specified. All CAPI geometry values (window size and position) are integers and represent pixel values relative to the topmost/leftmost visible pixel of the primary monitor.

Only a top level CAPI element is shown by **`display`** — that is, an instance of an **`interface`**. To display other CAPI elements (for example, buttons, editor panes, and so on), you must provide information about how they are to be arranged in the window. Such an arrangement is called a *layout* — you will learn more about layouts in **6 Laying Out CAPI Panes**.

On the other hand, **`contain`** automatically provides a default layout for any CAPI element you specify, and subsequently displays it. During development, it can be useful for displaying individual elements of interest on your screen, without having to create an interface for them explicitly. However, **`contain`** is only provided as a development tool, and should not be used for the final implementation of a CAPI element. See **11 Defining Interface Classes - top level windows** on how to display CAPI elements in an interface.

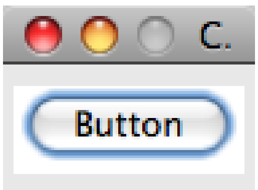
Note that a displayed CAPI element should only be accessed in its own thread. See **4.1 The correct thread for CAPI operations** for more information about this.

This is how you can create and display a button using **`contain`**.

1. Enter the following into a listener:

```
(setq button
      (make-instance 'push-button
                     :data "Button"))

(contain button)
```

Creating a push-button interface



This creates an interface which contains a single push-button, with a label specified by the `:data` keyword. Notice that you could have performed the same example using **`display`**, but you would also have had to create a layout so that the button could have been placed in an interface and displayed.

You can click on the button, and it will respond in the way you would expect (it will depress). However, no code will be run which performs an action associated with the button. How to link code to window items is the topic of the next section.

2.3 Linking code into CAPI elements

Getting a CAPI element to perform an action is done by specifying a *callback*. This is a function which is performed whenever you change the state of a CAPI element. It calls a piece of code whenever a choice is made in a window.

Note that the result of the callback function is ignored, and that its usefulness is in its side-effects.

1. Try the following:

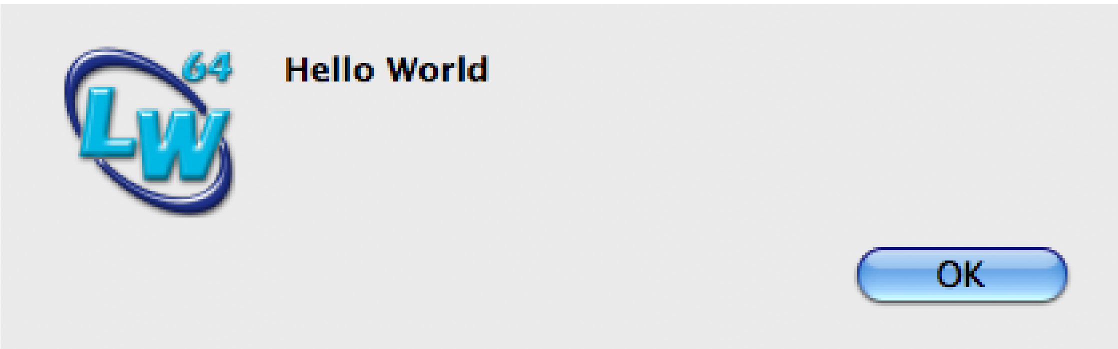
```
(setq push-button
      (make-instance 'push-button
                     :data "Hello"
                     :callback
                     #'(lambda (&rest args)
                         (display-message
                          "Hello World"))))
(contain push-button)
```

   Specifying a callback

2. Click on the **Hello** button.

   A dialog appears containing the message "Hello World".

   A dialog displayed by a callback.



   The CAPI provides the function **`display-message`** to allow you to pop up a dialog sheet containing a message and a Confirm button. This is one of many pre-defined facilities that the CAPI offers. You can also pop up a dialog window rather than a sheet, using **`prompt-with-message`**.

Note: When you develop CAPI applications, your application windows are run in the same Window system event loop as the LispWorks IDE. This - and the fact that in Common Lisp user code exists in the same global namespace as the Common Lisp implementation - means that a CAPI application running in the LispWorks IDE can modify the same values as you can concurrently modify from one of the the LispWorks IDE programming tools.

For example, your CAPI application might have a button that, when pressed, sets a slot in a particular object that you could also set by hand in the Listener. Such introspection can be useful but can also lead to unexpected values and behavior while testing your application code.

3 General Properties of CAPI Panes

This chapter contains information that does not belong in the more specific sections that follow, including functionality common to several (or most) pane classes. It also introduces classes allowing you to create more common windowing elements, beyond the few mentioned in **2 Getting Started**.

Before trying out the examples in this chapter, define the functions **`test-callback`** and **`hello`** in your Listener. The first displays the list of arguments it is given, and returns **`nil`**. The second just displays a message.

```
(defun test-callback (data interface)
  (display-message "Data ~S in interface ~S"
                   data interface))


(defun hello (data interface)
  (declare (ignore data interface))
  (display-message "Hello World"))
```

We will use these callbacks in the examples that follow.

3.1 Generic properties

Because CAPI elements are just like CLOS classes, many elements share a common set of properties. The remainder of this section describes the properties that all the classes described in this chapter inherit.

3.1.1 Scroll bars

The CAPI lets you specify horizontal or vertical scroll bars for any subclass of the **`simple-pane`** element (including all of the classes described in this chapter).

Horizontal and vertical scroll bars can be specified using the keywords **`:horizontal-scroll`** and **`:vertical-scroll`**. By default, both **`:vertical-scroll`** and **`:horizontal-scroll`** are **`nil`**.

3.1.2 Background and foreground colors

All subclasses of the simple pane element can have different foreground and background colors, using the **`:background`** and **`:foreground`** initargs of **`simple-pane`**. For example, including:

```
:background :blue
:foreground :yellow
```

in the **`make-instance`** of a text pane would result in a pane with a blue background and yellow text.

3.1.3 Fonts

The CAPI interface supports the use of other fonts for text in title panes and other CAPI objects, such as buttons, through the use of the **`:font`** initarg of **`simple-pane`**.. If the CAPI cannot find the specified font it reverts to the default font. The **`:font`** keyword applies to data following the **`:text`** keyword. The value is a graphics ports **font-description** object specifying various attributes of the font.

On systems running X Windows, the **xlsfonts** command can be used to list which fonts are available. The X logical font descriptor can be explicitly passed as a string to the **:font** initarg, which will convert them.

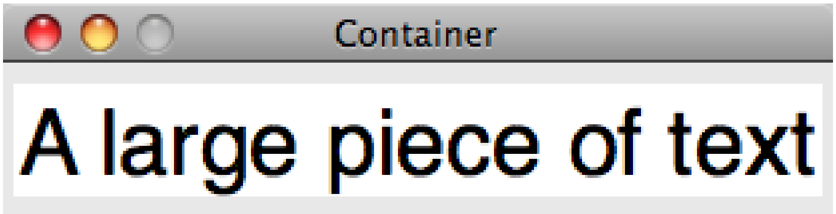
Here is an example of a **title-pane** with an explicit font:

```
(contain
 (make-instance 'title-pane
                :text "A title pane"
                :font (gp:make-font-description
                       :family "Times"
                       :size 12
                       :weight :medium
                       :slant :roman)))
```

Here is an example of using **:font** to produce a title pane with larger lettering. Note that the CAPI automatically resized the pane to fit around the text.

```
(contain
 (make-instance 'title-pane
                :text "A large piece of text"
                :font (gp:make-font-description
                       :family "Times"
                       :size 34
                       :weight :medium
                       :slant :roman)))
```

An example of the use of font descriptions



3.1.4 Mnemonics

This section applies to Microsoft Windows and GTK+ only.

Underlined letters in menus, titles and buttons are called mnemonics. The user can select the element by pressing the corresponding key.

3.1.4.1 Controlling Mnemonics

For individual buttons, menus, menu items and title panes, you can use the **:mnemonic** initarg to control them. For example:

```
(capi:contain (make-instance 'capi:push-button
                             :data "FooBar"
                             :mnemonic #\B))
```

For more information on mnemonics in buttons, see **3.10.4 Mnemonics in buttons**.

For information on controlling mnemonics in button panels, see **5.2.4 Mnemonics in button panels**. For information on controlling mnemonics in menus, see **8.6 Mnemonics in menus**.

The initarg **:mnemonic-title** allows you to specify the mnemonic in the title for many pane classes including **list-panel**, **text-input-pane** and **option-pane**. Also **grid-layout** supports *mnemonic-title* when *has-title-column-p* is true. For the details see **titled-object**.

3.1.4.2 Mnemonics on Microsoft Windows

On Microsoft Windows the user can make the mnemonics visible by holding down the **`Alt`** key.

Windows can hide mnemonics when the user is not using the keyboard. This is controlled in Windows 8 by:

Control Panel > Ease of Access > Ease of Access Center > Make the keyboard easier to use > Underline keyboard shortcuts and access keys

and in Windows XP by:

Control Panel > Display > Appearance > Effects > Hide underlined letters...

3.1.5 Focus

The *focus* is where keyboard gestures are sent.

You can specify that a pane should or should not get the focus by using the initarg **`:accepts-focus-p`** (defined for **`element`**). By default interactive elements except menus accept focus, and non-interactive elements do not accept focus, so normally you do not need to use **`:accepts-focus-p`**.

3.1.5.1 Initial focus

By default, when a window first appears the focus is in the top-left pane that accepts focus. You can override this by using the initarg **`:initial-focus`** or using the accessor **`pane-initial-focus`** on interfaces and layouts, and using the initarg **`:initial-focus-item`** for choices (**`check-button-panel`** for example).

3.1.5.2 Querying the focus

The function **`pane-descendant-child-with-focus`** can find a child pane that has the focus, when given as argument a pane with children such as a **`layout`**, an **`interface`**, or certain choices including a **`button-panel`** and **`toolbar`**.

The function **`pane-has-focus-p`** can be used to determine if a specific pane has the focus.

3.1.5.3 Setting the focus dynamically

The function **`set-pane-focus`** can be used to set the focus to a pane inside an active window. If you need to ensure that the window is active, you can use **`activate-pane`**, which activates the window and sets the focus. For panes that have children (as described in **3.1.5.2 Querying the focus**) the actual pane that receives the focus is the "initial focus", as described **3.1.5.1 Initial focus**.

When **`set-pane-focus`** is called, just before it actually sets the focus, it calls the generic function **`pane-got-focus`** with the interface and the pane. You can define your own method (specialized on your own interface class) to perform any processing that may be required.

3.1.6 Mouse cursor

The mouse cursor of a pane can be specified by the initarg **`:cursor`** or accessor **`simple-pane-cursor`**. The cursor to be used needs to be a result of a call to **`load-cursor`**.

It is possible to set an "override" cursor in an interface, which sets the cursors in all its panes. That is typically used to temporarily set the cursor while the interface is in a different input state from the normal state. This feature does not work on Cocoa.

3.2 Base classes

Most CAPI classes inherit from **`capi-object`**, which has a *plist* and a *name*. The subclasses of **`capi-object`** are:

`element` The class of all elements that corresponding to an underlying window system element. **`element`** defines geometry functionality including geometry hints (see **6.4 Specifying geometry hints**), and a few other basic properties. Note however that not all subclasses of **`element`** correspond to an underlying element: some of them are a composition of several elements, and some of them are layout elements.

Subclasses of **`element`** are **`menu`** for menus (chapter 8), and **`simple-pane`** for all other display elements. The subclasses contain **`layout`** (**6 Laying Out CAPI Panes**), which is used to arrange CAPI elements, and **`interface`** (**11 Defining Interface Classes - top level windows**), which represents a window, and classes that correspond to specific display elements like **`button`** (**3.10 Button elements**).

`callbacks` A mixin class for active elements that need to respond to user input, defining various callbacks (**3.4 Callbacks**). **`item`**, **`collection`** and **`menu-object`** (parent of **`menu`** and **`menu-component`**) inherit from **`callbacks`**.

`item` A mixin class for elements that have a single piece of text like **`menu-item`** and **`button`**. It can also be used as a way of making individual items in collections/choices (**5 Choices - panes with items**) have their own callbacks and properties. **`item`** inherits from **`callbacks`**.

`pinboard-object` The superclass of pinboard objects, are lightweight graphical objects which are displayed inside **`pinboard-layout`** (**12.3 Creating graphical objects**).

`collection` and subclass **`choice`**

Choice is the mixin class for all elements that have items (**5 Choices - panes with items**). **`collection`** (and hence **`choice`**) inherits from **`callbacks`**. The subclasses of **`choice`** that can be displayed inherit from **`simple-pane`** too.

3.3 Specifying titles

It is possible to specify a title for a window, or part of a window. Several of the examples that you have already seen have used titles. There are two ways that you can create titles:

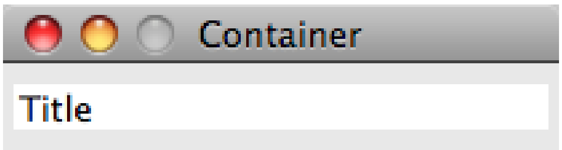
- Use the **`title-pane`** class.
- Specify a title directly to any subclass of **`titled-object`**.

3.3.1 Title panes

A **`title-pane`** is a blank pane into which text can be placed in order to form a title.

```
(setq title (make-instance 'title-pane
                           :visible-min-width 200
                           :text "Title"))

(contain title)
```

A title pane



3.3.2 Specifying titles directly

You can specify a title directly to all CAPI panes, using the **`:title`** keyword. This is much easier than using title-panes, since it does not necessitate using a layout to group two elements together.

Any class that is a subclass of **`titled-object`** supports the **`:title`** keyword. All of the standard CAPI panes inherit from this class. You can find all the subclasses of **`titled-object`** by using the Class Browser tool in the LispWorks IDE.

3.3.2.1 Window titles

Specify a title for a CAPI window by supplying the **`:title`** initarg for the **`interface`**, and access it with **`interface-title`**.

Further control over the title of your application windows can be achieved by using **`set-default-interface-prefix-suffix`** and/or specializing **`interface-extend-title`** as illustrated in **11.5.2 Controlling the interface title**.

You can call **`interface-display-title`** to get the string that is actually displayed (or would be displayed if the interface was displayed).

3.3.2.2 Titles for elements

The position of any title can be specified by using the **`:title-position`** keyword. Most panes default their *title-position* to **`:top`**, although some use **`:left`**.

You can place the title in a frame (like a groupbox) around its element by specifying **`:title-position :frame`**.

You may specify the font used in the title via the keyword **`:title-font`**.

The title of a **`titled-object`**, and its font, may be changed interactively with the use of **`setf`**, if you wish.

1. Create a push button by evaluating the code below:

```
(setq button (make-instance 'push-button
                            :text "Hello"
                            :title "Press: "
                            :title-position :left
                            :callback 'hello))

(contain button)
```

2. Now evaluate the following:

```
(apply-in-pane-process
 button #'(setf titled-object-title) "Press here: " button)
```

   As soon as the form is evaluated, the title of the pane you just created changes.

3. Lastly evaluate the following:

```
(apply-in-pane-process
 button #'(setf titled-object-title-font)
 (gp:merge-font-descriptions
  (gp:make-font-description :size 42)
  (gp:convert-to-font-description
   button
   (titled-object-title-font button))) button)
```

Notice how the window automatically resizes in steps 2 and 3, to make allowance for the new size of the title.

3.4 Callbacks

The class **`callbacks`** is the superclass of all the CAPI objects that receive callback calls in response to user gestures, excluding output panes. This includes collections and choices, buttons, menus, menu components, menu items and **`item-pinboard-object`**. The actual interaction depends on the specific class.

The arguments that callbacks are called with can be specified by the initarg **`:callback-type`**. When the argument contain the interface, the actual interface can be specified to be another interface by using **`attach-interface-for-callback`**. The function **`element-interface-for-callback`** can be used to find which interface is going to be used in a callback.

Callbacks can be aborted using **`abort-callback`**.

There is more detail about the callbacks available in choices in **5.10.3 Callbacks in choices**.

Note: **`output-pane`** and its subclasses implement callback calls by the *input-model* mechanism.

3.5 Displaying and entering text

There are a variety of ways in which an application can display text, accept text input or allow editing of text by the user:

Display panes	Show non-editable text.
Text input panes	Used for entering short pieces of text.
Editor panes	Used for dealing with large amounts of text such as files. Also offer full configurable editor functionality.
Rich text panes	Support formatted text. Available on Cocoa and Microsoft Windows only.

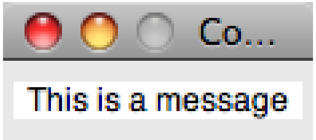
3.5.1 Display panes

You can use a **`display-pane`** to display text messages on the screen. The text in these messages cannot be edited, so they can be used by the application to present a message to the user. The **`:text`** initarg can be used to specify the message that is to appear in the pane.

1. Create a display pane by evaluating the code below:

```
(setq display (make-instance 'display-pane
                    :text "This is a message"))

(contain display)
```

A display pane



Note that the window title, which defaults to "Container" for windows created by **contain**, may appear truncated.

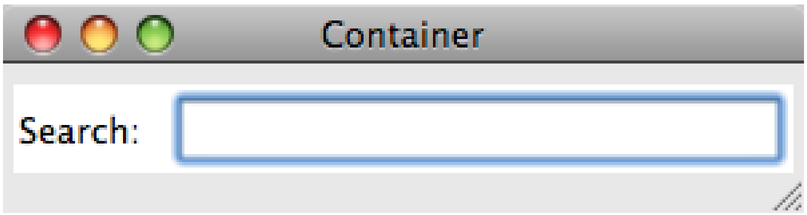
You can access the text (get and set) of a **display-pane** by the accessor **display-pane-text**. You can access the selection by **display-pane-selection-p**, **display-pane-selection**, **set-display-pane-selection** and **display-pane-selected-text**.

3.5.2 Text input panes

When you want the user to enter a line of text, such as a search string, use a **text-input-pane**.

```
(setq text (make-instance 'text-input-pane
                          :title "Search: "
                          :callback 'test-callback))

(contain text)
```

A text input pane



Notice that the default title position for text input panes is **:left**.

You can place text programmatically in the text input pane by supplying a string for the **:text** initarg, or later by calling **(setf text-input-pane-text)** in the appropriate process.

You can use **set-text-input-pane-selection** to control the selection in the text input pane:

```
(setq tip (make-instance 'capi:text-input-pane
                         :title "Search: "
                         :text "Foo Bar Baz"))

(capi:set-text-input-pane-selection
 tip
 (length "Foo ")
 (+ (length "Foo ") (length "Bar")))

(capi:contain tip)
```

text-input-pane has many callbacks which allow the program to perform various tasks as the user changes the text, the selection or the caret position, or enters/leaves the pane. It is possible to respond to specific keyboard gestures, characters or otherwise (like **Up** arrow). **text-input-pane** has also options for performing completion on the user input.

You can add toolbar buttons for easier user input in a **text-input-pane** via the **:buttons** initarg. This example allows the user to enter the filename of an existing Lisp source file, either directly or by selecting the file in a dialog raised by the **Browse File** button. There is also a **Cancel** button, but the default **OK** button is not displayed:

```
(capi:contain
 (make-instance
  'capi:text-input-pane
  :buttons
  (list :cancel t
        :ok nil
        :browse-file
        (list :operation :open
              :filter "*.LISP;*.LSP"))))
```

For a larger quantity of text use **multi-line-text-input-pane**.

On Cocoa, **text-input-pane** can also be made to look like a search field, using the initarg **:search-field** and related initargs.

For entering passwords use the subclass **password-pane**, which does not display the actual characters that the user types.

3.5.3 Editor panes

An **editor-pane** is a pane which displays text and allows the user to edit it. The text is held and manipulated in a separate module, the Editor, which is implemented in the "EDITOR" package.

The Editor is optimized to deal with large amounts of text, whether that is because a single document contains large amount of text or because the user wants to edit many texts at the same time. It has a large set of commands that the user can invoke to perform a variety of tasks, including many kinds of editing and search operations, integration with the LispWorks IDE, and various other tasks. It also has a programmatic interface to manipulate the text, which is exported from the package "EDITOR". The user interface and the programmatic interface are both documented in the *Editor User Guide*, and the LispWorks IDE uses **editor-pane** for editing.

The interaction of the Editor emulates either Emacs style or the native style of macOS, Microsoft Windows or KDE/Gnome as appropriate. There is a global default setting (native on Windows, Emacs elsewhere), which can be set in a runtime image by the Delivery keyword **:editor-style**. In particular, you fix the style for **editor-pane** in your interfaces by defining your method for **interface-keys-style**. See the chapter "Emulation" in the *Editor User Guide* for more detail about the different styles.

From the CAPI side you can access the editor structures that hold the text by using **editor-pane-buffer**, which returns an **editor:buffer** object which holds the text. You can then use the programmatic Editor interface to access and manipulate the text.

For example, the following code inserts the string **"foo"** in the end of the editor pane (really in the end of the buffer):

```
(let ((buffer (capi:editor-pane-buffer editor-pane)))
   (let ((point (editor:buffers-end buffer)))
      (editor:insert-string point "foo")))
```

Above, **point** is an **editor:point** object.

Alternatively, editor commands can be executed by passing the name of an editor command to **call-editor**.

Note that the editor objects can be accessed from any process (as opposed to the CAPI elements), because they use locks. Programmers can use the locks to group several editor operations so that they happen "atomically".

It is possible to specify that an **editor-pane** has an attached Echo Area which is where non-editing interactions (for example entering a command name or filename) occur. To add an Echo Area, use the **:echo-area** initarg. Otherwise, a special window pops up when such interaction needs to occur.

The variables ***editor-cursor-active-style***, ***editor-cursor-color***, ***editor-cursor-drag-style*** and ***editor-cursor-inactive-style*** can be used to control the appearance of the cursor. When adding an echo area, the inactive cursor style can be controlled separately by **`*editor-cursor-inactive-style*`**.

An **`editor-pane`** can have input callbacks (before and after) and a change callback. These are described in **3.5.3.1 Editor pane callbacks**.

On the CAPI side there are few additional functions that can be used on an **`editor-pane`**. These are described in **3.5.3.2 Additional editor-pane functions**.

3.5.3.1 Editor pane callbacks

You can use the initarg **`:change-callback`** to specify a function which is called whenever the editor buffer under the **`editor-pane`** changes. The value *change-callback* can be set either by:

```
(make-instance 'capi:editor-pane :change-callback ...)
```

or:

```
(setf capi:editor-pane-change-callback)
```

The current value can be queried by the accessor **`editor-pane-change-callback`**.

The *change-callback* function must have signature:

`change-callback` *pane point old-length new-length*

pane is the **`editor-pane`** itself.

point is an **`editor:point`** object where the modification to the underlying buffer starts. *point* is a temporary point, and is not valid outside the scope of the change callback. For more information about **`editor:point`** objects, see "Points" in the *Editor User Guide*.

old-length is the length of the affected text following *point*, prior to the modification.

new-length is the length of the affected text following *point*, after the modification has occurred.

Typical calls to the *change-callback* occur on insertion of text (when *old-length* is 0) and on deletion of text (when *new-length* is 0). There can be other combinations, for example, after executing the **Uppercase Region** editor command, *change-callback* be called with both *old-length* and *new-length* being the length of the region. The same is true for changing editor text properties.

The *change-callback* is always executed in the process of *pane* (as if by **`apply-in-pane-process`**).

The *change-callback* is permitted to modify the buffer of *pane*, and other editor buffers. The callback is disabled inside the dynamic scope of the call, so there are no recursive calls to the *change-callback* of *pane*. However, changes done by the callback may trigger *change-callback* calls on other **`editor-pane`**s, whether in the same process or in another process.

There is an example illustrating the use of *change-callback* in:

```
(example-edit-file "capi/editor/change-callback")
```

You can use the initargs **`:before-input-callback`** and **`:after-input-callback`** to add input callbacks which are called when **`call-editor`** is called. Note that the default *input-model* also generates calls to **`call-editor`**, so unless you override the default *input-model* these input callbacks are called for all keyboard and mouse gestures (other than gestures that are processed by a non-focus completer window).

In both cases (*before-input-callback* and *after-input-callback*) the argument is a function that takes two arguments: the editor pane itself and the input gesture (the second argument to **`call-editor`**).

call-editor may redirect gestures to another pane. For example, gestures to an **editor-pane** are redirected to the echo area while it is used. In this case *before-input-callback* is called more than once for the same gesture, but *after-input-callback* is called only once for each gesture, on the pane that actually processed the gesture.

3.5.3.2 Additional editor-pane functions

The contents of the buffer can be retrieved and set by the accessor **editor-pane-text**.

modify-editor-pane-buffer can be used to change the text and the filling at the same time.

editor-pane-line-wrap-marker, **editor-pane-line-wrap-face** and ***default-editor-pane-line-wrap-marker*** control the appearance of the marker that indicates wrapping of lines that are too long.

The function **editor-pane-selected-text** returns the selected text (if any), and **editor-pane-selected-text-p** checks if there is a selection.

You can call **set-default-editor-pane-blink-rate** to set the default blink rate of the cursor on all editor panes. You can specialize **editor-pane-blink-rate** to control the blink rate of specific panes, and use **editor-pane-native-blink-rate** to query the blink rate of the underlying GUI system. Note that the underlying system will normally allow the user to change this value.

The function **print-editor-buffer** can be used to print the contents of the editor buffer.

The function **set-editor-parenthesis-colors** can be used to control parenthesis coloring in Lisp mode.

Editor panes support composition of characters using input methods (see *composition-callback* in **output-pane**) by having a default callback **editor-pane-default-composition-callback**, which handles it mostly right. You can specify your own callback, which can also call **editor-pane-default-composition-callback** to do the actual work.

The **editor-pane** is geared towards editing files, and in particular it tries to guard against loss of work by keeping backup files and auto-save files, and asking the user before closing an unsaved buffer. When you use an **editor-pane** for other purposes, and therefore do not need all of this functionality, you should use temporary buffers. Create a temporary buffer by supplying the initarg **:buffer-name :temp**, or create your own temporary buffer explicitly by **(editor:make-buffer ... :temporary t)**.

You can make an **editor-pane** be non-editable by users by supplying the initarg **:enabled :read-only**, or completely disable it with **:enabled nil**.

3.6 Displaying rich text

On Microsoft Windows and Cocoa, **rich-text-pane** allows you to display and edit rich text. It supports character attributes such as font, size and color, and paragraph attributes such as alignment and tab-stops.

See this example:

```
(example-edit-file "capi/applications/rich-text-editor")
```

3.7 Hierarchy of panes

Every element that is displayed has a parent, which you can find by the **element** accessor **element-parent**. The ultimate ancestor is a **screen**, which you can find by **element-screen**. The element is inside some window which is associated with a CAPI interface instance (that is, an instance of subclass of **interface**) which is called the "top level interface" and can be found by by **top-level-interface**. Note that inside MDI on Microsoft Windows the top level interface is the one inside the MDI, rather than the enclosing MDI window. You can test whether an object is a top level interface by

top-level-interface-p. The function **element-container** returns the parent of the top level interface, that is the screen outside the MDI, but the **document-frame** inside the MDI.

Some elements have children. You can operate on the children of an element by using **map-pane-children** or **map-pane-descendant-children**. These functions will work on any element, and they will do nothing for elements without children.

The implementation of the panes you specify may internally involve generating more panes, and **element-parent**, **map-pane-children** and **map-pane-descendant-children** will find these. Thus when using these functions you cannot assume that you know the hierarchy, and you need to check if the pane that you got is the right one. For example, if you create a layout like this:

```
(setq layout
      (make-instance 'capi:row-layout
                     :description
(list (make-instance 'capi:list-panel))))
```

then doing something like:

```
(capi:map-pane-children layout
        #'(lambda (pane) (setf (capi:collection-items pane) nil)
```

may not work, because the list panel may not be a direct child of the layout. In most cases it is best to record the actual panes so you know where to access them (most commonly in a slot in the interface). Alternatively you can use **map-pane-descendant-children** with a *function* that checks each child pane before operating on it.

Note that all these functions give useful results only for displayed elements.

3.8 Accessing pane geometry

The functions **simple-pane-visible-height**, **simple-pane-visible-width**, and **simple-pane-visible-size** can be used to read the visible geometry of a pane. Other geometrical properties of a pane can be accessed by **with-geometry**, which binds variables to the various geometrical properties of the pane.

3.9 Special kinds of windows

3.9.1 Browser pane

On Microsoft Windows and Cocoa, **browser-pane** implements embedding of a basic web browser. It allows you to display HTML, navigate, refresh, handle errors, redirect to another URL, and so on.

3.9.2 OLE embedding and control

On Microsoft Windows **ole-control-pane** implements embedding of OLE control components. You can also embed CAPI windows inside other applications using **ole-control-component**. You define an OLE control component (an Automation class that implements OLE Control protocols) using **define-ole-control-component**, and other (non-LispWorks) applications can use it.

3.9.3 Cocoa views and application interfaces

On Cocoa, you can use **cocoa-view-pane** to display an arbitrary Cocoa View. You can specify the name of the Cocoa view class to create, and a function that is called to initialize it. The function **cocoa-view-pane-view** can be used to access the

Cocoa view after it has been created.

The class **cocoa-default-application-interface** is a special class for defining application interfaces, which gives you control of application-wide properties which are not associated with specific windows. This includes the Application menu and default menu bar items, Dock context menu, application message processing and display state of the whole application.

3.9.4 Slider, Progress bar and Scroll bar

The classes **slider** and **scroll-bar** implement panes that show the value of some quantity and allow the user to change it interactively.

slider is intended to be used in general for any pseudo-continuous quantity that the user should be able to manipulate.

scroll-bar is intended to be used for scrolling. Normally a scroll bar is specified simply by supplying the **:vertical-scroll** or **:horizontal-scroll** initarg when making the pane that needs scrolling, but in some circumstances an explicit scroll bar may be useful.

The class **progress-bar** implements a pane that shows the value of some quantity and is used to indicate progress in performing some task.

All of these classes inherit from **range-pane**, which defines the various values that are used and the orientation. In addition to the **range-pane** accessors, there is also the function **range-set-sizes** which you can use to set several values at the same time.

3.9.5 Text input range

text-input-range is a special pane for entering numeric values, allowing the user to either type the number or use buttons to adjust the value.

3.9.6 Stream panes

There are three subclasses of **editor-pane** which handle Common Lisp streams.

3.9.6.1 Collector panes

A **collector-pane** displays anything printed to the stream associated with it. Background output windows, for instance, are examples of collector panes.

```
(setq collector
      (make-instance 'collector-pane
                     :title "Example collector pane:"))

(contain collector)

(princ "abc" (collector-pane-stream collector))
```

The **collector-pane** has a mechanism to temporarily make it the child of a parent **switchable-layout**, so the user can see the output printed into it. The functions **map-typeout** and **unmap-typeout** do the switch, and the macro **with-random-typeout** can be used to do both switches and to also bind a variable to the stream of the **collector-pane**. This mechanism is used in the LispWorks IDE to show the output of **Compile Buffer** and other operations.

3.9.6.2 Interactive panes

An **interactive-pane** is the building block on which **listener-pane** is built.

```
(contain (make-instance 'interactive-pane
              :title "Interactive pane"))
```

You can simulate user input into an **interactive-pane** by **interactive-pane-execute-command**.

Note: **interactive-pane** is probably too difficult to use, due to the complexities involved with the interaction with the Editor. However, for its subclass **listener-pane**, the system deals with all these issues.

3.9.6.3 Listener panes

The **listener-pane** class is a subclass of **interactive-pane**, and allows you to create interactive Common Lisp sessions. You may occasionally want to include a listener pane in a tool (as, for instance, in the LispWorks IDE Debugger).

```
(contain (make-instance 'listener-pane
                        :title "Listener"))
```

The **listener-pane** activity would normally be interacting with the user, but you can also emulate user interaction using **listener-pane-insert-value**. Note also that since **listener-pane** is a subclass of **editor-pane**, you can use the full power of the Editor on it.

3.9.7 Shell pane

shell-pane is a pane that runs a sub-process ("shell", "console") and allows the user to interact with it.

3.10 Button elements

Button classes inherit from the class **button**, which defines most of the attributes of buttons. **button** inherits from **simple-pane** and **item**. Button panels can be created, and are described in **5 Choices - panes with items**.

There are three classes of buttons:

push-button	Never selected, just invokes the callback when clicked.
check-button	Toggles between selected and unselected each time it is clicked.
radio-button	When clicked is selected, and deselects all other buttons in the same panel.

A single **radio-button** does not really make sense and this class will normally be used only inside **radio-button-panel**. **check-button** and **push-button** are used both inside **check-button-panel** or **push-button-panel** and on their own. Note that when using a panel, you do not have to actually use **button** objects, because the panel generates them automatically, and most of the functionality of **button**s can be specified in the **button-panel**.

The text and the data that are associated with a button are defined by the the initargs and accessor inherited from **item**: **:data**, **:text**, **:print-function**, **item-data**, **item-text**, **item-print-function**. The function **print-capi-button** can be used to find what string is displayed (or will be displayed) for a button.

The callbacks of **button** are inherited from **callbacks** (via **item**). The **:selection-callback** (the initarg **:callback** can be used too) is the main callback, and **:retract-callback** is called for deselection.

button has various initargs and accessors controlling which image(s) to display, whether it is selected and/or enabled, and whether it is a **Cancel** button or the default button.

3.10.1 Push buttons

The **`:enabled`** keyword can be used to specify whether or not the button should be selectable when it is displayed. This can be useful for disabling a button in certain situations.

The following code creates a **`push-button`** which cannot be selected.

```
(setq offbutton (make-instance 'push-button
                  :data "Button"
                  :enabled nil))

(contain offbutton)
```

These **`setf`** expansions enable and disable the button:

```
(apply-in-pane-process
 offbutton #'(setf button-enabled) t offbutton)

(apply-in-pane-process
 offbutton #'(setf button-enabled) nil offbutton)
```

All subclasses of the **`button`** class can be disabled in this way.

3.10.2 Check buttons

Check buttons can be produced with the **`check-button`** element.

1. Enter the following in a Listener:

```
(setq check (make-instance 'check-button
                      :selection-callback 'hello
                      :retract-callback 'test-callback
                      :text "Button"))

(contain check)
```

A check button

Notice the use of **`:retract-callback`** in the example above, to specify a callback when the element is deselected.

Like push buttons, check buttons can be disabled by specifying **`:enabled nil`**.

3.10.3 Radio buttons

Radio buttons can be created explicitly with the **`radio-button`** element, but they are usually part of a button panel as described in **5 Choices - panes with items**. The **`:selected`** initarg is used to specify whether or not the button is selected, and the **`:text`** initarg can be used to label the button.

```
(contain (make-instance 'radio-button
```

```
                              :text "Radio Button"
                              :selected t))
```

An explicitly created radio button

Although a single radio button is of limited use, having an explicit radio button class gives you greater flexibility, since associated radio buttons need not be physically grouped together. Generally, the easiest way of creating a group of radio buttons is by using a button panel, but doing so means that they will be geometrically, as well as semantically, connected.

3.10.4 Mnemonics in buttons

This section applies to Microsoft Windows and GTK+ only.

The initarg **`:mnemonic`** allows you to specify a mnemonic for a button.

Alternatively you can specify the button text and its mnemonic together with the initarg **`:mnemonic-text`**, for example:

```
(contain
 (make-instance 'radio-button
                :mnemonic-text
                "Radio Button with a &Mnemonic"))
```

For all the details see **`button`**.

3.11 Adding a toolbar to an interface

A top level interface can have a toolbar, which is typically displayed at the top of the window and follows platform-standard behavior. On Cocoa, this will be a standard foldable toolbar.

For the details see **9 Adding Toolbars**.

3.12 Tooltips

A tooltip is a temporary window containing text which appears when the user positions the cursor over an element for a period. The appearance is slightly delayed and the text is usually short.

Tooltips are often used for brief help text and identification of GUI elements. For example the "X" button alongside the Filter area in the Process Browser tool in the LispWorks IDE has a tooltip "Clear filter". Tooltips can also be used to complete the display of partially hidden text, for example in the Debugger tool Backtrace view where the display of long variable values might be truncated.

You can implement tooltips for **`output-pane`**s, **`collection`**s, **`element`**s, **`menu-item`**s and **`toolbar-button`**s.

3.12.1 Tooltips for output panes

To implement tooltips in an **`output-pane`**, call **`display-tooltip`** via a **`:motion`** gesture in the pane's *input-model*. The tooltip text might depend on the cursor position or, in the case of a **`pinboard-layout`**, on the pinboard object under the cursor.

See this example:

```
(example-edit-file "capi/graphics/pinboard-help")
```

3.12.2 Tooltips for collections, elements and menu items

Supply the **`:help-callback`** initarg in an **`interface`**, along with a suitable **`:help-key`** initarg for each of its collections, elements and menu-items that should have a tooltip. *help-callback* should return a suitable string (which will be the tooltip text) when passed *type* **`:tooltip`** and the *help-key*.

See the manual page for **`interface`** for an example of a tooltip on a **`text-input-pane`**.

3.12.3 Tooltips for toolbar buttons

You can implement tooltips for a **`toolbar-button`** exactly as for collections and so on as described in **3.12.2 Tooltips for collections, elements and menu items**. See the example in **9.5 Specifying tooltips for toolbar buttons**.

However, if your **`toolbar-button`**s are grouped in a **`toolbar-component`** it is simpler to supply the **`:tooltips`** initarg. *tooltips* should be a list containing a string giving the tooltip text of each button in the component. See this example:

```
(example-edit-file "capi/applications/simple-symbol-browser")
```

3.13 Screens

A screen object (of class **`screen`** or a subclass) represents what CAPI thinks is the screen that the user sees. In principle it can be a **`mono-screen`**, but these days it is always **`color-screen`**. **`screen`** is subclass of **`capi-object`**, but not **`simple-pane`**.

You get a screen object by one of:

- Calling **`convert-to-screen`**.
- Calling **`element-screen`** on a displayed element.
- Calling **`screens`**.

`convert-to-screen` can take screen specification in various forms. On X GUI systems (GTK+ and Motif) this can be used to select which display to use. On Microsoft Windows on any pane that is displayed inside MDI returns the MDI **`document-container`**, but otherwise there is only one screen. On Cocoa there is always only one screen. **`convert-to-screen`** initializes the screen if needed.

From a displayed element you can find the screen by **`element-screen`**. Note that this returns the actual screen, even for a pane inside MDI.

The function **`screens`** returns a list of the currently active screens. This list is always of length 1 on Cocoa and Microsoft Windows, not including MDI.

A screen specification that **`convert-to-screen`** accepts can also be used to specify the screen on which to display an interface in a call to **`display`**.

You can find the geometry of the screen by the readers **`screen-width`** and **`screen-height`**, and its depth by **`screen-depth`**. Some physical properties can be found by the readers **`screen-width-in-millimeters`**, **`screen-height-in-millimeters`** and the function **`screen-logical-resolution`**. **`screen-number`** returns the screen number for X11 interface (GTK+ and Motif).

The area that is actually used for display may be restricted by some parts of the screen being dedicated to global features, for example menubar on Cocoa. The area that can be used for displaying by the application is called "internal geometry", which can be found by **`screen-internal-geometry`**.

A screen may correspond to several monitors. In this case it has a "virtual geometry", which is a rectangle containing all the physical screens, which can be found by **`virtual-screen-geometry`**. The coordinates of top-level windows are with respect to this rectangle. With multiple screens, **`screen-internal-geometry`** returns the internal geometry of the first (main) monitor. You can use **`screen-internal-geometries`** to find the internal geometries of all the monitors, and **`screen-monitor-geometries`** to find all the full geometries. You can use **`pane-screen-internal-geometry`** to find the internal geometry of the monitor on which the pane is displayed.

On the X interface the screen "dies" when the X connection gets broken for whatever reason. You can check for that by calling **`screen-active-p`**, which returns true for "live" screens and false otherwise.

You can find the CAPI interfaces that are displayed on a specific screen by **`screen-interfaces`**, and the active interface (as far as CAPI is concerned) by calling **`screen-active-interface`**. Note that this interface may be obscured by windows of another application.

On Microsoft Windows using MDI, the CAPI interface are children of a **`document-container`**, which is a "screen-like" object. In particular, it can be used as the screen argument of **`display`**, the internal geometry functions return the correct values, and **`screen-interfaces`** returns the interfaces.

4 General Considerations

This chapter describes general issues relating to the use of CAPI. Subsequent chapters address issues specific to the host window system, and then the use of particular CAPI elements.

4.1 The correct thread for CAPI operations

All operations on displayed CAPI elements need to be in the thread (that is, the **`mp:process`**) that runs their interface. On some platforms, **`display`** and **`contain`** make a new thread. On Cocoa, all interfaces run in a single thread.

Specifying an owner (using the keyword **`:owner`**) in a dialog, for example by calling **`display-dialog`** or **`popup-confirmer`**, is also "an operation" on the owner. See **10.4 Dialog Owners** for discussion of dialog owners.

In most cases this issue does not arise, because CAPI callbacks are run in the correct thread. However, if your code needs to communicate with a CAPI window from a random thread, it should use **`execute-with-interface`**, **`execute-with-interface-if-alive`**, **`apply-in-pane-process`** or **`apply-in-pane-process-if-alive`** to send the function to the correct thread.

This is why the brief interactive examples in this manual generally use **`execute-with-interface`** or **`apply-in-pane-process`** when modifying a displayed CAPI element. In contrast, the demo example in **11.4 Connecting an interface to an application** is modified only by callbacks which run in the demo interface's own thread, and so there is no need to use **`execute-with-interface`** or **`apply-in-pane-process`**.

Threads started by CAPI process events in the "standard" way, that is they call **`mp:general-handle-event`** on objects that are sent to them by **`mp:process-send`**. In particular, if you want to "schedule" an event to happen in the current after the current callback returns, you can use **`mp:current-process-send`**. For example, if the *display-callback* of an **`output-pane`** sometimes needs to start another interface, it would be a bad idea to do this inside the *display-callback*, so instead of:

```
(capi:display new-interface)
```

you can use:

```
(mp:current-process-send `(capi:display ,new-interface))
```

which will cause it to happen later.

On systems other than Cocoa, when you run something that is lengthy inside a CAPI process, you can process events in a similar way to the way CAPI processes them by calling **`process-pending-messages`**, which processes all pending events and returns. However that may not always work well, because the processing of the event can do arbitrary things, so you should always consider running the lengthy computation in another process.

If your code needs to cause visible updates whilst continuing to do further computation, see **7.5.1 Updating windows in real time**.

4.2 Redisplay

The setting of any CAPI property that should affect the display causes CAPI to redisplay the relevant elements. However, when what is displayed depends on a state which is not a CAPI state, and this state changes, you may need to cause CAPI to redisplay.

For example, you may have a **`list-panel`** where the items are some objects, and the *print-function* generates a string for each object, based on some property of the object (typically a slot value). If that property changes then the display also needs to change, but there is no way for CAPI to know that so you need to tell CAPI explicitly.

A simple way to achieve this is to set a CAPI state which will cause redisplay. For example, doing:

```
(setf (capi:collection-items my-pane) (capi:collection-items my-pane))
```

leaves *my-pane*'s items unchanged, but because the value is set CAPI redisplays all of the items. This approach, however, is both computationally expensive when done often with large number of items, and causes flickering on screen that can be avoided.

Instead you can use one of the following functions.

- To update specific items in a **`choice`**, use **`redisplay-collection-item`**.
- To update menus and buttons in a window, use **`redisplay-interface`**.
- To update part of a **`pinboard-layout`**, use **`redraw-pinboard-layout`**.
- To update specific pinboard objects, use **`redraw-pinboard-object`**.
- In a **`tree-view`**, you can also use **`tree-view-update-item`** in cases when the update involves moving the child in its parent or completely removing the child.

4.2.1 Atomic redisplay

Often you need several distinct updates to the display to appear simultaneously. For example when you set the text in several elements at the same time, or you set the text of an element and then also set the background. To ensure that multiple updates appear together, wrap the macro **`with-atomic-redisplay`** around the updates.

4.3 Support for multiple monitors

CAPI supports positioning (and querying the position of) windows on multiple monitors.

The function **`screen-monitor-geometries`** supports the notion of monitor geometry. The monitor geometry includes "system" areas such as the macOS menu bar and the Microsoft Windows task bar.

The functions **`screen-internal-geometries`** and **`pane-screen-internal-geometry`** support the notion of internal geometry. The internal geometry excludes the system areas.

There is a "primary monitor" which displays any system areas. The origin of the coordinate system (as returned by **`top-level-interface-geometry`** and **`screen-internal-geometry`**) is the topmost/leftmost visible pixel of the primary monitor. Thus the origin may be in a system area such as the macOS menu bar.

The function **`virtual-screen-geometry`** returns a rectangle just covering the full area of all the monitors associated with a screen.

Note that code which relies on the position of a window should not assume that a window is located where it has just been programmatically displayed, but should query the current position. This is because the geometry includes system areas where CAPI windows cannot be displayed. For more information about this see **7.2 Resizing and positioning**.

Note also that CAPI does not currently support multiple desktops, which are called workspaces in Linux distros, and called Spaces on macOS.

5 Choices - panes with items

Some elements of a window interface contain collections of items, for example rows of buttons, lists of filenames, and groups of menu items. Such elements are known in the CAPI as *collections*.

In most collections, items may be selected by the user — for example, a row of buttons. Collections whose items can be selected are known as *choices*. Each button in a row of buttons is either checked or unchecked, showing something about the application's state — perhaps that color graphics are switched on and sound is switched off. This selection state came about as the result of a *choice* the user made when running the application, or default choices made by the application itself.

The CAPI provides a convenient way of producing groups of items from which collections and choices can be made. The abstract class **`collection`** provides a means of specifying a group of items. The subclass **`choice`** provides groups of selectable items, where you may specify what initial state they are in, and what happens when the selection is changed. Subclasses of **`collection`** and **`choice`** used for producing particular kinds of grouped elements are described in the sections that follow.

All the choices described in this chapter can be given a print function via the **`:print-function`** keyword. This allows you to control the way in which items in the element are displayed. For example, passing the argument **`'string-capitalize`** to **`:print-function`** would capitalize the initial letters of all the words of text that an instance of a choice displays. The default is **`princ-to-string`**.

Collections and choices inherit from the abstract class **`callbacks`**, which defines callbacks that are called in response to user gestures.

Some of the examples in this chapter require the callback function **`test-callback`** and **`hello`** which were introduced in **3 General Properties of CAPI Panes**.

5.1 Items

`choice`s in general can take arbitrary Lisp objects as the *items*, and then the behavior of the items (how they are displayed, callbacks) is determined by the properties of the **`choice`**. It is possible to give individual properties to individual items by using objects of class **`item`**, which encapsulates the properties of an item in a **`choice`**. The *items* of a **`choice`** can be a mixture of arbitrary objects and **`item`** instances.

`item` has several subclasses which are intended for specific **`choice`** subclasses, and these are documented in the entries for the specific **`choice`**s. The predicate **`itemp`** determines whether its argument is an instance of **`item`**.

5.2 Button panel classes

This section discusses the immediate subclasses of **`choice`** which can be used to build button panels. If you have a group of several buttons, you can use the appropriate **`button-panel`** element to specify them all as a group, rather than using **`push-button`** or **`check-button`** to specify each one separately. There are three such elements altogether: **`push-button-panel`**, **`check-button-panel`** and **`radio-button-panel`**. The specifics of each are discussed below.

5.2.1 Push button panels

The arrangement of a number of push buttons into one group can be done with a **`push-button-panel`**. Since this provides a panel of buttons which do not maintain a selection when the user clicks on them, **`push-button-panel`** is a **`choice`** that does not allow a selection. When a button is activated it causes a **`:selection-callback`**, but the button does not maintain the selected state.

Here is an example of a push button panel:

```
(setq push-button-panel
      (make-instance 'push-button-panel
                     :items '(one two three four five)
                     :selection-callback 'test-callback
                     :print-function 'string-capitalize))

(contain push-button-panel)
```

A push button panel

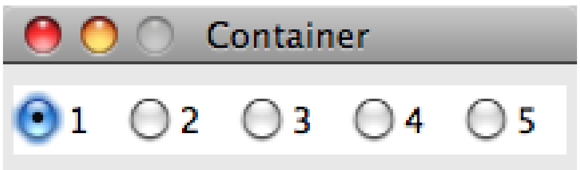
The layout of a button panel (for instance, whether items are listed vertically or horizontally) can be specified using the `:layout-class` keyword. This can take two values: `'column-layout` if you wish buttons to be listed vertically, and `'row-layout` if you wish them to be listed horizontally. The default value is `'row-layout`. If you define your own layout classes, you can also use these as values to `:layout-class`. Layouts, which apply to many other CAPI objects, are discussed in detail in **6 Laying Out CAPI Panes**.

5.2.2 Radio button panels

A group of radio buttons (a group of buttons of which only one at a time can be selected) is created with the `radio-button-panel` class. Here is an example of a radio button panel:

```
(setq radio (make-instance 'radio-button-panel
                :items (list 1 2 3 4 5)
                :selection-callback 'test-callback))


(contain radio)
```

A radio button panel



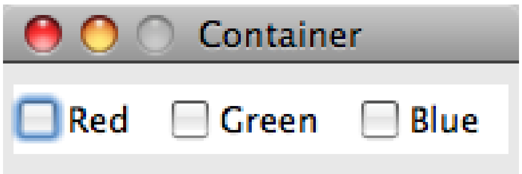
5.2.3 Check button panels

A group of check buttons can be created with the `check-button-panel` class. Any number of check buttons can be selected.

Here is an example of a check button panel:

```
(contain
 (make-instance
  'check-button-panel
  :items '("Red" "Green" "Blue")))
```

A check button panel



5.2.4 Mnemonics in button panels

On Windows and GTK+ you can specify the mnemonics (underlined letters) in a button panel with the **`:mnemonics`** initarg, for example:

```
(contain
 (make-instance 'push-button-panel
                :items '(one two three many)
                :mnemonics '(#\O #\T #\E :none)
                :print-function 'string-capitalize))
```

Notice that the value **`:none`** removes the mnemonic.

5.2.5 Programming button panels

The panels inherit the callbacks functionality from **`callbacks`**, most importantly the *selection-callback* and *retract-callback*, which are used as the default callbacks for the buttons.

The *items* functionality of button panel is inherited from **`collection`**. Typically you just use the initarg **`:items`** to specify the items, but in principle you can set the items dynamically. The other important functionality from **`collection`** is the *print-function* to define the strings that are displayed in the buttons.

Accessing the state of the buttons in **`check-button-panel`** and **`radio-button-panel`** is done by the selection functionality that is defined on **`choice`**. For example, making a **`check-button-panel`** with four buttons and the last is selected, and after two seconds selecting the first and the third:

```
(progn
  (setq cbp
        (capi:contain
         (make-instance 'capi:check-button-panel
                        :items '(1 2 3 4)
                        :selected-item 4)))
  (sleep 2)
  (capi:apply-in-pane-process
   cbp
   #'(lambda ()
       (setf (capi:choice-selected-items cbp)
             '(1 3)))))
```

All the button panel classes inherit from **`button-panel`**, which defines all the functionality of button panels. This includes a mechanism for specifying the layout of the buttons, images for the buttons, mnemonics, and also default and **Cancel** button. It also has an initarg **`:callbacks`** to define an individual selection callback for each item.

The function **`set-button-panel-enabled-items`** is used dynamically to enable/disable individual items in a panel.

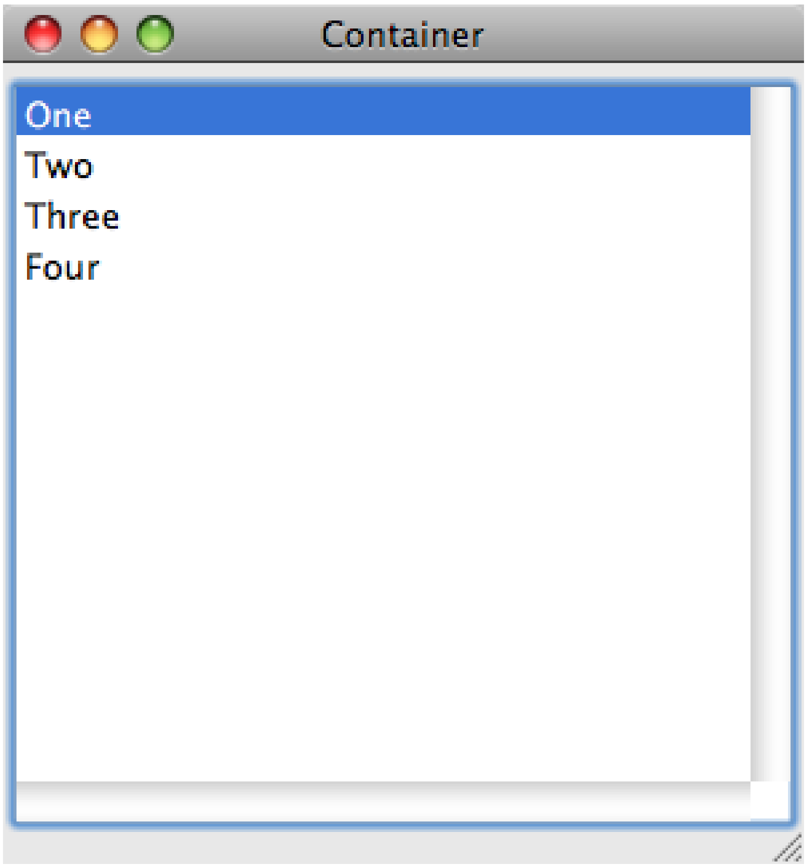
For more control over individual buttons, some (or all) of the items in a panel may be buttons themselves (that is, instances of a subclass of **`button`**). The behavior on an item that is actually a button is controlled by accessing the button.

5.3 List panels

Lists of selectable items can be created with the **list-panel** class. Here is a simple example of a list panel:

```
(setq list
      (make-instance 'list-panel
                     :items '(one two three four)
                     :visible-min-height '(character 2)
                     :print-function 'string-capitalize))


(contain list)
```

A list panel



Notice how the items in the list panel are passed as symbols, and a *print-function* is specified which controls how those items are displayed on the screen.

Any item on the list can be selected by clicking on it with the mouse.

By default, list panels are single selection — that is, only one item in the list may be selected at once. You can use the **:interaction** keyword to change this:

```
(setq list-panel
      (make-instance 'list-panel
                     :items (list "One" "Two" "Three" "Four")
                     :interaction :multiple-selection))

(contain list-panel)
```

You can add callbacks to any items in the list using the **:selection-callback** keyword.

```
(setq list-panel
      (make-instance 'list-panel
               :items (list "One" "Two" "Three" "Four")
               :selection-callback 'test-callback))

(contain list-panel)
```

5.3.1 List interaction

If you select different items in the list, only the last item you select remains highlighted. The way in which the items in a list panel interact upon selection can be controlled with the **`:interaction`** keyword.

The list produced in the example above is known as a single-selection list because only one item at a time may be selected. List panels are single-selection by default.

There are also multiple-selection and extended-selection lists available. The possible interactions for list panels are:

- **`:single-selection`** — only one item may be selected.
- **`:multiple-selection`** — more than one item may be selected.
- **`:extended-selection`** — see **5.3.2 Extended selection**.

To get a particular interaction, supply one of the values above to the **`:interaction`** keyword, like this:

```
(contain
 (make-instance
  'list-panel
  :items '("Red" "Green" "Blue")
  :interaction :multiple-selection))
```

Note that **`:no-selection`** is not a supported choice for list panels. To display a list of items with no selection possible you should use a **`display-pane`**.

5.3.2 Extended selection

Application users often want to make single *and* multiple selections from a list. Some of the time they want a new selection to deselect the previous one, so that only one selection remains — just like a **`:single-selection`** panel. On other occasions, they want new selections to be added to the previous ones — just like a **`:multiple-selection`** panel.

The **`:extended-selection`** interaction combines these two interactions. Here is an extended-selection list panel:

```
(contain
 (make-instance
  'list-panel
  :items '("Item" "Thing" "Object")
  :interaction :extended-selection))
```

Before continuing, here are the definitions of a few terms. The action you perform to select a single item is called the *selection gesture*. The action performed to select additional items is called the *extension gesture*. There are two extension gestures. To add a single item to the selection, the extension gesture is a click of the left button while holding down the **`Control`** key. For selecting a range of items, it is a click of the left button while holding down the **`Shift`** key.

5.3.3 Deselection, retraction, and actions

As well as selecting items, users often want to deselect them. Items in multiple-selection and extended-selection lists may be deselected.

In a multiple-selection list, deselection is done by clicking on the selected item again with either of the selection or extension gestures.

In an extended-selection list, deselection is done by performing the extension gesture upon the selected item. (If this was done using the selection gesture, the list would behave as a single-selection list and all other selections would be lost.)

Just like a selection, a deselection — or *retraction* — can have a callback associated with it.

For a multiple-selection list panel, there may be the following callbacks:

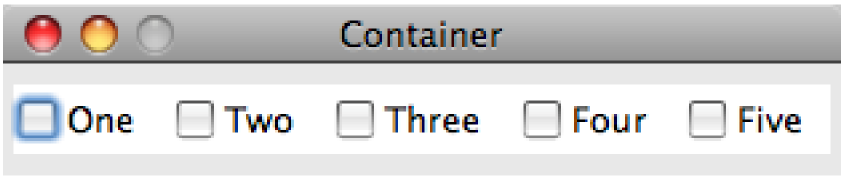
- **`:selection-callback`** — called when a selection is made.
- **`:retract-callback`** — called when a selection is retracted.

Consider the following example. The function **`set-title`** changes the title of the interface to the value of the argument passed to it. By using this as the callback to the **`check-button-panel`**, the title of the interface is set to the current selection. The *retract-callback* function displays a message dialog with the name of the button retracted.

1. Display the example window:

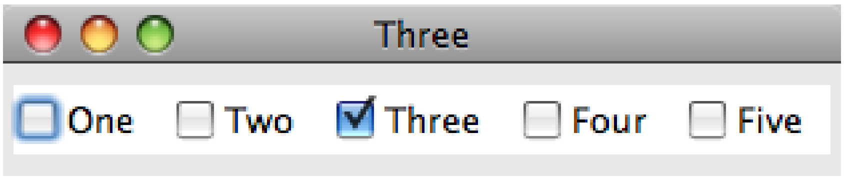
```
(defun set-title (data interface)
  (setf (interface-title interface)
        (format nil "~A" (string-capitalize data))))

(setq check-button-panel
      (make-instance 'check-button-panel
                     :items '(one two three four five)
                     :print-function 'string-capitalize
                     :selection-callback 'set-title
                     :retract-callback 'test-callback))

(contain check-button-panel)
```

The example check button panel before the callback.



2. Try selecting one of the check buttons. The window title will change:

The example check button panel after the callback.



3. Now de-select the button. Notice that the *retract-callback* is called.

For an extended-selection list panel, there may be the following callbacks:

- **`:selection-callback`** — called when a selection is made.

- **`:retract-callback`** — called when a selection is retracted.
- **`:extend-callback`** — called when a selection is extended.

Also available in extended-selection and single-selection lists is the action callback. This is called when you double-click on an item.

- **`:action-callback`** — called when a double-click occurs.

5.3.4 Selections in a list

List panels — all choices, in fact — can have selections, and you can set them from within Lisp. You can specify default settings and arrange for side-effects when a user selection is made. For the details see **5.10.2 Selections** ..

5.3.5 Images and appearance

A list panel can include images displayed on the left of each item. To include images supply the initarg **`:image-function`**. You can use images from an **`image-list`** via the initarg **`:image-lists`**.

Additionally, state images are supported on Microsoft Windows, GTK+ and Motif, via the initarg **`:state-image-function`** and, if required, **`:image-lists`**.

A list panel can have an alternating background color on Cocoa and GTK+, when specified by the initarg **`:alternating-background`**.

5.3.6 Filters

You can add a filter to a **`list-panel`** by passing the **`:filter`** initarg.

List panel filters are used in the LispWorks IDE, for example in the Inspector tool.

When a **`list-panel`** has a filter, you can the state of the filter by using **`list-panel-filter-state`**. The accessor **`collection-items`** on a **`list-panel`** with a filter returns the items after filtering. The function **`list-panel-unfiltered-items`** can be used to retrieve all the items. **`(setf collection-items)`** resets the filter, and **`(setf list-panel-unfiltered-items)`** can be used to set the items without affecting the filter. The function **`list-panel-items-and-filter`** can be used to get or set the unfiltered items and filter state together. **`(setf list-panel-items-and-filter)`** is especially useful, because setting the items and the filters separately causes the **`list-panel`** to redisplay twice.

5.3.7 Multi-column list panels

`multi-column-list-panel` is a subclass of **`list-panel`** which has several columns. Each line in a **`multi-column-list-panel`** displays several strings corresponding to a single item. **`multi-column-list-panel`** takes an initarg **`:item-print-functions`** which specifies how to generate the strings. The initarg **`:columns`** specifies column properties including width, alignment, and title.

The columns can have headers, which can be active (that is, they have callbacks). In particular, the headers can be made to sort the items based on some key and comparison function, by supplying the header's *selection-callback* as **`:sort`** and defining *sort-descriptions* (inherited from **`sorted-object`** via **`list-panel`**) with types that match the titles of the columns.

For an example see:

```
(example-edit-file "capi/choice/multi-column-list-panels")
```

5.3.8 Double list panel

double-list-panel is a **choice** that displays the items in two **list-panel**s side-by-side, and allows the user to move items between them. It is not a subclass of **list-panel**.

The selection interface functions (**choice-selected-items**, the **choice** accessor**choice-selection**, and so on) treat the items in one sub-panel as the selected items and the items in the other sub-panel as the non-selected items. **double-list-panel** takes more space, but is very convenient for the user when she needs to add or remove items from the selection, especially when there are many items.

5.3.9 Searching by keyboard input

list-panel has an initarg **:keyboard-search-callback** which allows you to define searches in the **list-panel** in response to user input. The function **list-panel-search-with-function** is intended to simplify writing the callback.

The default search uses a timeout to decide whether to:

- add an input character to the previous input to create the string to search, or:
- search for the character.

This timeout can be set by **set-list-panel-keyboard-search-reset-time**.

The *keyboard-search-callback* can actually be used to perform other tasks in response to user keyboard input.

For an example see:

```
(example-edit-file "capi/choice/list-panel-keyboard-search")
```

5.4 Trees

tree-view is a pane that displays a hierarchical list of items. Each item may optionally have an image and a checkbox.

Callbacks can be specified as for other choice classes. Additionally you can control how the nodes of the tree are expanded, and there is *delete-item-callback* available for use when the user presses the **Delete** key.

Tree views are used in the LispWorks IDE, for example in the **Output Data** view of the Tracer tool and the **Backtrace** area of the Debugger and Stepper tools.

5.4.1 Tree interaction

tree-view supports only the **:single-selection** *interaction* but you can have **:extended-selection** functionality by using the subclass **extended-selection-tree-view**.

5.4.2 Images and appearance

tree-view can include images displayed on the left of each item. To include images supply the initarg **:image-function**. You can use images from an **image-list** via the initarg **:image-lists**.

Additionally, state images are supported on Microsoft Windows, GTK+ and Motif, via the initarg **:state-image-function** and, if required, **:image-lists**.

A tree view can have an alternating background color on Cocoa and GTK+, when specified by the initarg **:alternating-background**.

5.5 Stacked trees

stacked-tree is a pane that displays a tree of items in a "stacked" drawing, where each item has an associated value and child items that represent a fraction of that value. Each item is displayed as a colored rectangle whose width corresponds to the value. Child items are displayed below the item to make a stack of rectangles.

The **Stacked Tree** tab of the Profiler tool in the LispWorks IDE is a situation where a stacked tree is useful.

For an example see:

```
(example-edit-file "capi/choice/stacked-tree")
```

5.6 Graph panes

Another kind of choice is the **graph-pane**. This is a special pane that can draw graphs, whose nodes and edges can be selected, and for which callbacks can be specified, as usual.

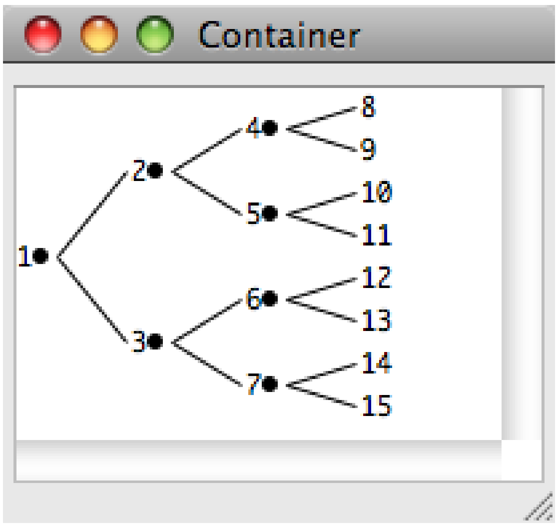
While **graph-pane** is a subclass of **choice** and hence **collection**, the concept of collection *items* is not applicable to a graph. Instead, the items in a **graph-pane** are constructed from a list of "roots" (arbitrary objects) which are specified by the initarg **:roots** and can be accessed later by **graph-pane-roots**, and a *children-function*. The roots define the initial nodes, and when the user expands a node, the *children-function* is called to compute the children, which is a list of more items, which specify the children nodes of the expanded node. Thus the actual items in the graph are changed as nodes are expanded or collapsed.

The concepts of selection, that is the functions **choice-selected-items** and so on, are applicable to **graph-pane**.

Here is a simple example of a graph pane. It draws a small rooted tree:

```
(contain
 (make-instance
  'graph-pane
  :roots '(1)
  :children-function
  #'(lambda (x)
      (when (< x 8)
       (list (* 2 x) (1+ (* 2 x)))))))
```

A graph pane



The graph pane is supplied with a **:children-function** which it uses to calculate the children of the root node, and from those children it continues to calculate more children until the termination condition is reached. For more details of this, see the manual page for **graph-pane**.

graph-pane provides a gesture which expands or collapses a node, depending on its current state. Click on the circle alongside the node to expand or collapse it.

You can associate selection, retraction, extension, and action callbacks with any or all elements of a graph. Here is a simple graph pane that has an action callback on its nodes.

First we need a pane which will display the callback messages. Executing the following form to create this pane:

```
(defvar *the-collector*
   (contain (make-instance 'collector-pane)))
```

Then, define the following four callback functions:

```
(defun test-action-callback (&rest args)
            (format (collector-pane-stream
                     *the-collector*) "Action"))

(defun test-selection-callback (&rest args)
            (format (collector-pane-stream *the-collector*)
                    "Selection"))

(defun test-extend-callback (&rest args)
            (format (collector-pane-stream *the-collector*)
                    "Extend"))

(defun test-retract-callback (&rest args)
            (format (collector-pane-stream *the-collector*)
                    "Retract"))
```

Now create an extended selection graph pane which uses each of these callbacks, the callback used depending on the action taken:

```
(contain
   (make-instance
      'graph-pane
      :interaction :extended-selection
      :roots '(1)
      :children-function
       #'(lambda (x)
           (when (< x 8)
                 (list (* 2 x) (1+ (* 2 x)))))
      :action-callback 'test-action-callback
      :selection-callback 'test-selection-callback
      :extend-callback 'test-extend-callback
      :retract-callback 'test-retract-callback))
```

The selection callback function is called whenever any node in the graph is selected.

The extension callback function is called when the selection is extended by middle clicking on another node (thus selecting it too).

The retract callback function is called whenever an already selected node is deselected.

The action callback function is called whenever an action is performed on a node (that is, whenever it gets a double-click, or **Return** is pressed while the node is selected).

5.6.1 Changing the graphics in the graph

graph-pane is actually a subclass of **pinboard-layout**, and displays the graph using elements (normally **pinboard-object**, but can also be **simple-pane**). You can specify the class of these elements, as well as a function to actually create the object for each node. This allows you to modify the appearance of the graph without affecting or accessing the topology of the graph.

You can also access the element that displays a **graph-object** by the reader **graph-object-element**, and manipulate it directly. See for example:

```
(example-edit-file "capi/graphics/graph-color-edges.lisp")
```

5.6.2 Controlling the layout

The roots of the graph are placed at one side of the panes and the graph grows into the pane. The side on which the roots are placed is defined by the *layout-function* and accessor **graph-pane-layout-function**, which takes one of the keyword values **:left-right**, **:top-down**, **:right-left** and **:bottom-up**, where the first word in a keyword is the side where the roots are placed. There is also an accessor **graph-pane-direction**, which maps **:forward** to/from **:left-right** and **:left-right**, and maps **:backward** to/from **:right-left** and **:bottom-up**, which makes it easier to set the *direction* without changing the vertical/horizontal dimension.

5.6.3 Accessing the topology of the graph

The topology of the graph is represented by **graph-node** objects and **graph-edge** objects. The list of **graph-node**s and **graph-edge**s of the **graph-pane** can be found by **graph-pane-edges** and **graph-pane-nodes**. Note, however, that these are subject to change as the user interacts with the graph.

You can find the node associated with an item (if any) by using **find-graph-node**. You can find the children of a supplied node by **graph-node-children**. You can find the edges from the node (that is, to its children) by the reader **graph-node-out-edges**, and edges in by **graph-node-in-edges**. You can also search for an edge between a parent and child by **find-graph-edge**. From a **graph-edge**, you can find the the parent and child that are connected by it by the accessors **graph-edge-from** and **graph-edge-to** respectively. It is possible to select specific nodes by **graph-pane-select-graph-nodes**, which takes a predicate that is applied to all the nodes.

You can find the geometry of a node, that is the part of the pane occupied by the element that is associated with the node, by the **graph-node** readers **graph-node-x**, **graph-node-y**, **graph-node-height** and **graph-node-width**. You can find whether a point in the pane is within the area of a graph object, either a **graph-node** or **graph-edge**, by using **graph-pane-object-at-position**.

It is possible to modify the graph explicitly by **graph-pane-delete-object**, **graph-pane-delete-objects**, **graph-pane-delete-selected-objects** and **graph-pane-add-graph-node**. However, that will be overridden next time the **graph-pane** computes the layout.

The user can interactively move nodes (and hence also edges) in the graph. If you need to know when that happens, you make a subclass of **graph-pane**, and then specialize **graph-pane-update-moved-objects** on it.

graph-node and **graph-edge** are both subclasses of **graph-object**, and inherit from it the readers **graph-object-object**, which returns the graph item associated with the **graph-object**, and **graph-object-element**, which returns the element that displays it (normally **pinboard-object**, but can also be **simple-pane**).

5.7 Option panes

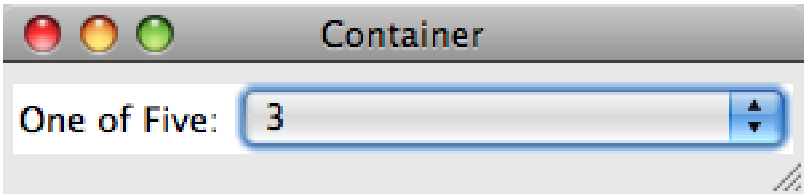
Option panes, created with the **option-pane** class, display the current selection from a single-selection list. When the user clicks on the option pane, the list appears and the user can make another selection from it. Once the selection is made, it is displayed in the option pane. In contrast to **text-input-choice**, the user cannot edit the selection.

The appearance of the **option-pane** list varies between platforms: a drop-down list box on Microsoft Windows; a combo box on GTK+ or Motif, and a popup list on Cocoa.

Here is an example option pane, which shows the choice of one of five numbers. The initial selection is controlled with **:selected-item**.

```
(contain
 (make-instance
  'option-pane
  :items '(1 2 3 4 5)
  :selected-item 3
  :title "One of Five:"))
```

An option pane



5.7.1 Option panes with images

You can add images to option pane items. Supply the **:image-function** initarg when creating the **option-pane**, as illustrated in:

```
(example-edit-file "capi/choice/option-pane-with-images")
```

5.8 Text input choice

The **text-input-choice** class allows arbitrary text input augmented with a choice like an **option-pane**. The user can edit the text after selecting it from the list.

See this example:

```
(example-edit-file "capi/elements/text-input-choice")
```

5.9 Menu components

Menus (covered in **8 Creating Menus**) can have components that are also choices. These components are groups of items that have an interaction upon selection just like other choices. The **:interaction** keyword is used to associate radio or check buttons with the group — with the values **:single-selection** and **:multiple-selection** respectively. By default, a menu component has an interaction of **:no-selection**.

See **8.3 Grouping menu items together** for more details.

5.10 General properties of choices

This section summarizes the general properties of choices.

5.10.1 Interaction

All choices have an interaction style, controlled by the **:interaction** initarg. The **radio-button-panel** and **check-button-panel** are simply **button-panel**s with their interactions set appropriately. The possible values for *interaction* are listed below.

:single-selection Only one item may be selected at a time: selecting an item deselects any other selected item.

:multiple-selection

A multiple selection choice allows the user to select as many items as she wants. A selected item may be deselected by clicking on it again.

:extended-selection

An extended selection choice is a combination of the previous two: only one item may be selected, but the selection may be extended to more than one item.

:no-selection Forces no interaction. Note that this option is not available for list panels. To display a list of items with no selection you should use a display pane instead.

Specifying an interaction style that is invalid for a particular choice causes an error.

The accessor **choice-interaction** is provided for accessing the *interaction* of a **choice**.

5.10.2 Selections

All choices have a selection. This is a state representing the items currently selected. The selection is represented as a list of indexes into the list of the choice's items, unless it is a single-selection choice, in which case it is just represented as an index. The indexes in the selection can be used to access the actual items using **get-collection-item**.

The initial selection is controlled with the initarg **:selection**. The **choice** accessor **choice-selection** is provided, and you can also use **(setf choice-selection)**.

Generally, it is easier to refer to the selection in terms of the items selected, rather than by indexes, so the CAPI provides the notion of a *selected item* and the *selected items*. The first of these is the selected item in a single-selection choice. The second is a list of the selected items in any choice.

The accessors **choice-selected-item** and **choice-selected-items** provide access to these conceptual slots, and you can also supply the values at **make-instance** time via the initargs **:selected-item** and **:selected-items**.

5.10.3 Callbacks in choices

All choices can have callbacks associated with them. Callbacks are invoked both by mouse button presses and keyboard gestures that change the selection or are "Action Gestures" such as **Return**. Different sorts of gesture can have different sorts of callback associated with them.

The following callbacks are available: **:selection-callback**, **:retract-callback** (called when a deselection is made), **:extend-callback**, **:action-callback** (called when a double-click occurs) and **:alternative-action-callback** (called when a modified double-click occurs). What makes one choice different from another is that they permit different combinations of these callbacks. This is a consequence of the differing interactions. For example, you cannot have an **:extend-callback** in a radio button panel, because you cannot extend selection in one.

Callbacks pass data to the function they call. There are default arguments for each type of callback. Using the **:callback-type** keyword allows you to change these defaults. Example values of *callback-type* are **:interface** (which causes the interface to be passed as an argument to the callback function), **:data** (the value of the selected data is passed), **:element** (the element containing the callback is passed) and **:none** (no arguments are passed). Also there is a variety of composite **:callback-type** values, such as **:data-interface** (which causes two arguments, the data and the interface, to be passed). For a complete description of **:callback-type** values, see the manual page for **callbacks**.

The following example uses a push button and a callback function to display the arguments it receives.

```
(defun show-callback-args (arg1 arg2)
  (display-message "The arguments were ~S and ~S" arg1 arg2))


(setq example-button
      (make-instance 'push-button
                     :text "Push Me"
                     :callback 'show-callback-args
                     :data "Here is some data"
                     :callback-type :data-interface))


(contain example-button)
```

Try changing the **:callback-type** to other values.

If you do not use the **:callback-type** argument and you do not know what the default is, you can define your callback function with lambda list **(&rest args)** to account for all the arguments that might be passed.

Specifying a callback that is invalid for a particular choice causes an error.

5.10.4 image-list, image-set and image-locator

Choices that need images for displaying items generally have an slot *image-function* which holds a function that returns the image to use for an item. The return value ultimately needs to evaluate to an **image** to display, but there are various ways to specify it. These include all the specifications that **load-image** understands. In addition, they can also be an integer which is an index into an **image-list** or an **image-locator**.

To use **image-list** in a **choice** you need to specify the **image-list** by the appropriate initarg, for example **:image-lists** for **tree-view**. See the entry for each specific class. Once the **choice** has **image-list**s, the *image-function* can return an index into the relevant list.

An **image-list** is an object that specifies an ordered set of images with a common width and common height. The images in the **image-list** can be **image** objects, image identifiers (pathname or symbol, which are automatically loaded by **load-image**), or **image-set** objects. You need to supply these objects when you make the **image-list** by **cl:make-instance**.

An **image-list** object can be used repeatedly in several panes. It is useful because it simplifies the handling of the images.

Example:

```
(example-edit-file "capi/choice/tree-view")
```

An **image-set** represents a group of images of the same size that are derived from a single object. For example, six images of 16x16 pixels each can be derived from a single image of 16x96 pixels. This is an example of the "general" **image-set**, which is created by **make-general-image-set**. In addition, you can create a scaled image set by either **make-scaled-general-image-set** or **make-scaled-image-set**. On Microsoft Windows, you can also create **image-set**s from resources in a DLL, either a bitmap resource by **make-resource-image-set**, or icon resource by **make-icon-resource-image-set**.

`image-set`s are useful because it is often convenient to hold a group of images as a combined larger image, which reduces the number of objects that needed to be dealt with. **`image-set`**s are used inside **`image-list`**s, and sometimes can be used directly, for example in **`toolbar`**. **`image-set`** can also be used in **`image-locator`**s.

Examples:

```
(example-edit-file "capi/choice/tree-view")
```

```
(example-edit-file "capi/elements/toolbar")
```

```
(example-edit-file "capi/choice/multi-column-list-panels")
```

An **`image-locator`** specifies one image out of an **`image-set`**, and it is created by **`make-image-locator`**. It can be used instead of an image in various places, most usefully as a result of the various *image-function*s.

Example:

```
(example-edit-file "capi/choice/multi-column-list-panels")
```

For **`choice`**s like **`tree-view`** or **`list-panel`**, you can include a sub-set from an **`image-set`** either by using image locators, or by including the **`image-set`** in an **`image-list`** and use the **`image-list`** in the **`choice`**. The latter technique is normally more convenient when all the **`image-set`** is used, but in other situations using **`image-locator`**s may be more convenient.

5.11 Operations on collections (choices) and their items

This section describes how you can access the items of a collection. In practice you will perform these operations on instances of subclasses of **`choice`**.

5.11.1 Accessing items

Given a collection and an index, you can retrieve the actual items in the collection by **`get-collection-item`**. Find the number of items in a collection at any point by **`count-collection-items`**. **`map-collection-items`** can be used to map a function over the collection items. **`print-collection-item`** can be used to "print" an item, that is generate the same string that will be displayed for this item. The **`collection`** accessor **`collection-items`** returns a list of the items in the collection, and can be used with **`setf`** to set the items.

5.11.2 Efficient manipulation of collection items

It is always possible to modify all the items of a collection by using **`setf`** with **`collection-items`** on it. However that can be expensive when called often with large numbers of items, and can cause flickering on screen. For typical choices (when *items-get-function* is **`svref`**), it is possible to modify the items of the choice more efficiently by using one of **`replace-items`**, **`remove-items`** or **`append-items`**.

Note: **`graph-pane`** and **`tree-view`** are not "typical" (their *items-get-function* is not **`svref`**) and therefore these functions cannot be used on these panes.

5.11.3 Searching in a collection

The function **`search-for-item`** can be used to find an item in a collection.

`find-string-in-collection` can be used to find a string in the printed items (that is, in the result of calling the print function). There is also **collection-find-string** which prompts the user for the string and then searches, and **collection-find-next-string** to continue the search from the previous match. **collection-last-search** can be used to retrieve the last search string, if any.

6 Laying Out CAPI Panes

The CAPI provides various layout classes which allow you to combine multiple window elements in a single window. This chapter provides an introduction to the different classes of layout available and the ways in which each can be used.

Layouts are created just like any other CAPI element, by calling **`make-instance`**. Each layout needs to have a *description* which is a list of the CAPI elements it contains. The description can be supplied via the **`:description`** initarg. It can also be supplied or modified later by calling **`(setf layout-description)`** in the layout's process. The *description* is interpreted by **`interpret-description`** as specifying a list of elements which are the "children" of the layout. The layout groups its children on the screen and specifies their geometry (*x* and *y* coordinates of top-left corner, *width* and *height*).

Only CAPI elements can be layout children. In this chapter "children" or "child" refers only to elements of these types:

- Instances of **`simple-pane`** and its subclasses.
- Instances of **`pinboard-object`** and its subclasses (discussed in **12 Creating Panes with Your Own Drawing and Input**).

For example, to put elements one above the other you make an instance of class **`column-layout`** with the elements as its *description*:

```
(defun put-in-a-column (list-of-elements)
  (make-instance 'column-layout
                 :description list-of-elements))
```

Since the result is a **`layout`**, you can put it in an **`interface`** and display it:

```
(defun display-in-a-column (list-of-elements)
  (display
   (make-instance 'interface
                  :layout (put-in-a-column list-of-elements))))

(display-in-a-column
 (list (make-instance 'text-input-pane
                      :text "Text input pane")
       (make-instance 'push-button
                      :data "Button")))

(display-in-a-column
 (loop for x below 10
       collect
       (make-instance 'push-button
                      :data (format nil "Button No. ~d" x))))
```

Layout themselves are subclasses of **`simple-pane`**, and hence can be children of other layouts, creating a hierarchical "tree" of layouts with other types of children as the "leaves". This is the normal way of laying out all the elements inside an interface. **`interface`** is also a subclass of **`simple-pane`** and can appear in the hierarchy, though usually **`interface`** is used only for the top-level window.

In general, the layouts need to know their childrens' geometrical requirements. These requirements are referred to as "constraints" and include the minimum and maximum width and height. Some of the child classes have default constraints, for example **`text-input-pane`** by default has both minimum and maximum height which allows showing one line, taking into account the height of the font. Most child classes do not have default constraints, and in effect have a minimum dimension of 0 and no maximum. Quite often that is good enough, but not always.

You can override the default constraints of an element by specifying geometrical "hints" (the word "constraint" is sometimes used to refer to the hint). Hints can be specified in many ways, for example the minimum width can be specified as enough to display 30 characters. Geometrical hints are typically specified by initargs when making a pane, but you can also set them dynamically. See **6.4 Specifying geometry hints** for details. In most cases, specifying the hints is sufficient (once you specify the hierarchy of layouts).

The function **`get-constraints`** computes the constraints in pixels based on the hints or the defaults, and returns the min/max of the width and height. Note that the result of **`get-constraints`** is dependent both on the hints themselves and other factors. For example, if the minimum width of an element is specified as "30 characters", changing the font of the element will cause **`get-constraints`** to return a different value. For more complex computations, it is also possible to define a **`calculate-constraints`** method, but in most cases the geometry hints are enough.

The layouts in general use **`get-constraints`** to get the constraints of their children, and take them into account when calculating the geometry of the elements and its own implicit constraints. For example, a **`row-layout`** puts elements side-by-side, and if it has two children with minimum width and height of 100, it will have an implicit minimum width of 200 and implicit minimum height of 100. The implicit constraints are used by **`get-constraints`** on the layout itself (by its parent), unless they are overridden by geometry hints or **`calculate-constraints`** on the layout.

The process of laying out starts at the top of the hierarchy, with the outer layout calling **`get-constraints`** on its children. If any of the children is a layout itself, it calls **`get-constraints`** of its children. Thus the **`get-constraints`** call is propagated down the hierarchy to all the tree, and the results are propagated back. Then the top layout lays out its children, that is it tells them their geometry, and again this is propagated down by each child which is a layout itself.

When a layout lays out its children, its uses its own geometry, the children's constraints and a layout-specific algorithm, which is implemented by **`calculate-layout`**. Thus when the documentation describes a layout of some class as "laying out its children in some way" it really means that this is what the applicable method of **`calculate-layout`** tries to achieve. Note that **`calculate-layout`** does not necessarily obey the constraints, and even the methods that intend to obey the constraints may fail to do so. For example, a **`row-layout`** with two children each of minimum width 100 which is given a width of 150 pixels will give only 50 to the second child. Conversely, when the layout has more space that the minimum required it usually distributes space between the elements that are not constrained by a maximum.

`calculate-layout` records the layout that it computed by setting the *x y width* and *height* in the geometries of the children (using **`with-geometry`**). The system then displays the children with the new geometry.

The hierarchy of layouts is laid out from the top layout of the top level interface when the interface is being displayed. After that, whenever the program makes a change to any element which may change its constraints, the system goes up the hierarchy until it finds a layout that it can tell is not going to need to change its constraints, and then lays out the children of that layout, as described above.

You can tell CAPI that the constraints of a pane may have changed and need to be recomputed (and hence maybe part of the hierarchy needs re-layout) by calling **`invalidate-pane-constraints`**.

Once again, you should make sure you have defined the **`test-callback`** function before attempting any of the examples in this chapter. Its definition is repeated here for convenience.

```
(defun test-callback (data interface)
  (display-message "Data ~S in interface ~S"
                   data interface))
```

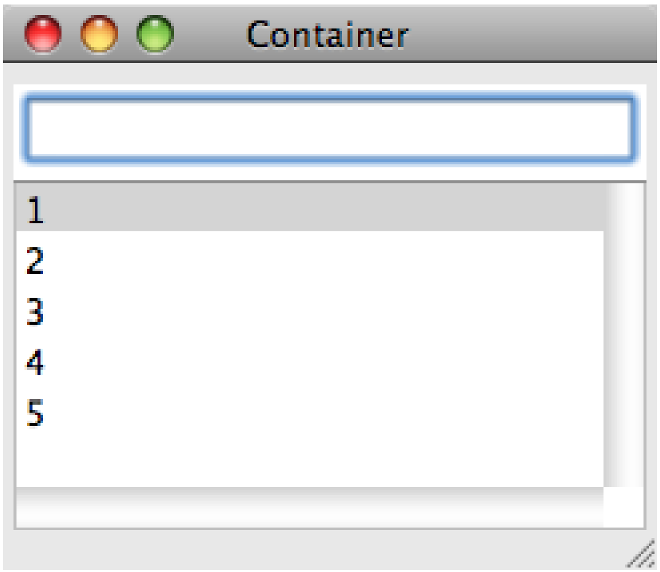
6.1 Organizing panes in columns and rows

You will frequently need to organize a number of different elements in rows and columns. The **`column-layout`** and **`row-layout`** elements are provided to make this easy.

The following is a simple example showing the use of **`column-layout`**.

```
(contain (make-instance 'column-layout
         :description (list
                      (make-instance 'text-input-pane)
                      (make-instance 'list-panel
                              :items '(1 2 3 4 5)))))
```

An example of using **column-layout**



1. Define the following elements:

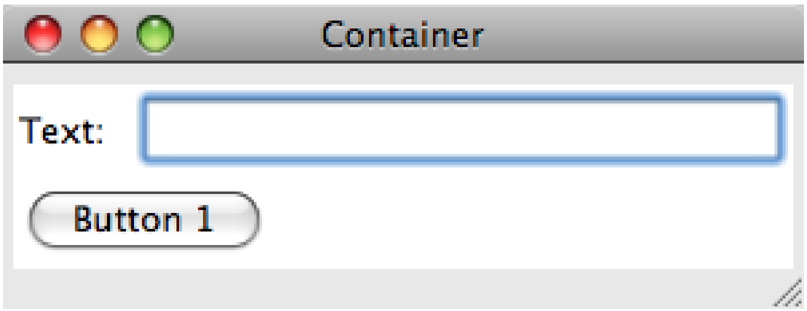
```
(setq button1 (make-instance 'push-button
                             :data "Button 1"
                             :callback 'test-callback))

(setq button2 (make-instance 'push-button
                             :data "Button 2"
                             :callback 'test-callback))

(setq editor (make-instance 'editor-pane
                            :text "An editor pane"))

(setq message (make-instance 'display-pane
                             :text "A display pane"))

(setq text (make-instance 'text-input-pane
                          :title "Text: "
                          :title-position :left
                          :callback 'test-callback))
```

These will be used in the examples throughout the rest of this chapter.

To arrange any number of elements in a column, create a layout using **column-layout**, listing the elements you wish to use. For instance, to display **title**, followed by **text** and **button1**, enter the following into a Listener:

```
(contain (make-instance 'column-layout
                        :description
                          (list text button1)))
```

A number of elements displayed in a column



To arrange the same elements in a row, simply replace **`column-layout`** in the example above with **`row-layout`**. If you run this example, close the column layout window first: each CAPI element can only be on the screen once at any time.

Layouts can be given horizontal and vertical scroll bars, if desired; the keywords **`:horizontal-scroll`** and **`:vertical-scroll`** can be set to **`t`** or **`nil`**, as necessary.

When creating panes which can be resized (for instance, list panels, editor panes and so on) you can specify the size of each pane relative to the others by listing the proportions of each. This can be done via either the **`:y-ratios`** keyword (for column layouts) or the **`:x-ratios`** keyword (for row layouts).

```
(contain (make-instance 'column-layout
          :description (list
                        (make-instance 'display-pane)
                        (make-instance 'editor-pane)
                        (make-instance 'listener-pane))
          :y-ratios '(1 5 3)))
```

You may need to resize this window in order to see the size of each pane.

Note that the heights of the three panes are in the proportions specified. The **`:x-ratios`** initarg will adjust the width of panes in a row layout in a similar way.

It is also possible to specify that some panes are fixed at their minimum size while others in the same row or column adjust proportionately when the interface is resized:

```
(contain
 (make-instance
  'column-layout
  :description
  (list
   (make-instance 'output-pane
                  :background :red
                  :visible-min-height '(:character 1))
   (make-instance 'output-pane
                  :background :blue
                  :visible-min-height '(:character 1))
   (make-instance 'output-pane
                  :background :red
                  :visible-min-height '(:character 3)))
  :y-ratios '(1 nil 3)
  :title "Resize this window vertically: the red panes maintain ratio 1:3, while the blue pane is f
ixed."
  ))
```

To arrange panes in your row or column layout with constant gaps between them, use the **`:gap`** initarg:

```
(contain
 (make-instance
  'column-layout
```

```
      :description (list
                    (make-instance 'output-pane
                                   :background :red)
                    (make-instance 'output-pane
                                   :background :white)
                    (make-instance 'output-pane
                                   :background :blue))
      :gap 20
      :title "Try resizing this window vertically"
      :background :gray))
```

To create resizable spaces between panes in your row or column layout, use the special value **nil** in the layout *description*:

```
(contain (make-instance 'column-layout
           :description (list
                         (make-instance 'output-pane
                                        :background :red)
                         nil
                         (make-instance 'output-pane
                                        :background :white)
                         nil
                         (make-instance 'output-pane
                                        :background :blue))
           :y-ratios '(1 1 4 1 1)
           :title "Try resizing this window vertically"
           :background :gray))
```

6.2 Other types of layout

Row and column layouts are the most basic type of layout class available in the CAPI, and will be sufficient for many things you want to do. A variety of other layouts are available as well, as described in this section.

6.2.1 Grid layouts

Row and column layouts only allow you to position a pane horizontally *or* vertically (depending on which class you use), but grid layouts let you specify both thus allowing you to create a complete grid of different CAPI panes.

grid-layout supports a title column, as illustrated in:

```
(example-edit-file "capi/layouts/titles-in-grid")
```

and it supports cells spanning multiple columns or rows, as illustrated in:

```
(example-edit-file "capi/layouts/extend")
```

grid-layout (and its subclasses **column-layout** and **row-layout**) is a subclass of **x-y-adjustable-layout**, which allows you to specify adjustments when you position the pane using the initargs **:x-adjust** and **:y-adjust**.

6.2.2 Simple layouts

A **simple-layout** has only one child. Where possible, the child is resized to fit the layout. Simple layouts are sometimes useful when you need to encapsulate a pane.

6.2.3 Pinboard layouts

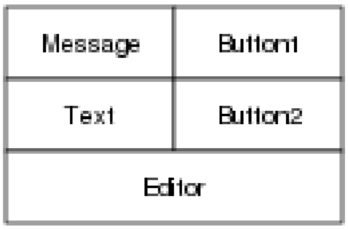
Pinboard layouts allow you to position a pane anywhere within a window, by specifying the *x* and *y* integer coordinates of the pane precisely. They are a means of letting you achieve any effect which you cannot create using the other available layouts, although their use can be correspondingly more complex. They are discussed in more detail in **12 Creating Panes with Your Own Drawing and Input**.

6.3 Combining different layouts

You will not always want to arrange all your elements in a single row or column. You can include other layouts in the list of elements used in any layout, thus enabling you to specify precisely how panes in a window should be arranged.

For instance, suppose you want to arrange the elements in your window as shown in **A sample layout**. The two buttons are shown on the right, with the text input pane and a message on the left. Immediately below this is the editor pane.

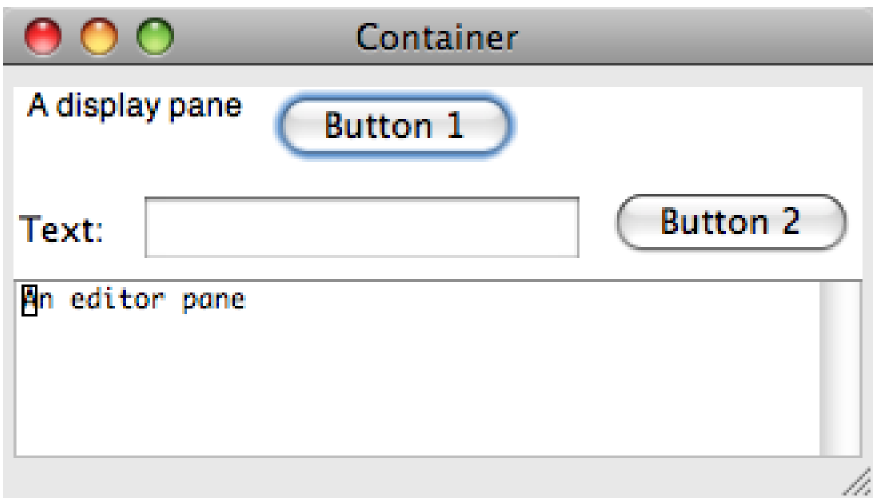
A sample layout



The layout in **A sample layout** can be achieved by creating two row layouts: one containing the display pane and a button, and one containing the text input pane and the other button, and then creating a column layout which uses these two row layouts and the editor.

```
(setq row1 (make-instance 'row-layout
                  :description (list message button1)))

(setq row2 (make-instance 'row-layout
                  :description (list text button2)))

(contain (make-instance 'column-layout
                        :description
                          (list row1 row2 editor)))
```

An instantiation of the sample layout



As you can see, creating a variety of different layouts is simple. This means that it is easy to experiment with different layouts, allowing you to concentrate on the interface design, rather than its code.

However, remember that each instance of a CAPI element must not be used in more than one place at the same time.

6.4 Specifying geometry hints

If you do not specify any hints, the CAPI uses the default constraints. In many cases that gives useful geometry already.

When you do need to specify the constraints, the normal way is to specify the hints for the element(s) when making them by passing the appropriate keywords. The available keywords and their meanings are explained in **6.4.1 Width and height hints**, and the potential values are explained in **6.4.2 Hint values formats**.

It also possible to set the hints later, either by **`set-geometric-hint`** to set a single hint or **`set-hint-table`** to set all of them.

It is also possible to specify initial constraints, which are applicable during the creation of the window, but not later. Typically that is used to force the initial window to be large enough, but later allowing the user to reduce the size.

6.4.1 Width and height hints

In CAPI, there are three kinds of geometry dimensions: external, visible and internal.

External and visible dimensions are two different ways to specify the dimensions of an element on the screen. The external dimension specifies the size of the element including its borders, while the visible dimension specifies the size of the pane inside its borders. Thus:

```
external-width  =  visible-width  + borders-width
external-height =  visible-height + borders-height
```

For a non-scrolling pane, internal dimensions mean the same as visible. For a scrolling pane, internal dimensions specify the size that the pane would need to display all of its data. For example, a **`list-panel`** with 100 items of which exactly 30 items are fully visible and each line is 15 pixels high has internal height of 100 x 15 = 1500 pixels and visible height of 30 x 15 = 450 pixels.

To get the right layout on the screen, you typically need to specify constraints on the width and height on the screen, which you do by specifying either the external constraints or visible constraints. This is the main way of using constraints.

The internal dimensions are needed only to compute the size of the scrollbars. Most elements implicitly compute their own internal dimensions. You should specify the minimum internal dimensions by **`:scroll-height`** and **`:scroll-width`** when you have an **`output-pane`** with scrollbar(s) which does ordinary scrolling (the default), so the pane can compute the size of the scrollbars. However, you can use **`set-horizontal-scroll-parameters`** and **`set-vertical-scroll-parameters`** instead.

The following keywords are used to specify geometrical constraints.

External constraints control the size that the pane takes up in its parent:

`:external-min-width`

The minimum width of the child in its parent.

`:external-max-width`

The maximum width of the child in its parent.

`:external-min-height`

The minimum height of the child in its parent.

:external-max-height

The maximum height of the child in its parent.

Visible constraints control the size of the part of the pane that you can see:

:visible-min-width The minimum visible width of the child.

:visible-max-width The maximum visible width of the child.

:visible-min-height

The minimum visible height of the child.

:visible-max-height

The maximum visible height of the child.

If the *visible-max-width* is the same as the *visible-min-width*, then the element is not horizontally resizable. If the *visible-max-height* is the same as the *visible-min-height*, then the element is not vertically resizable.

Internal constraints control the size of region used to display the contents of the pane: These are all deprecated.

:internal-min-width

The minimum width of the display region.

:internal-max-width

The maximum width of the display region.

:internal-min-height

The minimum height of the display region.

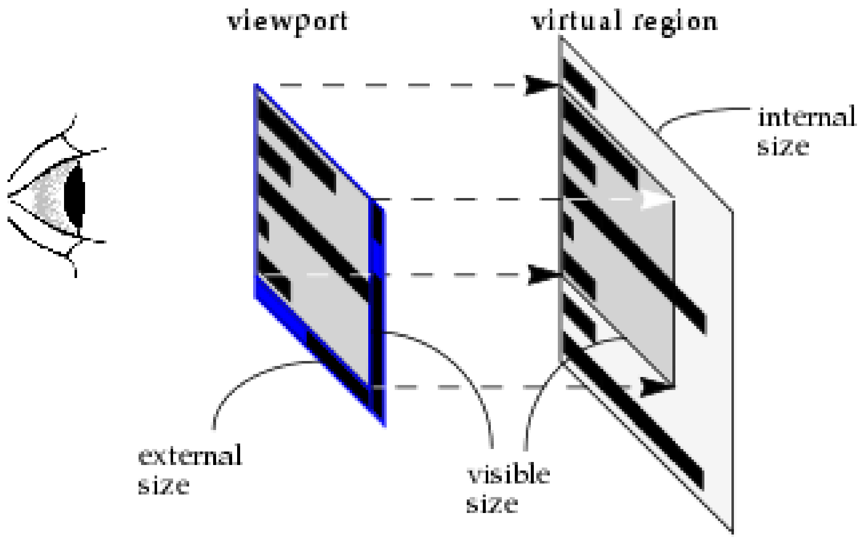
:internal-max-height

The maximum height of the display region.

In addition, methods for the generic function **calculate-constraints** can be defined on your pane classes to compute the internal geometries. Note that when scrolling the **:internal-max-width** and **:internal-max-height** are not meaningful and are ignored.

For a scrolling pane, the internal constraints control the size of region over which you can scroll and the visible constraints control the size of the viewport. Here is an illustration of the external, internal and visible sizes in a scrolling list panel with 8 items, 4 of which are fully visible and 1 is partially visible:

External, visible and internal sizes:



Initargs **`:min-width`**, **`:max-width`**, **`:min-height`** and **`:max-height`** are deprecated. They are synonyms for the visible constraints **`:visible-min-width`** and so on.

It is often wrong to constrain CAPI elements to fixed pixel sizes, as these constraints may lead to poorer layouts in some configurations.

6.4.1.1 Priority of constraints

The order of priority is the order in **6.4.1 Width and height hints**. That is, for a non-scrolling pane when there is only one independent constraint the preference order is:

External > Visible > Internal > **`calculate-constraints`**.

For a scrolling pane where there are two independent constraints the preference order for the external constraint is:

External > Visible.

and the preference order for the internal constraint is:

Internal > **`calculate-constraints`**.

6.4.2 Hint values formats

The possible values for the hints listed in **6.4.1 Width and height hints** are as follows:

integer	The size in pixels.
`t`	For **`:visible-max-width`**, **`t`** means use the value of **`:visible-min-width`**. For **`:visible-max-height`**, **`t`** means use the value of **`:visible-min-height`**.
`:text-width`	The width of any text in the element.
`:text-height`	The height of any text in the element.
`:screen-width`	The width of the screen.
`:screen-height`	The height of the screen.

A list starting with any of the following operators, followed by one or more hints:

max — the maximum size of the hints.

min — the minimum size of the hints.

+ — the sum of the hints.

- — the subtraction of hints from the first.

***** — the multiplication of the hints.

/ — the division of hints from the first.

A two element list specifying the size of a certain amount of text when drawn in the element:

(:character *integer***)** — the size of *integer* characters.

(character *integer***)** — the size of *integer* characters.

(:string *string***)** — the size of *string*.

(string *string***)** — the size of *string*.

A two-element list starting with **symbol-value**, and containing one other symbol:

(symbol-value *foo***)** — the size of the **symbol-value** of *foo*.

A list starting with **apply** or **funcall**, followed by a symbol and arguments:

(apply *function arg1 arg2* **...)** — the result of applying the function *function* to the arguments.

(funcall *function arg1 arg2* **...)** — the result of calling the function *function* with the arguments.

6.4.3 Initial constraints

You can use the initarg **:initial-constraints** to specify constraints that apply during creation of the element's interface, but not after the interface is displayed.

initial-constraints must be a plist of constraints, where the keywords are geometry hints as described above.

For example, this creates a window that starts at least 600 pixels high, but can be made shorter by the user, because that initial constraint is transient. However, the permanent height constraints on the two output panes remain in effect:

```
(contain
 (make-instance 'column-layout
                :description
                (list (make-instance 'output-pane
                                     :visible-min-height 100
                                     :background :red)
                      (make-instance 'output-pane
                                     :visible-min-height 200
                                     :background :blue))
                :initial-constraints '(:visible-min-height 600)))
```

6.5 Constraining the size of layouts

The size of a layout (often referred to as its *geometry*) is calculated automatically on the basis of the size of each of its children. The algorithm used takes account of *hints* provided by the children, and from the description of the layout itself. Hints are specified via the panes' initargs when they are created. The various pane classes have useful default values for these initargs.

6.5.1 Default Constraints

If you do not specify any hints, the CAPI calculates the on-screen geometry based on its default constraints. With this geometry the various elements are displayed with adequate space in the window.

This is designed to work regardless of variable factors such as the user's configuration, for example specifying large font sizes. It is often wrong to constrain CAPI elements to fixed pixel sizes, as these constraints may lead to poorer layouts in some configurations.

For information about the effect of constraints on scrolling, see **6.4.1 Width and height hints**.

6.5.2 Constraint Formats

Hints can take arguments in a number of formats, which are described in full under **6.4.2 Hint values formats**. When given a number, this should be an integer and the layout is constrained to that number of pixels. A constraint can also be specified in terms of character widths or heights, as shown in the next section.

6.5.2.1 Character constraints

In **6.3 Combining different layouts**, you created a window with five panes, by combining row and column layouts. Now consider changing the definition of the editor pane so that it is required to have a minimum size. This would be a sensible change to make, because editor panes need to be large enough to work with comfortably.

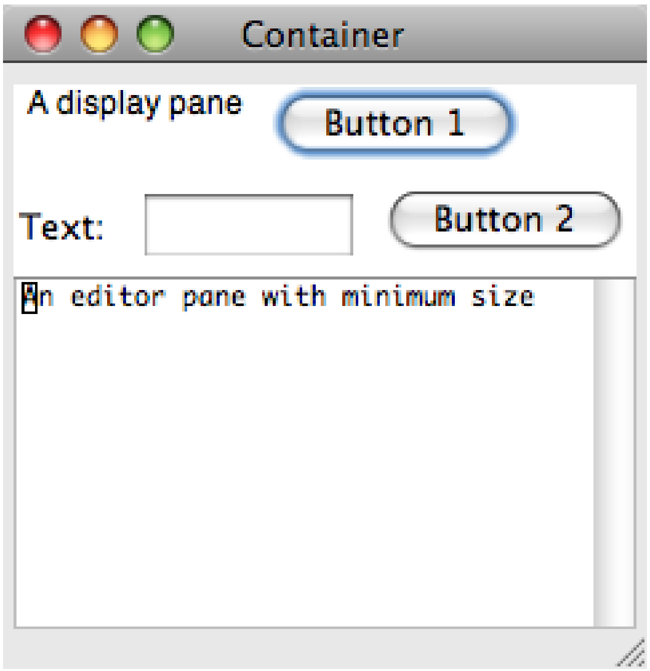
```
(setq editor2
      (make-instance 'editor-pane
                     :text "An editor pane with minimum size"
                     :visible-min-width '(:character 30)
                     :visible-min-height '(:character 10)))
```

Now display a window similar to the last example, but with the **`editor2`** editor pane. Note that it is only the description of the top-level column layout which differs. Before entering the following into the listener, you should close all the windows created in this chapter in order to free up the instances of **`button1`**, **`button2`** and so forth.

```
(contain (make-instance 'column-layout
                        :description
                        (list row1 row2 editor2)))
```

You will not be able to resize the window any smaller than this:

The result of resizing the sample layout



6.5.2.2 String constraints

To make a pane that is wide enough to accommodate a given string, use the **`:visible-min-width`** hint with a **`(:string`** *string***`)`** constraint.

In this example we also supply **`:visible-max-width t`**, which fixes the maximum visible width to be the same as the minimum visible width. Hence the pane is wide enough, but no wider:

```
(defvar *text* "Exactly this wide")

(capi:contain
 (make-instance 'capi:text-input-pane
                :text *text*
                :visible-min-width `(:string ,*text*)
                :visible-max-width t
                :font (gp:make-font-description
                       :size (+ 6 (random 30)))))
```

Note that the width constraint works regardless of the font used.

6.5.3 Changing the constraints

If you need to alter the constraints on an existing element, use the function **`set-hint-table`**. See how the interface in **6.5.2.1 Character constraints** resizes after this call:

```
(apply-in-pane-process editor2
 'set-hint-table editor2 '(:visible-min-width (:character 100)))
```

If you define your own **`pinboard-object`** class, ensure that its hint table matches the visible geometry and is kept synchronized after any movement of the object, otherwise redrawing may be incorrect.

Similarly if you draw pinboard objects under a **`transform`**, call **`set-hint-table`** with the transformed geometry to ensure correct redrawing.

6.6 Other pane layouts

The example below uses three predefined panes, which need to be defined as follows:

```
(setq red-pane (make-instance 'output-pane
                              :background :red))

(setq green-pane (make-instance 'output-pane
                                :background :green))

(setq blue-pane (make-instance 'output-pane
                               :background :blue))
```

6.6.1 Switchable layouts

A switchable layout allows you to place CAPI objects on top of one another and determine which object is displayed on top through Lisp code, possibly linked to a button or menu option through a callback. Switchable layouts are set up using a **switchable-layout** element in a **make-instance**. As with the other layouts, such as **column-layout** and **row-layout**, the elements to be organized are listed in the *description* slot, initialized in this example by the **:description** initarg:

```
(setq switching-panes (make-instance
                       'switchable-layout
                       :description (list red-pane green-pane)))

(contain switching-panes)
```

Note that the default pane to be displayed is the red pane, which was the first pane in the description list. The two panes can now be switched between using **switchable-layout-visible-child**:

```
(apply-in-pane-process
 switching-panes #'(setf switchable-layout-visible-child)
 green-pane switching-panes)

(apply-in-pane-process
 switching-panes #'(setf switchable-layout-visible-child)
 red-pane switching-panes)
```

6.6.2 Tab layouts

A **tab-layout** displays several tabs, and a single pane which contains the main contents.

In its simplest mode, a **tab-layout** is similar to a switchable layout, except that each pane is provided with a labelled tab, like the tabs on filing cabinet folders or address books. If the tab is clicked on by the user, the pane it is attached to is pulled to the front. Remember to close the switchable layout window created in the last example before displaying this:

```
(setq tab-layout
      (make-instance 'tab-layout
                     :items (list (list "one" red-pane)
                                  (list "two" green-pane)
                                  (list "three" blue-pane))
                     :print-function 'car
                     :visible-child-function 'second))

(contain tab-layout)
```

A tab layout

The example needs the `:print-function` to be `car`, or else the tabs will be labelled with the object numbers of the panes as well as the title provided in the list.

However, a tab layout can also be used in a non-switchable manner, with each tab responding with a callback to alter the appearance of only one pane. In this mode the `:description` keyword is used to describe the main layout of the tab pane. In the following example the tabs alter the choice of starting node for one graph pane, by using a callback to the `graph-pane-roots` accessor:

```
(defun tab-graph (items)
  (let* ((gp (make-instance 'graph-pane))
         (tl (make-instance 'tab-layout
               :description (list gp)
               :items items
               :visible-child-function nil
               :print-function (lambda (x) (format nil "~R" x))
               :callback-type :data
               :selection-callback #'(lambda (data)
                                       (setf (graph-pane-roots gp)
                                             (list data))))))
    (contain tl)))

(tab-graph '(1 2 4 5 7))
```

You can access the pane that is currently displayed in the `tab-layout` by `tab-layout-visible-child`, and you can obtain a list of the panes that have been displayed by calling `tab-layout-panes`.

6.6.3 Dividers and separators

If you need adjacent panes in a row or column to have a narrow user-movable divider between them, supply the special value **`:divider`** in the *description*. The divider allows the user to resize one pane into the space of the other. To see this in the column layout below, grab the divider between the two panes and then drag it vertically to resize both panes:

```
(contain (make-instance 'column-layout
                        :description (list green-pane
                                           :divider
                                           red-pane)))
```

The arrow keys can also be used to move the divider.

To include a narrow visible element between adjacent panes which cannot be moved (dragged) by the user, supply the special value **`:separator`** in the description.

If you also specify ratios, the ratio for each occurrence of either of these special values should be **`nil`** to specify that the narrow element is fixed at its minimum size:

```
(contain (make-instance 'column-layout
           :description (list
                         (make-instance 'output-pane
                                        :background :red)
                         :divider
                         (make-instance 'output-pane
                                        :background :white)
                         :separator
                         (make-instance 'output-pane
                                        :background :blue))
           :y-ratios '(1 nil 4 nil 1)
           :title "You can drag the divider, but not the separator"
           :background :gray))
```

Dividers and separators can also be placed between panes in a **`row-layout`** or even combinations of row and column layouts.

6.6.4 Static layout

`static-layout` is a layout that simply places each of its children where the geometry specifies (*x*, *y*, *visible-min-width* and *visible-min-height*). The children can be moved and resized by **`(setf static-layout-child-position)`** and **`(setf static-layout-child-size)`**.

An important subclass of **`static-layout`** is **`pinboard-layout`**, which is documented in **12.3 Creating graphical objects**. **`pinboard-layout`** is used to create your own kind of panes.

6.6.5 Interface toolbars

Your interface can have a toolbar which the user can configure by selecting and rearranging the buttons to display. To implement this, specify an *interface toolbar* as described in **9 Adding Toolbars**.

6.6.6 Docking layout

`docking-layout` allows docking/undocking of panes, which means interactively moving the panes between places in the interface (docking) and into standalone floating windows (undocking). The full functionality is available only on Microsoft Windows, while GTK+ gives very limited functionality. On Cocoa it is completely static. Docking layouts are especially useful for toolbars, but can contain other panes.

To allow moving a pane between different places in the interface, you need to group several **docking-layout**s. This done by using **make-docking-layout-controller** to create a controller object, and then passing the controller when making the **docking-layout** with the initarg **:controller**. You then place each **docking-layout** in a different place in the interface, by including it in the layout hierarchy of the interface in the usual way, and then it is possible to interactively move panes between all the **docking-layout**s that share the controller.

If you merely want to allow undocking, you do not need a controller.

The function **docking-layout-pane-docked-p** can be used to test whether a pane is docked in a specific **docking-layout**, and can be used with **cl:setf** to programmatically dock a pane in a specific **docking-layout** or to undock it (to do this, dock it to **nil**).

The function **docking-layout-pane-visible-p** can be used to test whether a pane is docked in one of the **docking-layout**s in the group of a **docking-layout** (that is, layouts with the same controller) or is undocked, and the **docking-layout** or the floating window is visible. It can be used with **cl:setf** to change the visibility of the **docking-layout** (if the pane is docked) or the floating window (undocked).

There is an example in:

```
(example-edit-file "capi/layouts/docking-layout")
```

6.6.7 Multiple-Document Interface (MDI)

In LispWorks for Windows, the CAPI supports MDI through the class **document-frame**. MDI is not supported on other platforms.

To use MDI in the CAPI, define an interface class that inherits from **document-frame**, and use the two special slots **capi:container** and **capi:windows-menu** as described below.

In your interface's layouts, use the symbol **capi:container** in the *description* to denote the pane inside the MDI interface in which child interfaces are added.

document-frame-container is a reader which returns the **document-container** of the **document-frame**.

Interfaces of any type other than subclasses of **document-frame** may be added as children. To add a child interface in your MDI interface, call **display** on the child interface and pass the MDI interface as the *screen* argument. This will display the child interface inside the container pane. To obtain a list of the child interfaces, call the **screen** reader function **screen-interfaces**, passing the frame's **document-container** as the *screen* argument.

You can use most of the normal CAPI window operations such as **top-level-interface-geometry** and **activate-pane** on windows displayed as children of a **document-frame**.

The slot **capi:windows-menu** contains the Windows Menu, which allows the user to manipulate child interfaces. The standard functionality of the Windows Menu is handled by the system and normally you will not need to modify it. However, you will want to specify its position in the menu bar. Do this by adding the symbol **capi:windows-menu** in the **:menu-bar** option of your **define-interface** form.

By default the menu bar is made by effectively appending the menu bar of the **document-frame** interface with the menu bar of the current child. You can customize this behavior with **merge-menu-bars**.

6.6.7.1 MDI example

This example uses **document-frame** to create a primitive **cl:apropos** browser.

Firstly we define an interface that lists symbols. There is nothing special about this in itself.

```
(capi:define-interface symbols-listing ()
  ((symbols :initarg :symbols))
  (:panes
   ( symbols-pane capi:list-panel
                  :items symbols
                  :print-function
                  'symbol-name))
  (:default-initargs
   :best-width '(character 40)
   :best-height '(character 10)))
```

Next we define the MDI interface. Note:

1. It inherits from **document-frame**.
2. **capi:container** is used in the layout description.
3. **capi:windows-menu** is in the **:menu-bar** list.
4. When the interface showing the symbols is being displayed, the MDI interface is passed as the *screen* argument to **display**.

Otherwise, this example uses standard Common Lisp and CAPI functionality.

```
(capi:define-interface my-apropos-browser
    (capi:document-frame)
  ((string :initarg :string))
  (:panes
   (package-list
    capi:list-panel
    :items
    (loop for package in (list-all-packages)
          when
          (let ((al (apropos-list string package)))
            (when al
              (cons (package-name package) al)))
          collect it)
    :print-function 'car
    :action-callback
    #'(lambda (mdi-interface name-and-symbols)
        (capi:display
         (make-instance
          'symbols-listing
          :symbols  (cdr name-and-symbols)
          :title (car name-and-symbols))
         :screen mdi-interface))
    :callback-type :interface-data)
   )
  (:menu-bar capi:windows-menu)
  (:layouts
   (main
    capi:row-layout
    '(package-list :divider capi:container)
    :ratios '(1 nil 4)))
  (:default-initargs
   :visible-min-height '(character 20)
   :visible-min-width '(character 100)))
```

To browse apropos of a specific string:

```
(capi:display
 (make-instance 'my-apropos-browser
                :string "EDITOR"))
```

6.7 Changing layouts and panes within a layout

To change to another layout, use **(setf pane-layout)**:

```
(setf layout
      (capi:contain
       (make-instance 'row-layout
                      :description
                      (list (make-instance 'title-pane :text "One")
                            (make-instance 'title-pane :text "Two"))
                      :visible-min-height 100)))

(apply-in-pane-process
 layout #'(setf pane-layout)
 (make-instance 'column-layout
                :description
                (list (make-instance 'title-pane :text "Three")
                      (make-instance 'title-pane :text "Four")))
 (element-interface layout))
```

To change the panes within a layout, use **(setf layout-description)**:

```
(setf layout
      (capi:contain
       (make-instance 'row-layout
                      :description
                      (list (make-instance 'title-pane :text "One")
                            (make-instance 'title-pane :text "Two"))
                      :visible-min-height 100)))

(apply-in-pane-process
 layout #'(setf layout-description)
 (list (make-instance 'title-pane :text "Three")
       (make-instance 'title-pane :text "Four")
       (make-instance 'title-pane :text "Five"))
 layout)
```

Note: A CAPI layout must not reuse panes that are already displayed in another layout.

7 Programming with CAPI Windows

An interface or its children can be altered programmatically in many ways. This chapter describes APIs for the most common of these.

Note: By default, each CAPI interface runs in its process. It is important to understand that an on-screen interface and its elements must be accessed only in the process of that interface. In most circumstances the user alters the interface by a callback inside the interface, which will automatically happen in the correct process. However, calls from other processes (including other CAPI interfaces) should use **`execute-with-interface`**, **`execute-with-interface-if-alive`**, **`apply-in-pane-process`** or **`apply-in-pane-process-if-alive`**.

7.1 Initialization

If necessary you can run code just before or just after your interface's windows are displayed on screen.

You can do this by defining a **`:before`** or **`:after`** method on the generic function **`interface-display`**. Your method will run just before or just after your interface is displayed on screen.

For example:

```
(defun make-text (self createdp)
  (multiple-value-bind (s m h dd mm yy)
      (decode-universal-time (get-universal-time))
    (format nil "Window ~S ~:[displayed~;created~] at ~2,'0D:~2,'0D:~2,'0D"
            self createdp h m s)))

(capi:define-interface dd () () (:panes (dp capi:display-pane)))

(defmethod capi:interface-display :before ((self dd))
  (with-slots (dp) self
    (setf (capi:display-pane-text dp)
          (make-text self t))))

(capi:contain (make-instance 'dd))
```

Sometimes initialization code can be put in the *create-callback* of your interface, though adding it in suitable methods for **`initialize-instance`** or **`interface-display`** is usually better.

7.2 Resizing and positioning

Programmatic resizing can be done using the function **`set-top-level-interface-geometry`**. For example, to double the width of an interface about its center:

```
(setf interface (contain (make-instance 'interface)))
```

Use the mouse or window manager-specific gesture to resize the interface, then evaluate:

```
(multiple-value-bind (x y w h)
    (top-level-interface-geometry interface)
  (execute-with-interface interface
                          'set-top-level-interface-geometry
                          interface
```

```
                              :x (round (- x (* 0.5 w)))
                              :y y
                              :width (* 2 w)
                              :height h))
```

All resize operations are subject to the constraints. The constraints can be altered programmatically as described in **6.5.3 Changing the constraints**.

Resize operations are also subject to automatic modification by the system in cases where the new window geometry coincides with a system area such as the macOS menu bar or the Microsoft Windows taskbar, as described in **7.2.1 Positioning CAPI windows**.

7.2.1 Positioning CAPI windows

You should not assume that a window is located where it has just been programmatically positioned. Instead you should query the current position by **`top-level-interface-geometry`**.

So if you wish to display CAPI interface windows *W1* and *W2* relative to each other. You should:

1. Display *W1* (by **`display`**), then:
2. Query position of *W1*, then:
3. Arrange for *W2* to have the desired relative position, for example in its **`make-instance`** or later by **`set-hint-table`**, then:
4. Display *W2*.

The reason for this is that the window system may disallow certain positions (for example on the macOS menu bar) therefore you cannot be certain of the position of *W1*.

7.3 Geometric queries

The visible size of a pane can be found by **`simple-pane-visible-height`** and **`simple-pane-visible-width`**, or **`simple-pane-visible-size`** (which returns two values, *width* and *height*). Other geometric values can be accessed using **`with-geometry`**. See **6.4.1 Width and height hints** for the meaning of visible, external and internal size.

The function **`convert-relative-position`** can be used to convert coordinates between one pane or screen to another pane or screen.

Inside a **`static-layout`** (including **`pinboard-layout`**) the function **`static-layout-child-position`** and **`static-layout-child-size`** can be used to find (and set) the coordinates of a child.

Setting coordinates of panes (other than inside a **`static-layout`**) is done by the layout mechanism which is described in **6 Laying Out CAPI Panes**. In most cases, you use geometric hints or set the scroll parameters, as described in **6.4 Specifying geometry hints**.

7.4 Scrolling

7.4.1 Programmatic scrolling

Programmatic scrolling is implemented with the generic function **`scroll`**. This example shows vertical scrolling in a **`list-panel`**:

```
(setf list-panel
```

```
          (contain
           (make-instance 'list-panel
                          :items (loop for i below 100 collect i)
                          :vertical-scroll t)))

    (apply-in-pane-process
     list-panel 'scroll list-panel :vertical :move 50)
```

11 Defining Interface Classes - top level windows shows how an **`editor-pane`** can be scrolled using editor commands.

An **`output-pane`** can be made to scroll - see **12.4 output-pane scrolling**.

You can also use the functions **`set-horizontal-scroll-parameters`** and **`set-vertical-scroll-parameters`** to affect scrolling operations.

The current scroll position can be found by using **`get-scroll-position`**. Using it later in a call to **`scroll`** with **`:move`** scrolls the pane back to the same position.

7.4.2 Scroll values and initialization keywords

The six **`:scroll-*`** **`simple-pane`** initargs for each dimension correspond to the six keyword arguments of **`set-horizontal-scroll-parameters`**/**`get-horizontal-scroll-parameters`** and **`set-vertical-scroll-parameters`**/**`get-vertical-scroll-parameters`** as follows:

Specifying scroll parameters: the correspondence between **`simple-pane`** initargs and keyword arguments

`simple-pane` initargs	keyword argument
`:scroll-horizontal-slug-size` **`:scroll-vertical-slug-size`**	**`:slug-size`**
`:scroll-start-x` **`:scroll-start-y`**	**`:min-range`**
`:scroll-width` **`:scroll-height`**	**`:max-range`**
`:scroll-initial-x` **`:scroll-initial-y`**	**`:slug-position`**
`:scroll-horizontal-step-size` **`:scroll-vertical-step-size`**	**`:step-size`**
`:scroll-horizontal-page-size` **`:scroll-vertical-page-size`**	**`:page-size`**

The values for all of these parameters should be real numbers. The set of values supplied for each dimension is treated independently from the other set.

The difference between the *max-range* and *min-range* specifies the range of scrolling. When applied to the scrollbar display, all the values are scaled by the ratio between the height/width of the scrollbar and the range, for example:

`slug-size-in-pixels =` *slug-size* **`* scrollbar-height-in-pixels / (`***max-range* **`-`** *min-range***`)`**

The *slug-position* is also translated by the *min-range*:

`slug-position-in-pixels = (`*slug-position* **`-`** *min-range***`) * scrollbar-height-in-pixels / (`***max-range* **`-`** *min-range***`)`**

The scrolling position of the pane is the *slug-position* (translated by the *min-range*) scaled by the ratio between the pane dimension (width or height) and the *slug-size*, that is:

pane-scrolling-position = (*slug-position* **-** *min-range***) * pane-dimension /** *slug-size*

When *slug-size* is not supplied or is **nil**, it is set to track the dimension of the pane, so the scaling factor above is 1, and all the other numbers can be considered as if specified in pixels in the internal coordinates of the pane. If *slug-size* is supplied, it is in effect creating a scaling factor between the values and the coordinates in the pane.

The *min-range* initial value defaults to 0, the *max-range* initial value defaults to either the width/height in pixels of the data in the pane if this is deducible, otherwise to the height of the pane. The latter is not useful, and typically the *max-range* is the one value that you have to specify. In many cases it is the only value you need to specify.

The initial *slug-position* defaults to 0.

The *step-size* defines the amount to scroll for a gesture that means step (typically clicking on the arrows at the ends of the scrollbar). It initially defaults to the dimension of a character in the pane in pixels. Note that this is normally useful only if *slug-size* is not set, otherwise it is scaled by **pane-dimension** / *slug-size*. If you set the *slug-size*, you probably want to set the *step-size* too.

page-size defines the amount to scroll for page gestures (typically clicking on the scroll bar outside the scroll slug). It initially defaults to *slug-size* - *step-size*, which is normally the useful value.

7.4.3 Automatic scrolling

Automatic scrolling of the parent to show the focus pane can be specified by using **scroll-if-not-visible-p**.

For **output-pane** with "internal" scrolling (see **12.4 output-pane scrolling**), you can force some area to become visible, that is scroll as needed, by using **ensure-area-visible**.

7.5 Updating pane contents

Use only the documented functions such as the accessors **(setf editor-pane-text)** and **(setf collection-items)** and so on to set the data in a pane. For details, see the manual pages for the particular pane class and its superclasses in **21 CAPI Reference Entries**.

7.5.1 Updating windows in real time

If your code needs to cause visible updates while continuing to do further computation, then you should run your computation in a separate thread which is not directly associated with the CAPI window.

Consider the following example where real work is represented by calls to **sleep**:

1. Evaluate this code:

```
(defun change-text (win text)
  (setf (title-pane-text win)
        text))

(defun my-callback (win)
  (change-text win "Go")
  (loop
   for i from 0 to 20 do
   (change-text win (format nil "~D" i))
   (sleep 0.1)))

(defun test ()
  (let* ((p1 (make-instance 'title-pane
```

```
                                :text "init"))
            (p2 (make-instance
                 'button :text "Go"
                 :callback-type :none
                 :callback #'(lambda ()
                               (my-callback p1)))))
       (contain
        (make-instance 'row-layout :description (list p1 nil p2))
        :width 200 :height 200)))
```

2. Run **(test)** and note that the updates do not appear until **my-callback** returns. This is because it uses only one thread.

3. Now try this modified callback which uses a worker thread to perform the calculations:

```
(defun my-work-function ()
  (let ((mbox (mp:ensure-process-mailbox)))
    ;; This should really have an error handler.
    (loop (let ((event (mp:process-read-event mbox
                          "Waiting for events")))
            (cond ((consp event)
                   (apply (car event) (cdr event)))
                  ((functionp event)
                   (funcall event)))))))

(setf *worker*
      (mp:process-run-function "Worker process" ()
                               'my-work-function))

(defun change-text (win text)
  (apply-in-pane-process win
                         #'(setf title-pane-text)
                         text win))

(defun my-callback (win)
  (mp:process-send
   *worker*
   #'(lambda ()
       (change-text win "Go")
       (loop
        for i from 0 to 20 do
        (change-text win (format nil "~D" i))
        (sleep 0.1)))))
```

4. Run **(test)** again: you should see the updates appear immediately.

A real application might also display an **Abort** button during the computation, with a callback that aborts the worker process.

Also see this example:

```
(example-edit-file "capi/elements/progress-bar-from-background-thread")
```

7.6 Edit actions on the active element

It is possible to perform standard edit actions like copy and paste on the current active element, which is not necessarily a CAPI pane, using the functions **active-pane-***edit-function*, for example **active-pane-copy**.

These functions find the active element and try to perform the operation on it. The active element can potentially not correspond to a CAPI pane, for example when prompting for a file the active element is somewhere in the dialog, which is a standard dialog of the windowing system rather than being a CAPI interface.

It is also possible to define what edit operations do when they are called on a pane in an interface class which you have defined, by specializing the **pane-interface-*** methods such as **pane-interface-copy-object**. For choices, there is also **item-pane-interface-copy-object**. Typically these methods will need to access the system clipboard, using **set-clipboard** and **clipboard** (see **18.6 Clipboard**).

7.7 Manipulating top-level windows

7.7.1 Visibility and focus

To bring a top level window to the front (on top of other windows) call **raise-interface**, and to put it behind other windows call **lower-interface**.

To hide a window call **hide-interface**, and to unhide it call **show-interface**.

To raise an interface and give the input focus to a pane inside it, call **activate-pane**. For more information about the input focus, see **3.1.5 Focus**.

You can test whether the interface in which a pane is contained is visible by calling **interface-visible-p**.

7.7.2 Iconifying and restoring windows

You can iconify an interface window as follows:

```
(setf (top-level-interface-display-state interface) :iconic)
```

You can also make it be hidden, maximized or restore it to normal, and you have the option to create it in one of these states initially. For the details see **top-level-interface-display-state**.

You can test whether an interface is iconified by calling **interface-iconified-p**.

7.7.3 Closing windows

To close a CAPI interface window unconditionally, call the generic function **destroy**.

To close a CAPI interface window such that its *confirm-destroy-function* is called first to allow the user to confirm, call **quit-interface**. You must call it in the window's process, for example in the callback of a menu item.

7.7.4 Finding interfaces

You can use the function **locate-interface** to find an interface of a specified class which is currently displayed. It uses the method **interface-match-p** to decide if there is any "matching" interface, in which case that is simply returned, otherwise it uses **interface-reuse-p** to decide if any instance of the class can be reused, in which case it reinitializes it using **reinitialize-interface** and returns it.

find-interface uses **locate-interface** to find an interface, and if succeeds it activates it, otherwise it creates a new interface. **find-interface** is used by the LispWorks IDE when starting the tools.

You can call **collect-interfaces** to obtain a list of displayed interfaces of a specific class.

It is possible to switch off locating of interfaces by calling **(setf reuse-interfaces-p)**. This causes **locate-interface** to always return **nil**, and hence **find-interface** will always create new interface. **Note:** The IDE uses a different switch for its own interfaces, which can be set from the **Preferences...** dialog.

7.7.5 Quitting applications

To make an application quit when one of its CAPI windows is closed, make that window's *destroy-function* call **`quit`**.

To arrange for a delivered CAPI application to quit automatically when all of its CAPI windows are closed, call **`deliver`** with **`:quit-when-no-windows t`**.

7.7.6 Preserving information when saving an IDE session

You can save a session in the LispWorks IDE, either programmatically by **`hcl:save-current-session`** or interactively from the **Tools** menu. If you integrate your own interfaces with the LispWorks IDE and want associated information to be preserved over session saving, you can define **`interface-preserve-state`** methods on your own interfaces. You can also use **`interface-preserving-state-p`** in the *destroy-callback* and **`interface-display`** methods to check for any destroying/displaying that is performed as part of session saving (as opposed to the normal **`display`**/**`destroy`** cycle).

8 Creating Menus

You can create menus for an application using the **`menu`** class. For more control you can also use **`menu-component`** and **`menu-item`**.

`menu`, **`menu-component`** and **`menu-item`** all inherit from the **`callbacks`** class, which defines callbacks that are called when the user selects an item in the menu. They also inherit from the **`menu-object`** class, which adds some menu-specific callback functionality, title and enabling.

You should make sure you have defined the **`test-callback`** and **`hello`** functions before attempting any of the examples in this chapter. Their definitions are repeated here for convenience.

```
(defun test-callback (data interface)
  (display-message "Data ~S in interface ~S"
                   data interface))

(defun hello (data interface)
  (declare (ignore data interface))
  (display-message "Hello World"))
```

The menus in the menu bar of a window are defined by the **`:menu-bar`** of the interface. See **`define-interface`**, the **`interface`** initarg **`:menu-bar-items`**, and **11.3.1 Adding menus**. The macro **`define-interface`** allows you to define menus by specifying the arguments that you would pass to **`cl:make-instance`** if you made them explicitly. The actual menus in the menu bar have the properties described in this chapter.

8.1 Creating a menu

A menu can be created in much the same way as any of the CAPI classes you have already met.

Enter the following into a Listener:

```
(setq menu
      (make-instance 'menu
                     :title "Foo"
                     :items '("One" "Two" "Three" "Four")
                     :callback 'test-callback))

(setq interface
      (make-instance 'interface
                     :menu-bar-items (list menu)))
(display interface)
```

This creates a CAPI interface with a menu, **`Foo`**, which contains four items. Choosing any of these items displays its arguments. Each item has the callback specified by the **`:callback`** keyword.

A submenu can be created simply by specifying a menu as one of the items of the top-level menu.

Enter the following into a Listener:

```
(setq submenu
      (make-instance 'menu
                     :title "Bar"
                     :items '("One" "Two" "Three" "Four")
```

```
                         :callback 'test-callback))
(setq menu
      (make-instance 'menu
                         :title "Baz"
                         :items (list 1 2 submenu 4 5)
                         :callback 'test-callback))
(contain menu)
```

This creates an interface which has a menu, called **Baz**, which itself contains five items. The third item is another menu, **Bar**, which contains four items. Once again, selecting any item returns its arguments.

Menus can be nested as deeply as required using this method.

Note: In general you must not use a CAPI menu object in multiple different places in menu bar(s) at the same time. This is because menu bar menus are created when the interface is displayed, and (like any other CAPI pane) cannot be used elsewhere at the same time. Supply distinct instances instead. The one exception is popup menus, which are actually created only when they are on the screen, so they can be used repeatedly and in different places.

8.2 Presenting menus

The most common way of presenting menus is in the menu bar. This is done by putting the menus in the menu bar of an interface, typically by using **`:menu-bar`** in **`define-interface`**. It is also possible to set the menu bar dynamically using **`(setf interface-menu-bar-items)`**.

On Cocoa, you may want to define the application menu, the menus that are shown when no interface is active, and maybe a Dock context menu. For these, you will need to define your own subclass of **`cocoa-default-application-interface`**, and use **`set-application-interface`** on an instance of this class. See entry for **`cocoa-default-application-interface`**.

Pane-specific menus are invoked automatically by the system for the appropriate user gesture. See **8.12 Popup menus for panes** for a full discussion of the mechanism that finds the menu to raise.

There is also a special pane **`popup-menu-button`**, which raises a menu when clicked.

In addition, you can raise a menu programmatically by calling **`display-popup-menu`**.

8.3 Grouping menu items together

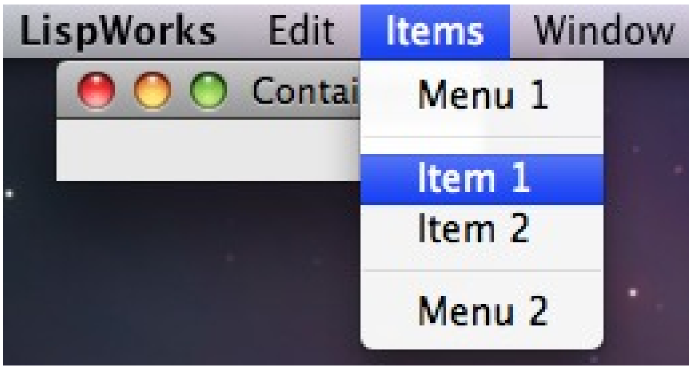
The **`menu-component`** class lets you group related items together in a menu. This allows similar menu items to share properties, such as callbacks, and to be visually separated from other items in the menus. Menu components are actually choices.

Here is a simple example of a menu component. This creates a menu called **Items**, which has four items. **Menu 1** and **Menu 2** are ordinary menu items, but **Item 1** and **Item 2** are created from a menu component, and are therefore grouped together in the menu.

```
(setq component (make-instance 'menu-component
                    :items '("item 1" "item 2")
                    :print-function 'string-capitalize
                    :callback 'test-callback))

(contain (make-instance 'menu
                        :title "Items"
                        :items
                        (list "menu 1" component "menu 2")
                        :print-function 'string-capitalize
                        :callback 'hello)
         :width 150
```

```
        :height 0)
```

A menu

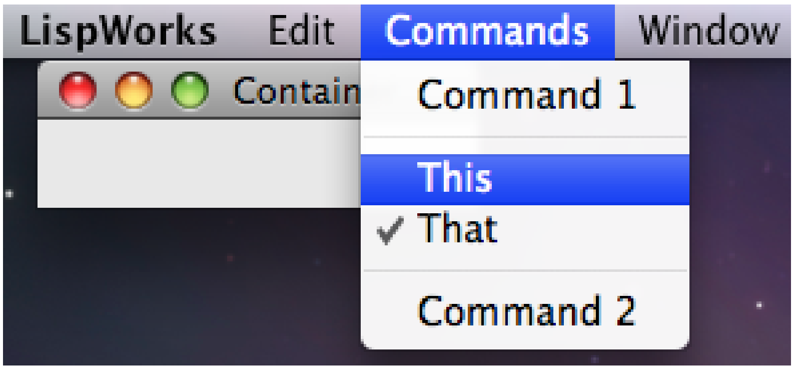


Menu components allow you to specify, via the **`:interaction`** keyword, selectable menu items — either as multiple-selection or single-selection items. This is like having radio buttons or check boxes as items in a menu, and is a popular technique among many GUI applications.

The following example shows you how to include a panel of radio buttons in a menu.

```
(setq radio (make-instance 'menu-component
                :interaction :single-selection
                :items '("This" "That")
                :callback 'hello))

(setq commands (make-instance 'menu
                :title "Commands"
                :items
                (list "Command 1" radio "Command 2")
                :callback 'test-callback))

(contain commands)
```

Radio buttons included in a menu



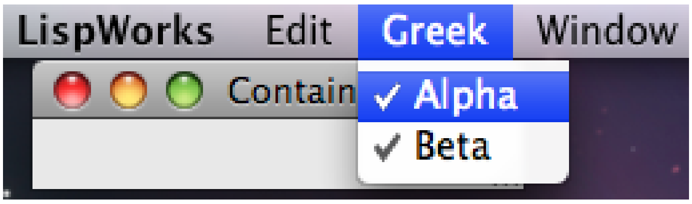
The menu items **This** and **That** are radio buttons, only one of which may be selected at a time. The other menu items are just ordinary commands, as you saw in the previous examples. Note that the CAPI automatically groups the items which are parts of a menu component so that they are separated from other items in the menu.

This example also illustrates the use of more than one callback in a menu, which of course is the usual case when you are developing real applications. Choosing either of the radio buttons displays one message on the screen, and choosing either **Command1** or **Command2** returns the arguments of the callback.

Checked menu items can be created by specifying **`:multiple-selection`** to the **`:interaction`** keyword, as illustrated below.

```
(setq letters (make-instance 'menu-component
                  :interaction :multiple-selection
                  :items (list "Alpha" "Beta")))

(contain (make-instance 'menu
                  :title "Greek"
                  :items (list letters)
                  :callback 'test-callback))
```

An example of checked menu items



Note how the items in the menu component inherit the callback given to the parent, eliminating the need to specify a separate callback for each item or component in the menu.

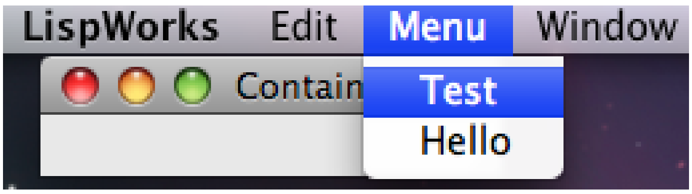
Within a menu or component, you can specify alternatives for a main menu item that are invoked by modifier keys. See **8.8 Alternative menu items** for more information.

8.4 Creating individual menu items

The **menu-item** class lets you create individual menu items. These items can be passed to menu-components or menus via the **:items** keyword. Using this class, you can assign different callbacks to different menu items.

```
(setq test (make-instance 'menu-item
                          :title "Test"
                          :callback 'test-callback))

(setq hello (make-instance 'menu-item
                          :title "Hello"
                          :callback 'hello))

(setq group (make-instance 'menu-component
                          :items (list test hello)))

(contain group)
```

Individual menu items



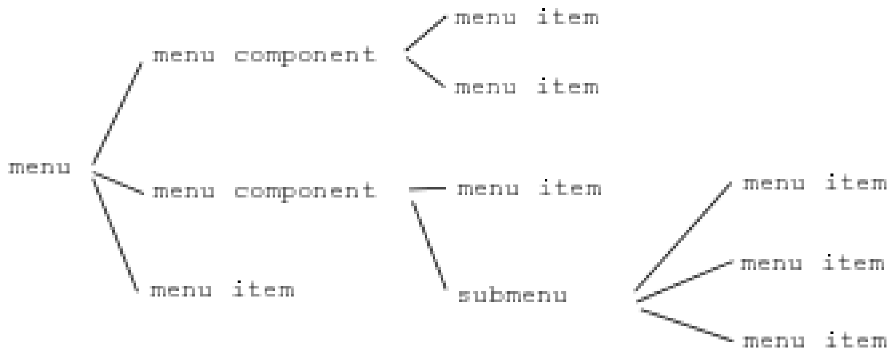
Remember that each instance of a menu item must not be used in more than one place at a time.

8.5 The CAPI menu hierarchy

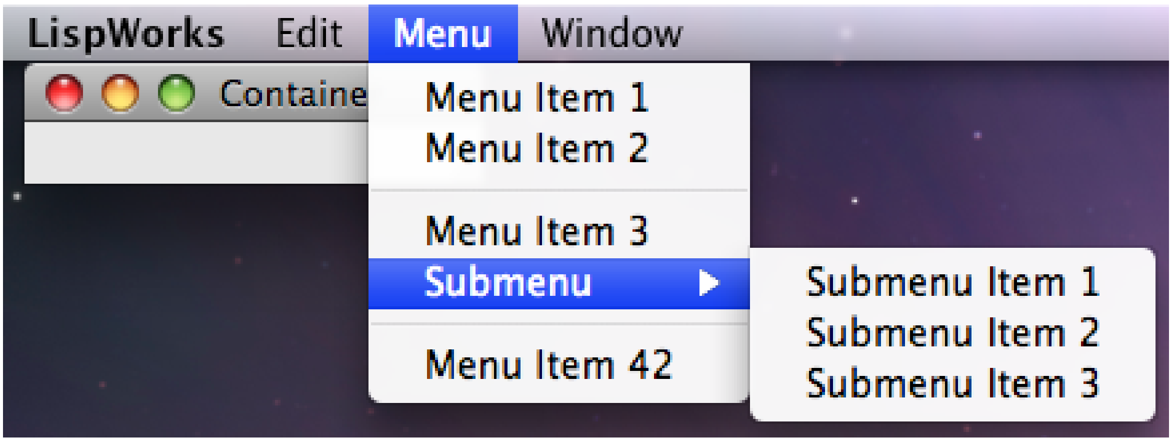
The combination of menu items, menu components and menus can create a hierarchical structure as shown schematically in **A schematic example of a menu hierarchy** and graphically in **An example of a menu hierarchy**. This menu has five elements, one of which is itself a menu (with three menu items) and the remainder are menu components and menu items. Items in a menu inherit values from their parent, allowing similar elements to share relevant properties whenever possible.

```
(defun menu-item-name (data)
  (format nil "Menu Item ~D" data))

(defun submenu-item-name (data)
  (format nil "Submenu Item ~D" data))

(contain
 (make-instance
  'menu
  :items
  (list
   (make-instance 'menu-component
                  :items '(1 2)
                  :print-function 'menu-item-name
                  )
   (make-instance 'menu-component
                  :items
                  (list 3
                        (make-instance
                         'menu
                         :title "Submenu"
                         :items '(1 2 3)
                         :print-function
                         'submenu-item-name))
                  :print-function 'menu-item-name)
   (make-instance 'menu-item
                  :data 42))
  :print-function 'menu-item-name))
```

A schematic example of a menu hierarchy



An example of a menu hierarchy



8.6 Mnemonics in menus

On Microsoft Windows and GTK+ you can control the mnemonics in menu titles and menu items using the initargs **:mnemonic**, **:mnemonic-title** (and if necessary **:mnemonic-escape**).

This example illustrates the various ways you can specify the mnemonics in a menu:

```
(contain
 (make-instance
  'menu
  :mnemonic-title "M&nemonics"
  :items
  (list
   (make-instance 'menu-item
                  :data "Menu Item 1"
                  :mnemonic #\1)
   (make-instance 'menu-item
                  :data "Menu Item 2"
                  :mnemonic 10)
   (make-instance 'menu-item
                  :mnemonic-title "Menu Item &3")
   (make-instance 'menu-item
                  :mnemonic-title "Menu Item !4"
                  :mnemonic-escape #\!)
   (make-instance 'menu-item
                  :data "Menu Item 5"
                  :mnemonic :default)
   (make-instance 'menu-item
                  :data "Menu Item 6"
                  :mnemonic :none))))
```

This example shows two ways to specify menu title mnemonics within the **:menus** option of a **define-interface** form. The first way, using **:mnemonic**, is the most natural:

```
(capi:define-interface menu-bar-mnemonics ()
  ()
  (:panes (pane1 capi:text-input-pane
                 :visible-min-width 200))
  (:layouts (main-layout
             capi:column-layout '(pane1)))
  (:menus
   (menu1 "Menu One"
          (("Foo"))
          :mnemonic #\O)
   (menu2 nil
          (("Bar"))
          :mnemonic-title "Menu &Two"))
  (:menu-bar menu1 menu2))

(capi:display (make-instance 'menu-bar-mnemonics))
```

8.7 Accelerators in menus

To define an accelerator key for a menu command, supply the initarg *accelerator* to the **menu-item**. See **menu-item** for the details.

8.7.1 Standard default accelerators

On Microsoft Windows and GTK+, by default a standard accelerator is added to a menu item if its title matches a standard menu command. The standard accelerators are:

Edit > Copy	`Ctrl+C`
Edit > Cut	`Ctrl+X`
Edit > Find...	`Ctrl+F`
Edit > Paste	`Ctrl+V`
Edit > Redo	`Ctrl+Y`
Edit > Replace...	`Ctrl+H`
Edit > Select All	`Ctrl+A`
Edit > Undo	`Ctrl+Z`
File > Close	`Ctrl+W`
File > Exit	`Ctrl+Q`
File > New	`Ctrl+N`
File > Open...	`Ctrl+O`
File > Print...	`Ctrl+P`
File > Save	`Ctrl+S`
Works > Refresh	`F5`

8.8 Alternative menu items

Menus can include "alternative" items, which are invoked if some modifiers are held while selecting the "main" item. The modifiers are defined by the **`:accelerator`** initarg of the item, which also allows the item to be invoked by a keyboard accelerator key if specified. On Cocoa, the title and accelerator of the alternative item appear when the appropriate modifier(s) are pressed.

A menu item becomes an alternative to an immediately previous item when it is made with initargs **`:alternative t`**. Each alternative item must have the same parent as its previous item. That is, they are within the same menu and menu component, as described in **8.3 Grouping menu items together**. More than one alternative item can be supplied for a given main item by putting them consecutively in the menu. The main item is the item preceding the first alternative item.

The main item and its alternative items forms a group of items. The accelerators of all items in the group must consist of the same key, but with different modifiers. If there is no need for an accelerator key, the main item should not have an accelerator and the alternative items should have accelerators with **`Null`** as the key, for example **`"Shift-Null"`**.

When the menu is displayed, only one item from the group will be shown. On Windows, GTK+ and Motif the main item is always displayed. Cocoa displays the item with the least number of modifiers initially, so to get a consistent cross-platform behavior, the main item should have the least number of modifiers. On Cocoa, pressing modifier keys that match alternative items changes the title and accelerators displayed for the item.

When the user selects an item with the modifiers pressed, the appropriate alternative item is selected.

To make a **`menu-item`** an alternative item, pass the initarg **`:alternative t`** and a suitable value for the initarg **`:accelerator`**.

There is an example illustrating alternative menu items in:

```
(example-edit-file "capi/elements/accelerators")
```

Note: Accelerators of alternative items do not work on Motif.

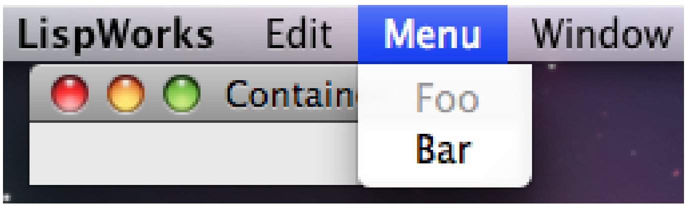
8.9 Disabling menu items

A function can be specified via the **:enabled-function** initarg (inherited from **menu-object**), that determines whether or not the menu, menu item, or menu component is enabled. By default, a menu object is always enabled.

Consider the following example:

```
(defvar *on* nil)

(contain
 (make-instance 'menu
                :items
                (list
                 (make-instance
                 'menu-item
                 :title "Foo"
                 :enabled-function
                 #'(lambda (menu) *on*))
                (make-instance
                 'menu-item
                 :title "Bar"))))
```

A menu with a disabled menu item



Changing the value of ***on*** between **t** and **nil** in the Listener, using **setq**, results in the menu item changing between the enabled and disabled states.

8.9.1 Dialogs and disabled menu items

By default, items in the menu bar menus and sub-menus are disabled while a dialog is on the screen on top of the active window. You can override this by passing a suitable value for the **menu-item** initarg **:enabled-function-for-dialog**.

8.10 Menus with images

You can add images to menu items. Supply the **:image-function** initarg when creating the **menu**, as illustrated in:

```
(example-edit-file "capi/elements/menu-with-images")
```

Note: on some platforms support for images in menus is limited to menu items without text and/or images without transparency. If **pane-supports-menus-with-images** returns true, then images are fully supported in menus.

8.11 The Edit menu on Cocoa

This section is only applicable to LispWorks for Macintosh.

LispWorks for Macintosh adds a minimal **Edit** menu to all CAPI interfaces when running in the LispWorks IDE, which makes the edit gestures **`Command+V`**, **`Command+C`** and **`Command+X`** work in every interface displayed in the LispWorks IDE.

However, to implement these gestures in your CAPI/Cocoa runtime application, you must include an **Edit** menu explicitly in your interface definition, as described in **11.3.1 Adding menus**.

Here is a minimal example of an **Edit** menu:

```
(edit-menu
 "Edit"
 (("Cut" :callback 'capi:active-pane-cut
   :enabled-function 'capi:active-pane-cut-p)
  ("Copy" :callback 'capi:active-pane-copy
   :enabled-function 'capi:active-pane-copy-p)
  ("Paste" :callback 'capi:active-pane-paste
   :enabled-function 'capi:active-pane-paste-p))
 :callback-type :interface)
```

To remove the automatic menu when running your program in the LispWorks IDE, pass the initarg **`:auto-menus nil`** when making the interface.

Note that, in the presence of an application interface (see **`cocoa-default-application-interface`**), a CAPI interface with no menus of its own and with **`:auto-menus nil`** uses the menu bar from the application interface.

8.12 Popup menus for panes

The CAPI tries to display a popup menu for a pane when the **`:post-menu`** gesture is entered by the user (mouse-right-click or **`Shift+F10`** on Microsoft Windows, GTK+ or Motif, control-click on Cocoa). See below for the special case of **`output-pane`**.

It first tries to get a menu for the pane. There are two mechanisms by which it can get a menu: which is tried depends on the value of *pane-menu*.

1. If the pane's initarg *pane-menu* is not **`:default`** in the call to **`make-instance`**, then its value is used. If the value is a function or a fbound symbol, it is called with four arguments: the pane, data (this is the selected object if there is a selection), x, y. It should return a menu. If it is not a function or a fbound symbol, it should be a menu, which is used directly. The **`:pane-menu`** mechanism is useful when the menu needs to be dependent on the location of the mouse inside the pane, or when each pane requires a unique menu. In other cases, the other mechanism is more useful.

2. If *pane-menu* is **`:default`** (this is the default value), CAPI calls the generic function **`make-pane-popup-menu`** with two arguments: the pane and its interface. The result should be a menu.

If the chosen mechanism does not produce a menu, the CAPI does not do anything in response to **`:post-menu`**.

The system definition of **`make-pane-popup-menu`** calls **`pane-popup-menu-items`** with the pane and the interface, and if this returns a non-nil list, it calls **`make-menu-for-pane`** to make the menu. You can define **`make-pane-popup-menu`** methods that specialize on your pane or interface classes, but in most cases it is more useful to add methods to **`pane-popup-menu-items`**. **`make-menu-for-pane`** is used to generate the menu, and it makes the menu such that by default all setup callbacks are done on the pane itself, rather than on the interface. **`make-pane-popup-menu`** is useful when the application needs a menu with the same items as the items on the popup menu, typically to add it to the menu bar.

In **`output-pane`**, you control the input behavior using the *input-model*. By default, the system assigns **`:post-menu`** and **`:keyboard-post-menu`** (**`Shift+F10`**) to a callback that raises a menu as described above, but your code can override this in the *input-model*.

Note: Accelerators are ignored in a *pane-menu*.

8.13 Displaying menus programmatically

You can programmatically display a menu by using **display-popup-menu** (which is used internally to raise the context menu). The menu that **display-popup-menu** displays can be any properly constructed **menu** object, for example:

```
(defun popup-animal-menu (animal interface)
  (let* ((items (list (string-append
                        "Get a picture of a " animal)
                       (string-append
                        "Send a postcard to " animal)))
         (menu (make-instance 'capi:menu :items items)))
    (capi:display-popup-menu menu :owner interface)))

(capi:contain (make-instance 'capi:list-panel
                             :items
                             '("zebra" "dog" "parrot")
                             :selection-callback
                             'popup-animal-menu))
```

Click on an item to see the menu.

You can use **popup-menu-force-popdown** to force a popup menu down (that is, make it disappear). This is useful for writing scripts that emulate user interactions.

8.14 The Application menu

This section is only applicable to LispWorks for Macintosh.

The CAPI includes an interface to the Application menu supporting standard macOS behaviors in your delivered LispWorks for Macintosh applications.

See these examples:

```
(example-edit-file "capi/applications/cocoa-application")
```

```
(example-edit-file "delivery/macos/single-window-application")
```

```
(example-edit-file "delivery/macos/multiple-window-application")
```

and the manual entries in the reference section, starting with **cocoa-default-application-interface**.

9 Adding Toolbars

You can add a toolbar for an interface using the **`interface`** initarg **`:toolbar-items`**. This creates a toolbar which is automatically positioned correctly in the window, which the user can customize, and which has platform-standard behavior such as folding on Cocoa. Such a toolbar is referred to as an *interface toolbar*.

You can also create toolbars using the **`toolbar`** class explicitly, and arrange them using layouts in the same way as other elements. This approach is used to implement buttons on a **`text-input-pane`** as seen in various tools in the LispWorks IDE such as the Class Browser, but you should note that it has some disadvantages. For more information see **9.9 Non-standard toolbars**.

Toolbar buttons typically have images. The examples in this chapter use three standard image identifiers. To run the example code that follows, first evaluate this form:

```
(setq file-images (list :std-file-new
                        :std-file-open
                        :std-file-save))
```

You also should define these callback functions before attempting any of the examples in this chapter:

```
(defun test-callback (data interface)
  (display-message "Data ~S in interface ~S"
                   data interface))

(defun print-callback (data interface)
  (declare (ignore data interface))
  (display-message "Print Something"))

(defun hello (data interface)
  (declare (ignore data interface))
  (display-message "Hello World"))
```

9.1 Creating a toolbar button

To create a toolbar button you can do:

```
(setf print-button
      (make-instance 'toolbar-button
                     :image :std-print
                     :text "Print Something"
                     :name :print-something))
```

You should supply *image*, *text* and *name*. This is because the user can customize the toolbar such that one (or all) of these appear, as described in **9.6 Modifying toolbars**.

A **`toolbar-button`** cannot be displayed directly. To include it in an interface toolbar, do:

```
(display
 (make-instance
  'interface
  :toolbar-items (list print-button)))
```

9.2 Creating a toolbar with several buttons

Let us create three more buttons:

```
(setf file-buttons
      (loop for image in file-images
            collect
            (make-instance 'toolbar-button
                           :image image
                           :name image
                           :text
                           (string-capitalize
                            (substitute #\Space #\-
                                        (string image))))))
```

and then include them along with the print button defined in **9.1 Creating a toolbar button**:

```
(display
 (make-instance
  'interface
  :toolbar-items (append file-buttons (list print-button))))
```

Remember that each instance of a toolbar button must not be used in more than one place at a time.

It is possible to include to include toolbar buttons which are not initially displayed, but which are available for the user to add. For the details, see **9.6 Modifying toolbars**.

9.2.1 Grouping toolbar buttons

The **`toolbar-component`** class lets you group related buttons together in a toolbar. This allows similar buttons to:

- Share properties such as callbacks.
- Be visually separated from other buttons in the toolbar.
- On Microsoft Windows, form a separately dockable group of items.

Toolbar components are actually choices similar to button panels. By default, their *interaction* is **`:single-selection`**.

We can amend our example using toolbar components to group the file buttons separately from the print button:

```
(display
 (make-instance
  'interface
  :toolbar-items (list
                  (make-instance 'toolbar-component
                                 :items file-buttons)
                  (make-instance 'toolbar-component
                                 :items (list print-button)))
  :visible-min-width 200))
```

9.2.2 Implicitly-created buttons

A **`toolbar-component`** may contain arbitrary Lisp objects as items. For each such object, a toolbar button is automatically created, using the appropriate elements of the component's *images*, *names*, *texts* and *tooltips* lists.

```
(display
 (make-instance
  'interface
```

```
:toolbar-items
(list (make-instance 'toolbar-component
                     :items file-images
                     :images file-images
                     :names file-images
                     :texts
                     (mapcar 'string-capitalize file-images)
                     :tooltips
                     (mapcar 'string-downcase file-images)
                     :selection-callback
                     (lambda (data interface)
                        (display-message "callback data ~S" data))
                     ))))
```

Rather than *selection-callback* above, you could supply *callbacks* to specify callback functions for each button.

9.3 Specifying the image for a toolbar button

There are several ways to supply the *image* for a toolbar button, including direct specification of an **image** object. The simplest approach is to use a symbol which is registered as an image identifier, including the pre-registered standard images, as in the preceding examples. For details of this and the other way to supply images, see **toolbar-button**.

You can, if desired, supply an alternative image which is displayed while the button is selected in a **:multiple-selection** component (see **9.7 Advanced toolbar features**), using the initarg *selected-image*.

9.3.1 Specifying images for a group of toolbar buttons

In a **toolbar-component** it is possible to specify images for the buttons by supplying an **image-set** as the *default-image-set*, along with integers in the *images* initarg specifying the index for the image of each button:

```
(display
 (make-instance
  'interface
  :toolbar-items
  (list
   (make-instance
    'toolbar-component
    :items '(1 2) :names '(1 2) :texts '("One" "Two")
    :images '(0 1)
    :default-image-set
    (make-general-image-set
     :image-count 5
     :id
     (gp:read-external-image
      (example-file
       "capi/elements/images/toolbar-radio-images.bmp")
      :transparent-color-index 7))))))
```

9.4 Specifying toolbar callbacks

Supply the *selection-callback* initarg to specify a callback for a toolbar button:

```
(setf print-button
      (make-instance 'toolbar-button
                     :image :std-print
                     :text "Print File"
                     :selection-callback 'print-callback))
```

You can also supply *selection-callback* for a **toolbar-component**. This specifies the same callback function for each button in the component.

To specify different callback functions for each button in a **toolbar-component**, either make the buttons explicitly as above, or supply the *callbacks* initarg.

9.4.1 Sharing toolbar callbacks with menu items

Where you want a toolbar button to perform the same command as a menu item, use the **:remapped** initarg.

remapped should match (by **cl:equalp**) the *name* of the **menu-item**:

```
(display
 (make-instance
  'interface
  :menu-bar-items
  (list
   (make-instance 'menu
                  :items
                  (list
                   (make-instance 'menu-item
                                  :name 'say-hello
                                  :data "Hello"
                                  :callback
                                  'test-callback))))
  :toolbar-items
  (list
   (make-instance 'toolbar-button
                  :image :std-file-new
                  :remapped 'say-hello))))
```

9.4.2 Other types of callback for a toolbar button

You can, if desired, supply a *retract-callback* which is called when the button is deselected in a **:multiple-selection** component. You can also make a button display a dropdown menu nearby. See **9.7 Advanced toolbar features** for the details.

9.5 Specifying tooltips for toolbar buttons

There are two ways to implement tooltips in an interface toolbar:

- Group the buttons in a **toolbar-component** and supply the **:tooltips** initarg. *tooltips* should be a list containing a string for each button in the component. For an example of this see:

  ```
  (example-edit-file "capi/applications/simple-symbol-browser")
  ```

- Alternatively you can implement a tooltip for each **toolbar-button** exactly as for collections and so on as described in **3.12.2 Tooltips for collections, elements and menu items**. Supply *help-key* for the **toolbar-button** and *help-callback* for the **interface**, as follows:

  ```
  (setf print-button
        (make-instance 'toolbar-button
                       :image :std-print
                       :text "Print Something"
                       :help-key 'foo))

  (defun do-help (interface pane type help-key)
  ```

```
    (when (eq type :tooltip)
      (when (eq help-key 'foo)
        "Tooltip help")))

  (display
   (make-instance
    'interface
    :toolbar-items
    (list print-button)
    :help-callback 'do-help))
```

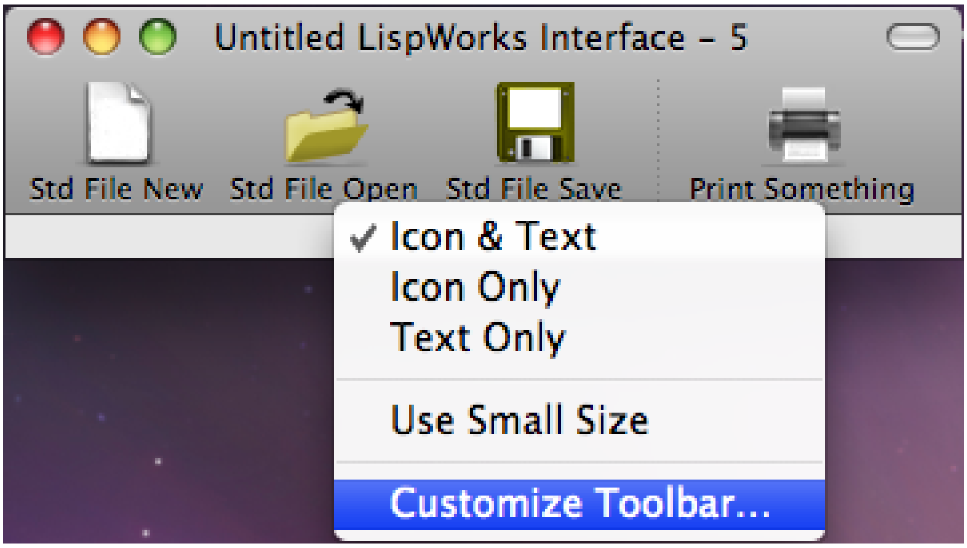
9.6 Modifying toolbars

An interface toolbar can be customized by the user. It can also be manipulated programmatically.

9.6.1 User-customization of toolbars

The user can change *toolbar state*, that is the set of visible toolbar items, their order and their appearance. The user does this via the context menu on the toolbar. This menu includes commands to display the button images or titles (or both), and a **Customize Toolbar...** command to alter the set of items, including separators and spaces, and the order in which the items appear.

The toolbar context menu



To raise the customization dialog programmatically, call **`interface-customize-toolbar`**.

You can supply a default toolbar state in the initarg *default-toolbar-states*. This is used when the user presses the **Default** button in the **Customize Toolbar** dialog. You can read this value with **`interface-default-toolbar-states`**.

You can control the initial toolbar state by supplying the initarg *toolbar-states*.

9.6.2 Changing an interface toolbar programmatically

You can read and change the *toolbar-states* slot programmatically. Its value should be a *toolbar state plist*.

Be aware that *toolbar-states* may not be the same each time you read it, because the user may have changed it as described in **9.6.1 User-customization of toolbars**.

For the details, see the accessor **`interface-toolbar-state`**.

9.7 Advanced toolbar features

9.7.1 Toolbar items other than buttons with images

A **toolbar-component**, a **toolbar** or the interface toolbar may also contain CAPI panes as items, which will appear within the toolbar. This is typically used with **text-input-pane**, **option-pane**, and **text-input-choice**. Each pane should have *toolbar-title* (see **simple-pane**) specified, to provide the text that is shown for the toolbar item:

```
(display
 (make-instance
  'interface
  :toolbar-items (list
                  (make-instance 'toolbar-component
                                 :items (list print-button))
                  (make-instance 'text-input-pane
                                 :text "Text Input Pane"
                                 :visible-min-width :text-width
                                 :toolbar-title "Text Input Pane")
                  (make-instance 'text-input-choice
                                 :items
                                 (list "Text Input Choice1"
                                       "Text Input Choice2")
                                 :visible-min-width :text-width
                                 :toolbar-title "Text Input Choice")
                  (make-instance 'option-pane
                                 :items
                                 (list "Option Pane1"
                                       "Option Pane2")
                                 :visible-min-width :text-width
                                 :toolbar-title "Option Pane")
                  )
  :visible-min-width 500))
```

Note: Some platforms may not recommend placing text input panes and so on in a toolbar. You may wish to consult the appropriate user interface guidelines before adding such a toolbar in your application.

Note: Each **toolbar-button** or **simple-pane** in the *toolbar-items* list (including those within a **toolbar-component**) should have a *name* that is not **cl:eql** to any other item in the list. These names are needed to support **:items** in **interface-toolbar-state** and the **:toolbar-states** initarg.

Toolbar buttons can display text, which should be in the *data* or *text* slot inherited from **item**. You can specify whether text and/or image is displayed, using **:display** in the *toolbar-states* initarg or **interface-toolbar-state**.

9.7.2 Alternative interaction in a toolbar

You can make a **toolbar-component** with *interaction* **:multiple-selection** and then each of its buttons may have a *retract-callback* which is called when the user clicks a selected button to deselect it.

9.7.3 Toolbar buttons with menus

You can add a menu to a toolbar button, which is displayed via a separate smaller button next to the main button. To do this, supply *dropdown-menu* or *dropdown-menu-function*. See **toolbar-button** for the details.

9.8 Disabling toolbar items

To disable a toolbar button you can set its *enabled* slot to **`nil`**. Alternatively supply it with a suitable *enabled-function*. For more information about this, see **`toolbar-object`**.

You can disable and enable a **`toolbar-component`** in the same way.

9.9 Non-standard toolbars

You can create toolbars using the **`toolbar`** class explicitly, and arrange them like other elements, using layouts. This approach differs from using an *interface toolbar* as described in the preceding sections of this chapter. Note that, while it allows you some flexibility this approach can produce non-standard appearance, does not support user-customization, and does not support folding on Cocoa. Other than this, non-standard toolbars support all the features described in the preceding sections of this chapter, and additionally:

- You can disable and enable a **`toolbar`** using its *enabled* or *enabled-function* slot.
- There are two further options for a button with a dropdown menu.

  It can be merged with the separate smaller button such that it displays only the menu and does not respond to its *selection-callback*.

  Alternatively, it can display the menu only after being pressed down for a while, and respond to the *selection-callback* when pressed only briefly. In this case the smaller button does not appear.

  See **`toolbar-button`** for the details.
- You can make a toolbar button which displays an **`interface`** (and does not respond to its *selection-callback*) by supplying *popup-interface*.

There is an example here:

```
(example-edit-file "capi/elements/toolbar")
```

9.9.1 Changing a non-standard toolbar dynamically

The best way to change a non-standard toolbar is to use a **`switchable-layout`**. Include a **`toolbar`** instance in each of two or more child layouts, of which only one is visible at a time.

There is an example here:

```
(example-edit-file "capi/layouts/switchable")
```

10 Dialogs: Prompting for Input

A dialog is a window that is displayed transiently to interact with the user. While a dialog is on screen it is placed in front of other windows and user input is directed to it. Dialogs are used for interactions that are relatively rare, and so do not deserve a permanent place on the screen, and for alerting the user about something that they need to be aware of. For example, when an application needs to know where to save a file, it typically prompts with a file dialog. If there is a problem during saving the file, it would normally alert the user by some other dialog.

Dialogs can also be cancelled, meaning that the application should cancel the current operation. In order to let you know whether or not the dialog was cancelled, CAPI dialog functions always return two values. The first value is the return value itself, and the second value is **`t`** if the dialog returned normally and **`nil`** if the dialog was cancelled.

On Cocoa you can control whether a CAPI dialog is application-modal or window-modal. In the latter case the user can interact with the application's other windows while the dialog is on screen.

The CAPI provides both a large set of predefined dialogs and the means to create your own. This chapter takes you through some example uses of the predefined dialogs, and then shows you how to create custom built dialogs.

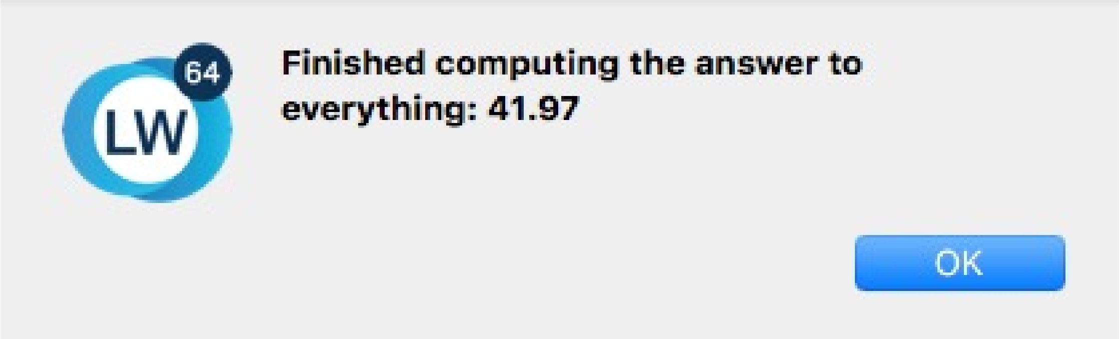
The last section briefly describes a way to get input for completions via a special non-modal window.

10.1 Some simple dialogs

The simplest form of dialog is a message dialog, which is used to inform the user of some event, typically the end of a long operation. You can use **`display-message`** for this.

```
(display-message
 "Finished computing the answer to everything: ~a" 41.97)
```

A message dialog



When you want to ensure that the messages dialog is associated with (that is, owned by) a specific pane, you can use **`display-message-for-pane`**. There is also **`prompt-with-message`**, which can be used for displaying the message in a window-modal sheet on Cocoa.

```
(display-message
             "This function is ~S"
             'display-message)
```

A second message dialog

Another simple dialog asks the user a question and returns **`t`** or **`nil`** depending on whether the user has chosen yes or no. This function is **`confirm-yes-or-no`**.

```
(confirm-yes-or-no
    "Do you own a pet?")
```

A message dialog prompting for confirmation

For more control over such a dialog, use the function **`prompt-for-confirmation`**.

10.2 Prompting for values

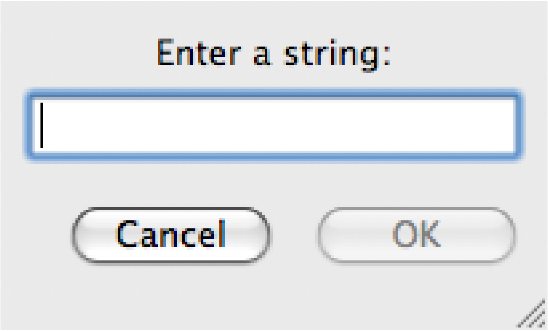
The CAPI provides a number of different dialogs for accepting values from the user, ranging from accepting strings to accepting whole Lisp forms to be evaluated.

10.2.1 Prompting for strings

The simplest of the CAPI prompting dialogs is **`prompt-for-string`** which returns the string you enter into the dialog.

```
(prompt-for-string
 "Enter a string:")
```

A dialog prompting for a string



An initial value can be placed in the dialog by specifying the keyword argument **`:initial-value`**.

10.2.2 Prompting for numbers

The CAPI also provides a number of more specific dialogs that allow you to enter other types of data. For example, to enter an integer, use the function **`prompt-for-integer`**. Only integers are accepted as valid input for this function.

```
(prompt-for-integer
 "Enter an integer:")
```

There are a number of extra options which allow you to specify more strictly which integers are acceptable. Firstly, there are two arguments **`:min`** and **`:max`** which specify the minimum and maximum acceptable integers.

```
(prompt-for-integer
 "Enter an integer in the inclusive range [10,20]:"
 :min 10 :max 20)
```

If this does not provide enough flexibility you can specify a function that validates the result with the keyword argument **`:ok-check`**. This function is passed the current value and must return non-nil if it is a valid result.

```
(prompt-for-integer
 "Enter an odd integer:"
 :ok-check 'oddp)
```

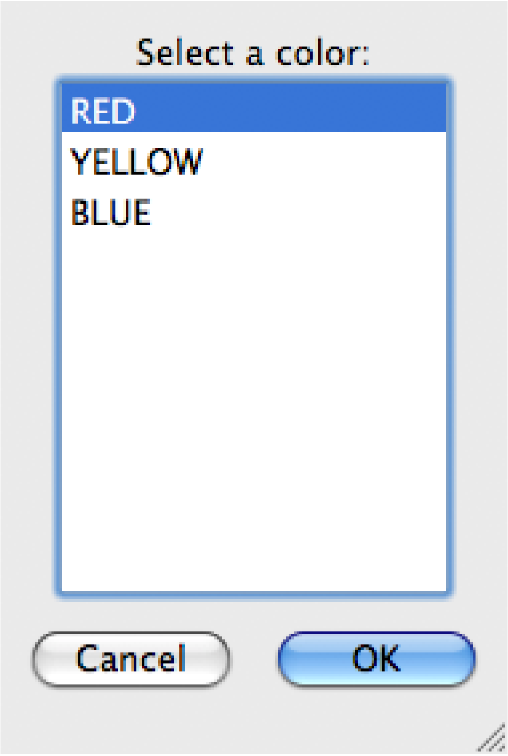
Try also the function **`prompt-for-number`**.

10.2.3 Prompting for an item in a list

If you would like the user to select an item from a list of items, the function **`prompt-with-list`** should handle the majority of cases. The simplest form just passes a list to the function and expects a single item to be returned.

```
(prompt-with-list
 '(:red :yellow :blue)
 "Select a color:")
```

A dialog prompting for a selection from a list



You can also specify the interaction style that you would like for your dialog, which can be any of the interactions accepted by a choice. The specification of the interaction style to this choice is made using the keyword argument **`:interaction`**:

```
(prompt-with-list
 '(:red :yellow :blue)
 "Select a color:"
 :interaction :multiple-selection)
```

By default, the dialog is created using a **`list-panel`** to display the items, but the keyword argument **`:choice-class`** can be specified with any choice pane. Thus, for instance, you can present a list of buttons.

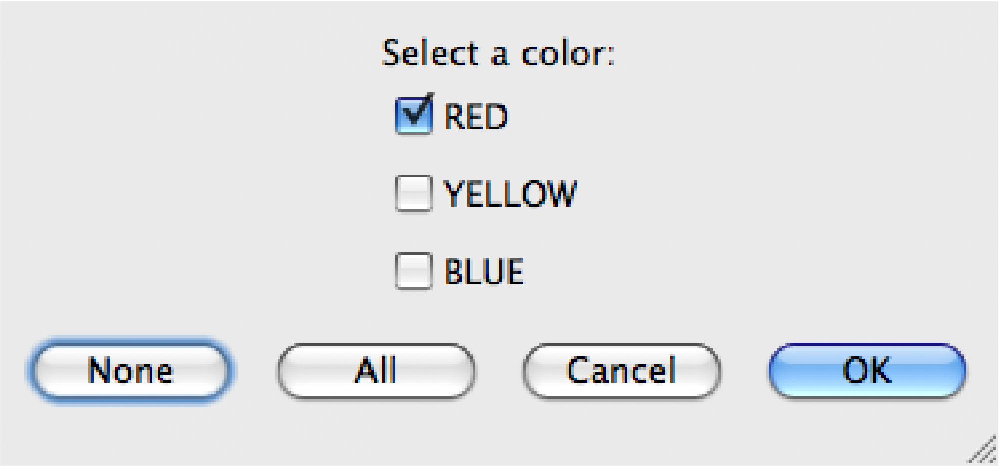
```
(prompt-with-list
 '(:red :yellow :blue)
 "Select a color:"
 :interaction :multiple-selection
 :choice-class 'button-panel)
```

Selection from a button panel

Finally, as with any of the prompting functions, you can specify additional arguments to the pane that has been created in the dialog. Thus to create a column of buttons instead of the default row, use:

```
(prompt-with-list
 '(:red :yellow :blue)
 "Select a color:"
 :interaction :multiple-selection
 :choice-class 'button-panel
 :pane-args
 '(:layout-class column-layout))
```

Selection from a column of buttons



There is a more complex example in:

```
(example-edit-file "capi/choice/prompt-with-buttons")
```

10.2.4 Prompting for files

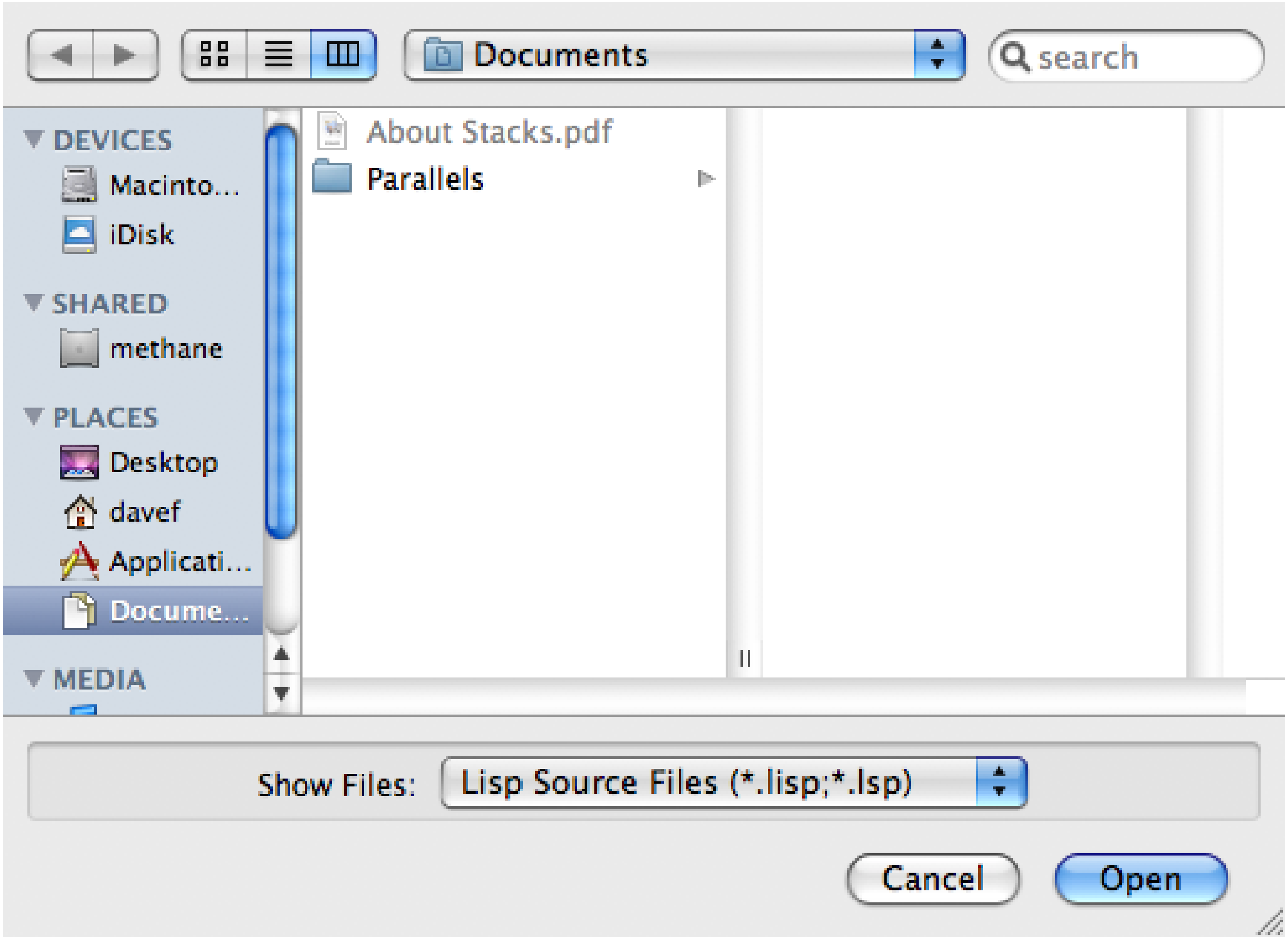
To prompt for a file, use the function **prompt-for-file**:

```
(prompt-for-file
 "Enter a file:")
```

You can also specify a starting pathname:

```
(prompt-for-file
  "Enter a filename:"
  :pathname (get-temp-directory))
```

Selection of a file



Try also the function **prompt-for-directory**.

10.2.5 Prompting for fonts

To obtain a **gp:font** object from the user call **prompt-for-font**.

10.2.6 Prompting for colors

To obtain a color specification from the user call **prompt-for-color**.

10.2.7 Prompting for Lisp objects

The CAPI provides a number of dialogs specifically designed for creating Lisp aware applications. The simplest is the function **prompt-for-form** which accepts an arbitrary Lisp form and optionally evaluates it.

```
(prompt-for-form
 "Enter a form to evaluate:"
 :evaluate t)
```

```
(prompt-for-form
 "Enter a form (not evaluated):"
 :evaluate nil)
```

Another useful function is **prompt-for-symbol** which prompts the user for an existing symbol. The simplest usage accepts any symbol, as follows:

```
(prompt-for-symbol
 "Enter a symbol:")
```

If you have a list of symbols from which to choose, then you can pass **prompt-for-symbol** this list with the keyword argument **:symbols**.

Finally, using **:ok-check** you can accept only certain symbols. For example, to only accept a symbol which names a class, use:

```
(prompt-for-symbol
 "Enter a class-name symbol:"
 :ok-check #'(lambda (symbol)
               (find-class symbol nil)))
```

Cocoa programmers will notice that the dialog sheet displayed by this form prevents input to other LispWorks windows while it is displayed. For information about creating dialog sheets which are not application-modal, see **10.3 Window-modal Cocoa dialogs**.

10.3 Window-modal Cocoa dialogs

By default, CAPI dialogs on Cocoa use sheets which are application-modal. This means that the application does not allow the user to interact with its other windows until the sheet is dismissed.

This section describes how to create CAPI dialogs which are window-modal on Cocoa. This is done with portable code, so Windows, GTK+ and Motif programmers may wish to code their CAPI dialogs as described in this section, which would ease a future port to the Cocoa GUI.

10.3.1 The :continuation argument

All CAPI dialog functions take a keyword argument *continuation*. This is a function which is called with the results of the dialog.

You do not need to construct the continuation argument yourself, but rather call the dialog function inside **with-dialog-results**.

10.3.2 A dialog which is window-modal on Cocoa

To create a dialog which is window-modal on Cocoa, call the dialog function inside the macro **with-dialog-results** as in this example:

```
(with-dialog-results (symbol okp)
    (prompt-for-symbol
     "Enter a class-name symbol:"
     :ok-check #'(lambda (symbol)
                   (find-class symbol nil)))
  (when okp
    (display-message "symbol is ~S" symbol)))
```

On Microsoft Windows, GTK+ and Motif this displays the dialog, calls **display-message** when the user clicks **OK**, and then returns. The effect is no different to what you saw in **10.2.7 Prompting for Lisp objects**.

On Cocoa, this creates a sheet and returns. **display-message** is called when the user clicks **OK**. The sheet is window-modal, unlike the sheet you saw in **10.2.7 Prompting for Lisp objects**.

For more details, see the manual page for **`with-dialog-results`**.

10.4 Dialog Owners

When a dialog appears, it should be "owned" by some window. The main effect of this "ownership" is that the dialog is always in front of the owner window. When either the dialog or the owner is raised, the other follows.

All CAPI functions which display a dialog allow you to specify the owner.

10.4.1 The default owner

When a dialog is displayed and the owner is not supplied or is given as **`nil`**, the CAPI tries to identify the appropriate owner. In particular, in the case where a dialog pops up in a process in which a CAPI interface is displayed, by default the CAPI uses this interface as the owner window. This case covers most situations.

10.4.2 Specifying the owner

If the default is not appropriate, then the programmer needs to supply the owner. This *owner* argument can be any CAPI pane that is currently displayed, and the top level interface of the pane is used as the actual owner. A CAPI pane owner must be running in the current thread (see the *process* argument to **`display`**). Creating cross-thread ownership can lead to deadlocks.

The *owner* can also be a **`screen`** object, which tells the system on which screen to put the dialog, but none of the windows will be the dialog's owner.

The *owner* can be supplied by the keyword argument **`:owner`** in functions such as **`display-dialog`** and **`print-dialog`**. Other functions such as **`prompt-for-string`** and **`prompt-for-file`** can be supplied an owner in the **`:popup-args`** list as a pair **`:owner`** *owner*.

10.5 Creating your own dialogs

The CAPI provides a number of built-in dialogs which should cover the majority of most programmers' needs. However, there is always the occasional need to create custom built dialogs, and the CAPI makes this very simple, using the function **`popup-confirmer`** which displays any CAPI interface as a dialog, and the functions **`exit-confirmer`** to return from such a dialog.

10.5.1 Using popup-confirmer

The function **`popup-confirmer`** is a higher level function provided to add the standard buttons to dialogs. In order to create a dialog using **`popup-confirmer`**, all you need to do is to supply a pane to be placed inside the dialog along with the buttons and the title. The function also expects a title, like all of the prompter functions described earlier.

```
(popup-confirmer
 (make-instance
  'text-input-pane
  :callback-type :data
  :callback 'exit-dialog)
 "Enter a string")
```

Since interfaces and layouts are panes too, the *pane* argument to **`popup-confirmer`** can be a layout or an interface, and often it is. Layouts are used for simple combinations of panes, and interfaces are used for complex dialogs. All the dialogs in the LispWorks IDE which are not either native, just a message or asking for a single item of input are interfaces displayed by

popup-confirmer. As an example, you can load the Othello example file:

```
(example-edit-file "capi/applications/othello")
```

which defines an interface **othello-board**. You can then run this as a dialog:

```
(capi:popup-confirmer
 (make-instance 'othello-board) "Play Othello")
```

Note that it works as usual, except that the menubar is not displayed.

Here is a simple example using a layout to ask the user for five strings:

```
(let* ((panes
        (loop repeat 5
              collect
              (make-instance 'capi:text-input-pane)))
       (layout (make-instance 'capi:column-layout
                              :description panes)))
  (multiple-value-bind (res okp)
      (capi:popup-confirmer layout
                            "Enter some strings")
    (declare (ignore res))
    (when okp
      (loop for pane in panes
            collect
            (capi:text-input-pane-text pane)))))
```

An interface intended for display by **popup-confirmer** can also be displayed by **display** (not at the same time), in which case it is just another window. That is especially useful during development of your dialog code, because you can then work on the callbacks while the interface is displayed.

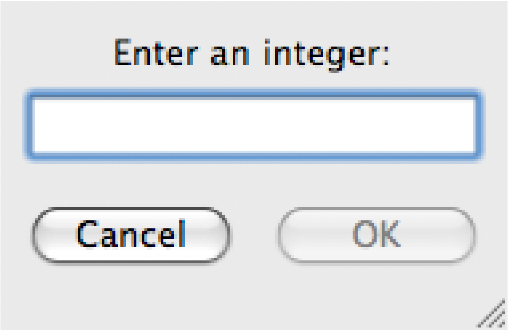
A common thing to want to do with a dialog is to get the return value from some state in the pane specified. For instance, in order to create a dialog that prompts for an integer the string entered into the **text-input-pane** would need to be converted into an integer. It is possible to do this once the dialog has returned, but **popup-confirmer** has a more convenient mechanism. The function provides a keyword argument, **:value-function**, which gets passed the pane, and this function should return the value to return from the dialog. It can also indicate that the dialog cannot return by returning a second value which is non-nil.

In order to do this conversion, **popup-confirmer** provides an alternative exit function to the usual **exit-dialog**. This is called **exit-confirmer**, and it does all of the necessary work on exiting.

You now have enough information to write a primitive version of **prompt-for-integer**.

```
(defun text-input-pane-integer (pane)
 (let* ((text
         (text-input-pane-text pane))
        (integer
         (parse-integer
          text
          :junk-allowed t)))
   (or (and (integerp integer) integer)
       (values nil t))))


(popup-confirmer
  (make-instance
   'text-input-pane
   :callback 'exit-confirmer)
  "Enter an integer:"
  :value-function 'text-input-pane-integer)
```

A example using **popup-confirmer**



Note that the dialog's **OK** button never becomes activated, yet pressing **Return** once you have entered a valid integer *will* return the correct value. This is because the **OK** button is not being dynamically updated on each keystroke in the **text-input-pane** so that it activates when the pane contains a valid integer. The activation of the **OK** button is recalculated by the function **redisplay-interface**, and the CAPI provides a standard callback, **:redisplay-interface**, which calls this as appropriate.

Thus, to have an **OK** button that becomes activated and deactivated dynamically, you need to specify the *change-callback* for the **text-input-pane** to be **:redisplay-interface**.

```
(popup-confirmer
 (make-instance
  'text-input-pane
   :change-callback :redisplay-interface
  :callback 'exit-confirmer)
 "Enter an integer:"
 :value-function 'text-input-pane-integer)
```

Note that the **OK** button now changes dynamically so that it is only ever active when the text in the **text-input-pane** is a valid integer.

Note that the **Escape** key activates the **Cancel** button - this too was set up by **popup-confirmer**.

The next thing that you might want to do with your integer prompter is to make it accept only certain values. For instance, you may only want to accept negative numbers. This can be specified to **popup-confirmer** by providing a validation function with the keyword argument **:ok-check**. This function receives the potential return value (the value returned by the value function) and it must return non-nil if that value is valid. Thus to accept only negative numbers we could pass **minusp** as the **:ok-check**.

```
(popup-confirmer
 (make-instance
  'text-input-pane
  :change-callback :redisplay-interface
  :callback 'exit-confirmer)
 "Enter an integer:"
 :value-function 'text-input-pane-integer
 :ok-check 'minusp)
```

10.5.2 Using display-dialog

popup-confirmer creates an interface (of an internal class) around the pane that you give it which displays the pane and the buttons it adds, and then calls **display-dialog** to actually display it. If you have an interface and do not want any of the buttons, you can call **display-dialog** directly.

display-dialog takes an interface (unlike **popup-confirmer**, which can take any pane) and displays it as a dialog. The keyword arguments can be used to control the exact behavior. You can use **exit-dialog** and **abort-dialog** to dismiss the dialog programmatically.

10.5.3 Modal and non-modal dialogs

By default **`popup-confirmer`** and **`display-dialog`** create modal dialog windows which prevent input to other application windows until they are dismissed by the user clicking on a button or another appropriate gesture. You can change this behavior by passing the *modal* keyword argument.

10.5.4 Getting the current dialog

The function **`current-popup`** can be used to find the current popup pane, if there is any, and is useful inside callbacks.

The function **`current-dialog-handle`** returns the "handle" of the dialog in the underlying GUI system, which may be useful in some circumstances.

10.6 In-place completion

'In-place completion' allows the user to select from a list of possible completions displayed in a special non-modal window which appears in front of an input pane (such as an **`editor-pane`** or a **`text-input-pane`**) but does not grab the input focus.

To raise this special window and select a completion from it, the user invokes certain keyboard gestures including **`Up`**, **`Down`** and **`Return`**. The full set of keys for operations on an in-place completion window are described **10.6.1 In-place completion user interface**. The user can also continue typing her input in which case the list of possible completions is updated to reflect the text in the input pane.

10.6.1 In-place completion user interface

This section describes the user interface of in-place completion.

In-place completion is available in the LispWorks IDE, in the Editor tool and also in tools that ask for a named object such as the Class Browser and the Generic Function Browser. Set the **Preferences... Environment > General > Use in-place completion** option to use in-place completion in the LispWorks IDE, and see *LispWorks IDE User Guide* for further details.

In-place completion is also available to you to use in your CAPI applications. You may wish to adapt the remainder of this section for your end-user documentation. See **10.6.2 Programmatic control of in-place completion** for information on how to implement it.

10.6.1.1 Invoking in-place completion in text-input-pane and editor-pane

In a **`text-input-pane`** that supports in-place completion, any of the gestures **`Up`**, **`Down`**, **`PageUp`**, and **`PageDown`** invokes the in-place completion unless it is already displayed.

In an **`editor-pane`**, completion commands invoke in-place completion by default, though you can make them use dialogs instead by setting **`editor:*use-in-place-completion*`** to **`nil`**.

There are several Editor commands that invoke in-place completion unconditionally:

`Abbreviated in-place Complete Symbol`

Completes the symbol before the point, taking the string as abbreviation.

`In-Place Complete Symbol`

Completes the symbol before the point.

In-Place Complete Input

Echo Area: Complete the input in the echo area. For file input, does file completion.

In-Place Expand File Name

Expand the file name at the current point.

In-Place Expand File Name with space

Expand the file name at the current point, allowing spaces.

See the *Editor User Guide* for information on binding these commands to keyboard gestures. See **call-editor** for information on calling them from CAPI.

10.6.1.2 Keyboard input handling while the in-place window is displayed

Keyboard input while the in-place window is displayed goes to the input pane, but some of the input gestures are redirected to the in-place window. By default, the following gestures are redirected:

Up, **Down**, **PageUp**, **PageDown**

Change the selection in the list of completions in the obvious way.

Return

Perform the completion using the current selected item in the list. In non-file-completion, or in file-completion when the item is not a directory, the in-place window disappears. In file-completion when the selected item is a directory, the in-place window changes to display the list of files in the completed directory.

Escape

Causes the in-place window to disappear, without doing anything else. Note that if the text in the input pane was edited while the in-place window was displayed, these edits are not undone.

Control+Return

Toggles the filter.

Control+Shift+Return

Toggles redirection of characters to the filter. A filter is a **text-input-pane** which filters the list of completions based on its contents. While the filter is on, the list of completions shows only the completions that match the filter.

While the filter is visible and enabled, all character input plus **Backspace** are redirected to the filter. The filter can be disabled by **Control+Shift+Return**, which means it still filters, but characters go to the the input pane.

The functionality of the in-place completion filter is the same as the standard filter for **list-panel**. For a full description of the pattern matching see "Regular expression searching" in the *Editor User Guide*.

Control+Shift+R, **Control+Shift+E**, **Control+Shift+C**

Change the setting in the filter.

Other keyboard input goes to the input pane.

While the filter is off (the default), or when the filter is on and disabled, plain characters go to the input pane, and hence change the text in it.

When the filter is on and is enabled, plain characters go to the filter.

10.6.1.3 Performing a completion

In a **text-input-pane**, performing a completion means replacing part of the text in the pane by the selected completion. In a file-completion, only the last part of the text (from the last directory separator) is replaced.

If a **text-input-pane** was made with *complete-do-action* true, once the completion was performed, if it is not file-completion and the completion is a directory, the callback of the pane is invoked.

In an **editor-pane**, while the in-place window is displayed, the editor highlights the part of the text that will be replaced. In non-file-completion it is the beginning of the "symbol", as seen by the editor, and the end of the "symbol". In a file-completion it is the part of the filename after the last directory separator.

Performing the completion in an **editor-pane** means replacing the highlighted text by the selected completion. The replacement is done as a single separate operation (for example **Undo** will undo the replacement separately from any previous changes).

10.6.1.4 Interaction while the in-place window is displayed

Any operation that affects the text between the start of the relevant text (this is the start in a **text-input-pane**, and the highlighted area in an **editor-pane**) and the current cursor causes the in-place window to recompute the possible completions and display the new list. These operations include not only actual changes to the text, but also cursor movement.

In an **editor-pane**, if the insertion point moves out of the highlighted area then the in-place window goes away.

If the input pane loses the focus, the in-place window goes away, except on Motif.

10.6.2 Programmatic control of in-place completion

You can add in-place completion to your application as described in this section.

10.6.2.1 Text input panes

A **text-input-pane** will do in-place completion if you pass either of these initargs:

:file-completion with value **t** or a pathname designator, or:

:in-place-completion-function with value a suitable function designator.

You can add a filter to the in-place window by passing the initarg **:in-place-filter**. Additionally you can control the functionality for file completion by passing **:directories-only** and **:ignore-file-suffices**. The keyword arguments **:complete-do-action** and **:gesture-callbacks** also interact with in-place completion.

The in-place completion can be invoked explicitly for a **text-input-pane** by calling **text-input-pane-in-place-complete**.

See the manual page for **text-input-pane** for details.

10.6.2.2 Editor panes

An **editor-pane** does in-place completion when your code calls the function **editor:complete-in-place**.

10.6.2.3 Other CAPI panes

You can also implement in-place completion on arbitrary CAPI panes by calling **`prompt-with-list-non-focus`**.

11 Defining Interface Classes - top level windows

Interface classes (subclasses of **interface**) are (mainly) used to define top level windows and the components inside them. Normally, each kind of a window in an application is specified by a different interface class. Complex dialogs are also typically presented using an interface class.

An interface class can also be used to create a component made of several elements. This is especially useful when these elements need to interact, because the syntax of **define-interface** makes it easier to refer to elements in the interface. To distinguish between this usage and the more typical case where an interface instance corresponds to a window, the latter case is referred to as a "top level interface" (also "top level window"). The parent of a top level interface is a **screen** (or **document-container** inside MDI on Microsoft Windows) rather than another pane.

An interface class is defined by the macro **define-interface** (normally, **cl:defclass** inheriting from an interface class works too). **define-interface** is an extension of **cl:defclass** with additional options for specifying display elements. After an interface class is defined it can be used to display a window or a dialog by calling **display** or **display-dialog** on an instance of it. For example:

```
(capi:define-interface my-interface ()
  ()
  (:panes (my-display-pane capi:display-pane :text "Some text"))
  (:default-initargs :title "My title"))

(capi:display (make-instance 'my-interface))
```

11.1 The define-interface macro

The macro **define-interface** is used to define subclasses of **interface**, the superclass of all CAPI interface classes.

It is an extension to **defclass**, which provides the functionality of that macro as well as the specification of the panes, layouts, and menus from which an interface is composed. It takes the same arguments as **defclass**, and supports the additional options **:panes**, **:layouts**, **:menus**, and **:menu-bar**.

If you specify **:panes** but no **:layouts**, then on creating your interface the CAPI will create a **column-layout** and arrange the panes in it in the order they are defined. For real applications you will need some control over how the panes are laid out, and this is supplied via the **:layouts** option.

Each component of the interface is named in the code, and a slot of that name is added to the class created. When an instance of the class is made, each component is created automatically and placed in its slot.

To access a pane, layout or menu in an instance of your interface class you can define an accessor, like the **viewer** pane in **11.3 Adapting the example**, or simply use **with-slots**.

When defining a component, you can use other components within the definition simply by giving its name. You can refer to the interface itself by the special name **capi:interface**.

There are examples using **define-interface** in:

```
(example-edit-file "capi/applications/pong")
```

```
(example-edit-file "capi/applications/othello")
```

11.2 An example interface

Here is a simple example of interface definition done with **define-interface**:

```
(define-interface demo ()
  ()
  (:panes
   (page-up push-button
            :text "Page Up")
   (page-down push-button
              :text "Page Down")
   (open-file push-button
              :text "Open File"))
  (:layouts
   (row-of-buttons row-layout
                   '(page-up page-down open-file)))
  (:default-initargs :title "Demo"))
```

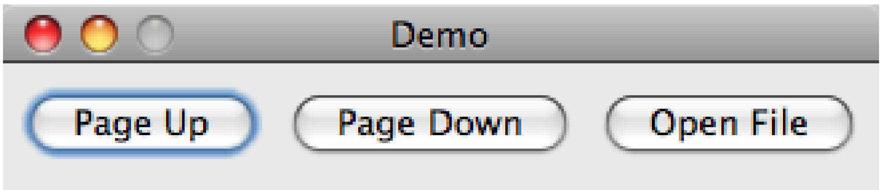
An instance of this interface can be displayed as follows:

```
(display (make-instance 'demo))
```

At the moment the buttons do nothing, but they will eventually do the following:

- **Open File** will bring up a file prompter and allow you to select a filename from a directory. Later on, we will add an editor pane to display the chosen file's contents.
- **Page Down** will scroll downwards so that you can view the lower parts of the file that cannot be seen initially.
- **Page Up** will scroll upwards so that you can return to parts of the file seen before.

A demonstration of a CAPI interface



Later on, we will specify callbacks for these buttons to provide this functionality.

The **(:default-initargs :title "Demo")** part at the end is necessary to give the interface a title. If no title is given, the default name is "Untitled CAPI Interface".

11.2.1 How the example works

Examine the **define-interface** form to see how this interface was built. The first part of this form is shown below:

```
(define-interface demo ()
   ()
```

This part of the macro is identical to **defclass** — you provide:

- The name of the interface class being defined.

- The superclasses of the interface (defaulting to **interface**).
- The slot descriptions.

The interesting part of the **define-interface** form occurs after these **defclass**-like preliminaries, where it lists the elements that define the interface's appearance. Here is the **:panes** part of the definition:

```
(:panes
 (page-up push-button
          :text "Page Up")
 (page-down push-button
            :text "Page Down")
 (open-file push-button
            :text "Open File"))
```

Two arguments — the name and the class — are required to produce a pane. You can supply slot values as you would for any CLOS object.

The **:panes** list specifies panes that are made when the interface is made. However it does not specify which panes are displayed: that is controlled dynamically by the interface's layout which may contain all, some or none of the panes in the **:panes** list. The interface may also display other panes that are made explicitly, though this is less common.

Here is the **:layouts** part of the definition:

```
(:layouts
 (row-of-buttons row-layout
                 '(page-up page-down open-file)))
```

Three arguments — the name, the class, and any child layouts — are required to produce a layout. Notice how the children of the layout are specified by using their component names.

The interface information supplied in this section is a series of specifications for panes and layouts. It could also specify menus and a menu bar. In this case, three buttons are defined. The layout chosen is a row layout, which displays the buttons side by side at the top of the pane.

11.3 Adapting the example

The **:panes** and **:layouts** keywords can take a number of panes and layouts, each specified one after the other. By listing several panes, menus, and so on, complicated interfaces can be constructed quickly.

To see how simply this is done, let us add an editor pane to our interface. We need this to display the text contained in the file chosen with the **Open File** button.

The editor pane needs a layout. It could be added to the **row-layout** already built, or another layout could be made for it. Then, the two layouts would have to be put inside a third to contain them (see **6 Laying Out CAPI Panes**).

The first thing to do is add the editor pane to the panes description. The old panes description read:

```
(:panes
 (page-up push-button
          :text "Page Up")
 (page-down push-button
            :text "Page Down")
 (open-file push-button
            :text "Open File"))
```

The new one includes an editor pane named **viewer**.

```
(:panes
 (page-up push-button
          :text "Page Up")
 (page-down push-button
            :text "Page Down")
 (open-file push-button
            :text "Open File")
 (viewer editor-pane
         :title "File:"
         :text "No file selected."
         :visible-min-height '(:character 8)
         :reader viewer-pane))
```

This specifies the editor pane, with a stipulation that it must be at least 8 characters high. This allows you to see a worthwhile amount of the file being viewed in the pane.

Note the use of **`:reader`**, which defines a reader method for the interface which returns the editor pane. Similarly, you can also specify writers or accessors. If you omit accessor methods, it is still possible to access panes and other elements in an interface instance using **`with-slots`**.

The interface also needs a layout containing the editor pane along with the buttons. The old layouts description read:

```
(:layouts
 (row-of-buttons row-layout
                 '(page-up page-down open-file)))
```

The new one reads:

```
(:layouts
 (main-layout column-layout
              '(row-of-buttons viewer))
 (row-of-buttons row-layout
                 '(page-up page-down open-file))
 )
```

This encapsulates the new pane **`viewer`** into a **`column-layout`** called **`main-layout`**. This is used as the default layout, specified by setting the **`:layout`** initarg to **`main-layout`** in the **`:default-initargs`** section. If there is no default layout specified, uses the first one listed.

By putting the layout of buttons and the editor pane in a column layout, their relative position has been controlled: the buttons appear in a row above the editor pane.

The code for the new interface is now as follows:

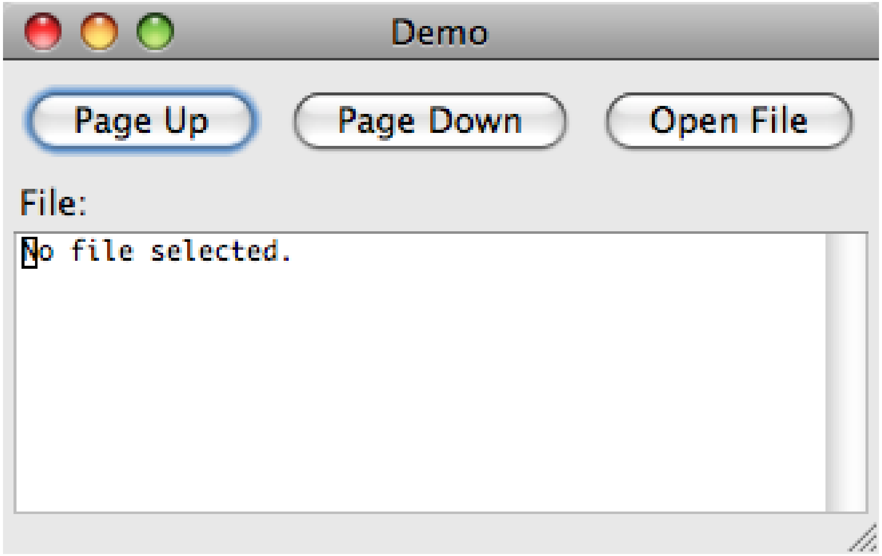
```
(define-interface demo ()
  ()
 (:panes
  (page-up push-button
           :text "Page Up")
  (page-down push-button
             :text "Page Down")
  (open-file push-button
             :text "Open File")
  (viewer editor-pane
          :title "File:"
          :text "No file selected."
          :visible-min-height '(:character 8)
          :reader viewer-pane))
 (:layouts
   (main-layout column-layout
                '(row-of-buttons viewer))
   (row-of-buttons row-layout
```

```
                  '(page-up page-down open-file)))
  (:default-initargs :title "Demo"))
```

Displaying an instance of the interface by entering the line of code below produces the window in **A CAPI interface with editor pane**:

```
(display (make-instance 'demo))
```

A CAPI interface with editor pane



11.3.1 Adding menus

To add menus to your interface you must first specify the menus themselves, and then a menu bar of which they will be a part.

Let us add some menus that duplicate the proposed functionality for the buttons. We will add:

- A **File** menu with a **Open** option, to do the same thing as **Open File**.
- A **Page** menu with **Page Up** and **Page Down** options, to do the same things as the buttons with those names.

The extra code needed in the **`define-interface`** call is this:

```
(:menus
 (file-menu "File"
            ("Open"))
 (page-menu "Page"
            ("Page Up" "Page Down")))
(:menu-bar file-menu page-menu)
```

Menu definitions give a slot name for the menu, followed by the title of the menu, a list of menu item descriptions, and then, optionally, a list of keyword arguments for the menu.

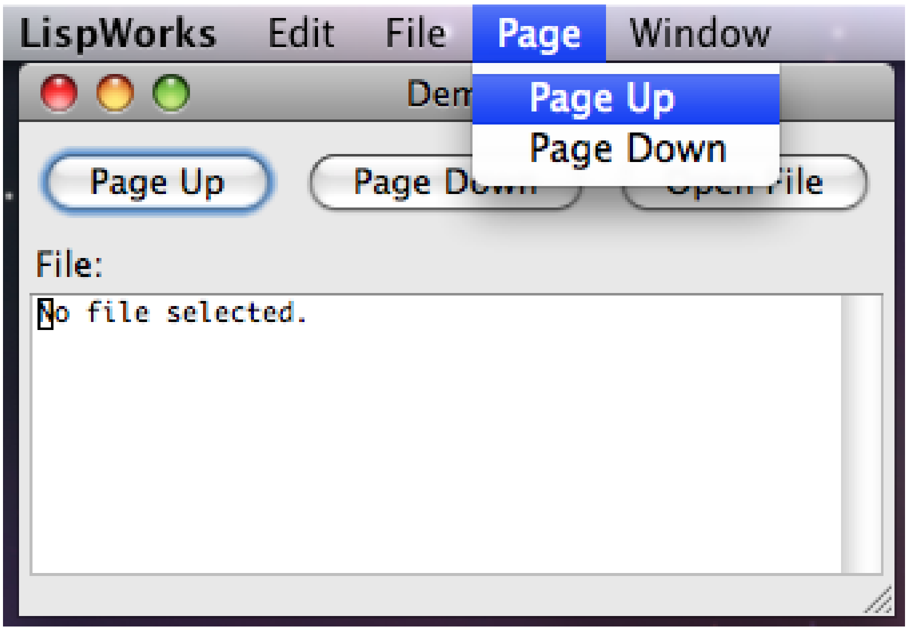
In this instance the menu item descriptions are just strings naming each item, but you may wish to supply initialization arguments for an item — in which case you would enclose the name and those arguments in a list.

The menu bar definition simply names all the menus that will be on the bar, in the order that they will appear. By default, of course, the environment may add menus of its own to an interface — for example the **Window** menu in the LispWorks IDE.

The code for the new interface is:

```
(define-interface demo ()
  ()
  (:panes
   (page-up push-button
            :text "Page Up")
   (page-down push-button
              :text "Page Down")
   (open-file push-button
              :text "Open File")
   (viewer editor-pane
           :title "File:"
           :text "No file selected."
           :visible-min-height '(:character 8)
           :reader viewer-pane))
  (:layouts
    (main-layout column-layout
                 '(row-of-buttons viewer))
    (row-of-buttons row-layout
                    '(page-up page-down open-file)))
  (:menus
   (file-menu "File"
              ("Open"))
   (page-menu "Page"
              ("Page Up" "Page Down")))
  (:menu-bar file-menu page-menu)
  (:default-initargs :title "Demo"))
```

A CAPI interface with menu items



The menus contain the items specified — try it out to be sure.

11.4 Connecting an interface to an application

Having defined an interface in this way, you can connect it up to your program using callbacks, as described in earlier chapters. Here we define some functions to perform the operations we required for the buttons and menus, and then hook them up to the buttons and menus as callbacks.

The functions to perform the page scrolling operations are given below:

```
(defun scroll-up (data interface)
 (call-editor (viewer-pane interface)
```

```
               "Scroll Window Up"))

(defun scroll-down (data interface)
 (call-editor (viewer-pane interface)
              "Scroll Window Down"))
```

The functions use the generic function **call-editor** which calls an editor command (given as a string) on an instance of an **editor-pane**. The editor commands **Scroll Window Up** and **Scroll Window Down** perform the necessary operations for **Page Up** and **Page Down** respectively.

The function to perform the file-opening operation is given below:

```
(defun file-choice (data interface)
  (let ((file (prompt-for-file "Select a File:")))
    (when file
      (setf (titled-object-title (viewer-pane interface))
            (format nil "File: ~S" file))
      (setf (editor-pane-text (viewer-pane interface))
            (file-string file)))))
```

This function prompts for a filename and then displays the file in the editor pane.

The function first produces a file prompter through which a file may be selected. Then, the selected file name is shown in the title of the editor pane (using **titled-object-title**). Finally, the file name is used to get the contents of the file and display them in the editor pane (using **editor-pane-text**).

The correct callback information for the buttons is specified as shown below:

```
(:panes
  (page-up push-button
           :text "Page Up"
           :selection-callback 'scroll-up)
  (page-down push-button
             :text "Page Down"
             :selection-callback 'scroll-down)
  (open-file push-button
             :text "Open File"
             :selection-callback 'file-choice)
  (viewer editor-pane
          :title "File:"
          :text "No file selected."
          :visible-min-height '(:character 8)
          :reader viewer-pane))
```

All the buttons and menu items operate on the editor pane **viewer**. A reader is set up to allow access to it.

The correct callback information for the menus is specified as shown below:

```
(:menus
  (file-menu "File"
             (("Open"))
             :selection-callback 'file-choice)
  (page-menu "Page"
             (("Page Up"
               :selection-callback 'scroll-up)
              ("Page Down"
               :selection-callback 'scroll-down)))
```

In this case, each item in the menu has a different callback. The complete code for the interface is listed below — try it out.

```
(capi:define-interface demo ()
  ()
  (:panes
      (page-up capi:push-button
               :text "Page Up"
               :selection-callback 'scroll-up)
      (page-down capi:push-button
                 :text "Page Down"
                 :selection-callback 'scroll-down)
      (open-file capi:push-button
                 :text "Open File"
                 :selection-callback 'file-choice)
      (viewer capi:editor-pane
              :title "File:"
              :text "No file selected."
              :visible-min-height '(:character 8)
              :reader viewer-pane))
   (:layouts
     (main-layout capi:column-layout
                  '(row-of-buttons viewer))
     (row-of-buttons capi:row-layout
                     '(page-up page-down open-file)))
   (:menus
     (file-menu "File"
                (("Open"))
                :selection-callback 'file-choice)
     (page-menu "Page"
                (("Page Up"
                  :selection-callback 'scroll-up)
                 ("Page Down"
                  :selection-callback 'scroll-down))))
   (:menu-bar file-menu page-menu)
   (:default-initargs :title "Demo"))
```

11.5 Controlling the appearance of the top level window

This section describes ways to control the appearance and behavior of the top level window displaying our CAPI interface.

11.5.1 Window styles

The **`interface`** initarg *window-styles* allows you to control a wide range of visible properties of the top level window including borders, shadows and so on.

window-styles also allows you to specify that the window can be moved by dragging on its background, or cannot be minimized, or acts as a windoid, or is visible only when the application is the current application, and so on.

Many of these properties are specific to the windowing system and are therefore not supported on all platforms. See **`interface`** for the details.

11.5.2 Controlling the interface title

A top level interface has a title, which normally appears at the top. This title is used by the Window Browser tool in the LispWorks IDE and also by system tools that deal with windows. The title is set either by the **`interface`** initarg **`:title`** or the accessor **`interface-title`**.

In addition, you can specify a prefix and/or suffix that is added to the titles of all the interfaces in an application, by using **`set-default-interface-prefix-suffix`**.

The title string is constructed by the generic function **`interface-extend-title`**. The default method constructs it from the title of the interface and the prefix/suffix, if any. For finer control, you can define **interface-extend-title** method(s) for specific interface class(es).

When you change something that may cause the title to change, that is some value that **interface-extend-title** uses, you can use one of **update-interface-title**, **update-screen-interface-titles** or **update-all-interface-titles** to cause the titles to be recomputed.

11.5.3 Indicating a changed document

Some windowing systems support a visible indication that a displayed document has been edited, helping users to see that it needs saving. To implement this in a CAPI interface, set **interface-document-modified-p** at suitable times.

You can extend the definition of the viewer pane in our example like this:

```
(viewer capi:editor-pane
       :title "File:"
       :text "No file selected."
       :visible-min-height '(:character 8)
       :reader viewer-pane
       :change-callback 'check-viewer-modified)
```

and define the *change-callback* as follows:

```
(defun check-viewer-modified (viewer point old-length new-length)
  (declare (ignore point old-length new-length))
  (setf (capi:interface-document-modified-p
         (capi:element-interface viewer))
        (editor:buffer-modified
         (capi:editor-pane-buffer viewer))))
```

Note: Currently **interface-document-modified-p** has an effect only on Cocoa.

11.6 Querying and modifying interface geometry

The functions **screen-monitor-geometries**, **screen-internal-geometries** and **pane-screen-internal-geometry** support the notions of monitor geometry (which includes "system" areas such as the macOS menu bar and the Microsoft Windows task bar) and internal geometry (which excludes the system areas).

Note that code which relies on the position of a window should not assume that a window is located where it has just been programmatically displayed, but should query the current position by **top-level-interface-geometry**. This is because the geometry includes system areas where CAPI windows cannot be displayed.

11.6.1 Support for multiple monitors

CAPI supports multiple monitors by providing functions such as **screen-internal-geometries** to query "screen rectangles" representing the area of each monitor. The function **virtual-screen-geometry** returns a rectangle just enclosing all the screen rectangles.

There is a "primary monitor" which displays any system areas. The origin of the coordinate system (as returned by **top-level-interface-geometry** and **screen-internal-geometry**) is the topmost/leftmost visible pixel of the primary monitor. Thus (0,0) may be in a system area such as the macOS menu bar.

Note also that CAPI does not currently support multiple desktops, which are called workspaces in Linux distros, and called Spaces on macOS.

11.6.2 Saving and restoring top-level geometry

You can specify that that the geometry of a top level interface should be saved when the interface is closed and be used to define the geometry of the interface when it is opened again (potentially in a different invocation of the application). You need to define a method of **`top-level-interface-save-geometry-p`** that returns true for the interface class. You normally also need to specify where to save the geometry, using **`top-level-interface-geometry-key`**.

12 Creating Panes with Your Own Drawing and Input

The CAPI provides a wide range of built-in panes, but it is still fairly common to need to create panes of your own. In order to do this, you need to specify both the input behavior of the pane (how it reacts to keyboard and mouse events) and its output behavior (how it displays itself). The class **`output-pane`** is provided for this purpose.

An **`output-pane`** is a fully functional graphics port. This allows it to use all of the graphics ports functionality to create graphics, and it also has a powerful input model which allows it to receive mouse and keyboard input.

`output-pane` has a subclass **`pinboard-layout`**, to which you can add graphic objects, which makes it easier to organize the interaction when it becomes complex. **`pinboard-layout`** is probably the more useful class.

12.1 Displaying graphics

In order to display your own drawings, you need to provide a function to the **`output-pane`** that is called to redraw sections of the pane when they are exposed. This function is called the *display-callback*, and it is automatically called in the correct process. When the CAPI needs to redisplay a region of an **`output-pane`**, it calls that output pane's *display-callback* function, passing it the pane and the region in question.

For example, to create a pane that has a circle drawn inside it, do the following:

```
(defun draw-a-circle (pane x y width height)
  (gp:draw-circle pane 100 100 50))

(contain
 (make-instance
  'output-pane
  :display-callback 'draw-a-circle)
 :best-width 300
 :best-height 300)
```

Notice that the callback in this example ignores the region that needs redrawing and just redraws everything. This is possible because the CAPI clips the drawing to the region that needs redisplaying, and hence only the needed part of the drawing gets done. For maximum efficiency, it would be better to only draw the minimum area necessary.

The arguments **`:best-width`** and **`:best-height`** specify the initial width and height of the interface. More detail can be found in the manual page for **`interface`**.

Now that we can create output panes with our own display functions, we can create a new class of window by using **`defclass`** as follows.

```
(defclass circle-pane (output-pane)
  ()
  (:default-initargs
   :display-callback 'draw-a-circle))

(contain (make-instance 'circle-pane))
```

Compatibility Note: you must ensure that all drawing occurs inside the display-callback. In previous versions of LispWorks, we documented examples where drawing was done outside the display-callback, but this was always a bad idea because it was not coordinated updates triggered by the window system. On macOS Big Sur and later, drawing outside the display-callback will not work and may cause errors.

12.2 Receiving input from the user

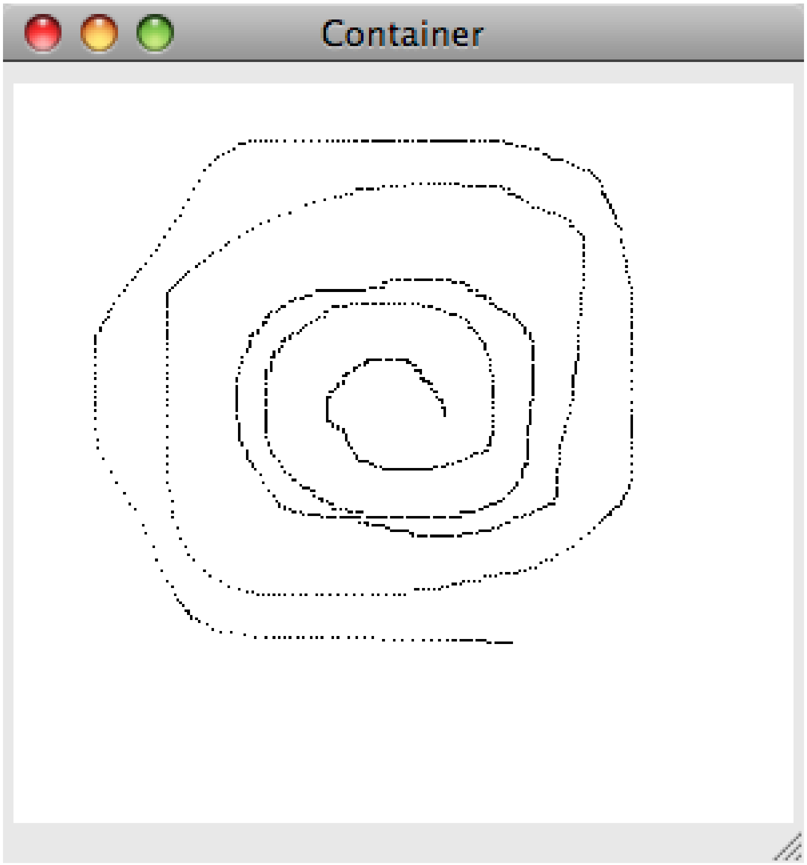
The CAPI supports receiving input from the user through the use of an *input model*, which is a mapping of events to the callbacks that should be run when they occur. The input model is specified by the initarg **`:input-model`**.

When the event callback is called, it gets passed the **`output-pane`** and the *x* and *y* integer coordinates of the mouse pointer at the time of the event. A few events also pass additional information as necessary; for example, keyboard events also pass the key that was pressed.

For example, we can create a very simple drawing pane by adding a callback to draw a point whenever the left button is dragged across the pane. This is done using the function **`draw-point`** as follows:

```
(defun display-a-message (pane x y)
  (display-message-for-pane pane "clicked at ~d/~d" x y))

(contain (make-instance 'output-pane
                        :input-model '(((:button-1 :press)
                                        display-a-message))))
```

An interactive output pane



The input model above seems quite complicated, but it is just a list of event to callback mappings, where each one of these mappings is a list containing an event specification and a callback. An event specification is also a list containing keywords specifying the type of event required.

There is an example input model in:

```
(example-edit-file "capi/graphics/pinboard-test")
```

and more examples are listed in **20.1 Output pane examples**.

For the full *input-model* syntax, see **12.2.1 Detailed description of the input model**.

12.2.1 Detailed description of the input model

The input model provides a means to get callbacks on mouse, keyboard and touch gestures in an **`output-pane`**. An *input-model* is a list of mappings from gesture to callback, where each mapping is a list:

(*gesture callback* **.** *extra-callback-args***)**

gesture specifies the type of gesture, which can be Gesture Spec, character, button, modifier change, key, command, cursor motion or multi-touch. These are described in the following sections. User input is processed as described in **12.2.1.10 Processing user input**.

Note: it is recommended you follow the style guidelines and conventions of the platform you are targeting when mapping gestures to callbacks.

12.2.1.1 Gesture Spec mappings

In a Gesture Spec mapping, *gesture* can be simply the keyword **`:gesture-spec`**, which matches any keyboard input. For specific mappings, *gesture* is a list:

(:gesture-spec *data* **[***modifier***]*)**

in which *data* is a character object or an integer between 0 and **`char-code-limit`** (interpreted as the character object obtained by **`code-char`**), or a keyword naming a function key, and each *modifier* is one of the keywords **`:shift`**, **`:control`** and **`:meta`**. Note that the modifier **`:meta`** is received only when the keys style is **`:emacs`** (see **`interface-keys-style`**).

Also *data* can be a string which is interpreted as a Gesture Spec as if by **`sys:coerce-to-gesture-spec`**. See the *LispWorks® User Guide and Reference Manual* for a description of this and other functions for manipulating Gesture Spec objects.

Note: on Cocoa you cannot receive **`Command`** key gestures via Gesture Spec mapping in *input-model*. To receive **`Command`** key gestures you should add corresponding menu items with accelerators. See **`menu-item`** for information about accelerators.

12.2.1.2 Character mappings

In a character mapping, *gesture* can be simply the keyword **`:character`**, which matches any character input. For specific mappings, *gesture* can be a list containing a single character object *char*, or a list:

(*char***)**

Note: where input would match both a Gesture Spec mapping and a character mapping, the Gesture Spec mapping takes precedence.

Note: in LispWorks 7.0 and later versions the **`cl:character`** type does not support the bits attribute. To represent keyboard input with modifier keys, see **12.2.1.1 Gesture Spec mappings**.

12.2.1.3 Button mappings

In a button mapping, *gesture* should be list:

(*button action* **[***modifiers***]*)**

where *button* is one of **:button-1**, **:button-2** or **:button-3** denoting the mouse buttons. *action* is one of **:press**, **:release**, **:second-press**, **:third-press**, **:nth-press** and **:motion**, and each *modifier* is one of the keywords **:shift**, **:control**, **:meta** and **:hyper**. The **:meta** modifier will be the **Alt** key on most keyboards. On Cocoa, the **:hyper** modifier is interpreted as the **Command** key for button and motion gestures. On Windows, the **:hyper** modifier is currently never generated, so gesture mappings using it will never be invoked.**:third-press** and **:nth-press** are supported only on Cocoa and Motif.

Button mappings with *action* **:nth-press** are matched on the nth button click made in quick succession, but only when there is not a more specific match with **:press**, **:second-press** or **:third-press**. The callback for **:nth-press** receives an extra argument which is the count of clicks.

12.2.1.4 Modifier change mappings

In a modifier change mapping, *gesture* is **:modifier-change**, which generates a callback whenever the state of a modifier (**Control**, **Shift** and **Meta** key, **Command** on Cocoa, and **Caps Lock**) changes.

The *callback* is called with the output pane, *x* and *y*, an integer *mods*, followed by *extra-callback-args* if any. *mods* is calculated as a logior of **sys:gesture-***-bit values. The bits that that may be set in *mods* are:

- **sys:gesture-spec-shift-bit**
- **sys:gesture-spec-control-bit**
- **sys:gesture-spec-meta-bit**
- **sys:gesture-spec-hyper-bit**
- **sys:gesture-spec-caps-lock-bit**

Note that **sys:gesture-spec-hyper-bit** is set when **Command** is pressed.

Note that for Caps Lock, the callback is generated when the state of the Caps Lock changes, not when the **Caps Lock** key is pressed or released.

The pane gets the callback only when it has the focus. If the pane receives the focus and the state of the modifiers is different from what it was the last time the pane had the focus, a callback is generated at that time. That means that tracking the state using the callback is reliable while the pane has the focus, but not while the pane does not have the focus.

For an example, see:

```
(example-edit-file "capi/output-panes/modifier-change")
```

12.2.1.5 Key mappings

Key mappings are intended for detecting low-level keyboard input. In a key mapping, *gesture* should be a list:

(:key [*keyname***]** *action* **[***modifiers***]*)**

where the optional *keyname* is a character naming a key (no modifiers) or one of the valid Gesture Spec keywords documented in the entry for **sys:make-gesture-spec**, *action* is one of **:press** or **:release** and each modifier is one of the keywords **`:shift`**, **`:control`** and **`:meta`**. The callback will receive a **`sys:gesture-spec`**, with its data set to an integer ASCII code or a keyword representing the primary item on the key and its modifiers representing the set of modifiers pressed. The **`:meta`** modifier will be the **`Alt`** key on most keyboards. On Cocoa, the **`:hyper`** modifier is interpreted as the **`Command`** key for **`:key`** input.

12.2.1.6 Motion mappings

In a motion mapping, *gesture* can either be defined in terms of dragging a button (in which case it is defined as a button gesture with *action* **`:motion`**), or it can be defined for motions while no button is down by just specifying the keyword **`:motion`** with no additional arguments.

12.2.1.7 Command mappings

In a command mapping, gesture should be a command which is defined using **`define-command`**, and provides an alias for a gesture. The following commands are predefined:

`:post-menu`
: **`(:button-3 :release)`** on Microsoft Windows.

  `(:button-3 :press)` on Motif.

  `(:button-1 :press :control)` on macOS.

`:control-post-menu`
: **`(:button-3 :press :control)`** on Microsoft Windows, Motif and macOS.

`:keyboard-post-menu`
: **`(:gesture-spec :f10 :shift)`** on Microsoft Windows, Motif and macOS.

12.2.1.8 Touch mappings

On Cocoa and Windows *input-model* can contain mappings for multi-touch gestures from devices that can generate them (trackpad or touchscreen). These include zoom, rotate, pan, swipe (Cocoa only), two finger tap (Windows only), press and tap (Windows only), and beginning and end of sequences of gestures.

In a touch mapping *gesture* should be of the form:

`(:touch` *multi-touch-keyword***`)`**

where *multi-touch-keyword* specifies the type of gesture as listed below. For all multi-touch gestures the callback receives as arguments the *pane*, and the *x* and *y* of the event. There are also an additional one or two arguments for each specific gesture. The extra arguments are always relative to the previous state, so each event can be interpreted on each own. Use *extra-callback-args* if any are added in the end.

multi-touch-keyword should be one of:

`:zoom`
: The callback receives an extra argument which is the zoom factor.

`:rotate`
: The callback receives an extra argument which is the angle to rotate, anti-clockwise in radians.

`:pan`
: The callback receives two extra arguments, the *delta-x* and *delta-y*, which are the amount to scroll in the *x* and *y* directions.

`:swipe`
: The callback receives an extra argument which is one of the keywords **`:left`**, **`:right`**, **`:up`** or **`:down`**.

  `:swipe` is supported only on Cocoa.

:two-finger-tap The callback receives an extra argument which is the distance between the fingers.

:two-finger-tap is supported only on Windows.

:press-and-tap The callback receives two extra arguments, which are the *delta-x* and *delta-y* of the tapping finger from the resting finger.

:press-and-tap is supported only on Windows.

:begin-end The callback receives an extra argument *begin-p* which is a boolean, **t** for beginning of a sequence of events and **nil** for end. The beginning and end of sequences are determined by the underlying device implementation, which tries to identify what the user regards as a single operation.

12.2.1.9 Notes about touch mappings

Because the callbacks receive relative values, you do not need the **:begin-end** events to interpret them. These events are useful when you want to do things which correspond to user operations, for example recording a state for undo or committing a change.

They are also useful if you want to restrict the type of events that are processed inside each operation. For example, your pane may have a flag that the callbacks check and set which is used to allow only one kind of gesture to have an effect in each sequence.

The *x* and *y* coordinates are the coordinates which should be used as the center of operation. On Windows, you can track the *x* and *y* in **:zoom** and **:rotate** events, and do panning while rotating or zooming.

On Cocoa, a sequence of events (starting and ending with **:begin-end** events) can contain either **:zoom** and **:rotate** events or **:pan** events, but not a mixture of **:pan** and **:rotate** or **:zoom**. On Windows all these three types of events can be mixed in principle.

:swipe events (Cocoa only) are three finger brushing. **:swipe** events are always on their own, and are not enclosed in pairs of **:begin-end** callbacks.

On Cocoa, pan should generally act as a scrolling gesture, so normally you should not need to use it.

Windows touch events are described in the MSDN in:

Dev Center - Desktop > Design > Guidelines > Guidelines > Interaction > Touch

http://msdn.microsoft.com/en-us/library/windows/desktop/dn742468(v=vs.85).aspx.

Note that on Windows the **Control+Mousewheel** gesture generates **:zoom** events and **Shift+Mousewheel** generates **:rotate**.

The entries in the *input-model* look like this:

```
((:touch :zoom) my-zoom-callback)

((:touch :pan) my-pan-callback)

((:touch :rotate) my-rotate-callback)

((:touch :begin-end) my-begin-end-callback)

#+macosx
```

```
((:touch :swipe) my-swipe-callback))

#+mswindows
((:touch :two-finger-tap) my-two-finger-tap-callback)

#+mswindows
((:touch :press-and-tap) my-press-and-tap-callback)
```

The corresponding callbacks have these signatures:

my-zoom-callback *pane x y zoom-factor*

my-pan-callback *pane x y delta-x delta-y*

my-rotate-callback *pane x y delta-angle*

my-begin-end-callback *pane x y begin-p*

my-swipe-callback *pane x y direction-keyword*

my-two-finger-tap-callback *pane x y distance*

my-press-and-tap-callback *pane x y distance-x distance-y*

12.2.1.10 Processing user input

When user input matches a gesture *gesture*, the *callback* is called with the gesture callback arguments followed by any user-supplied *extra-callback-args*.

The gesture callback arguments contain three standard arguments, and for some gestures there is a fourth argument. The standard three arguments are:

output-pane x y

where (*x*, *y*) is the cursor position.

The following gestures have a fourth argument:

:gesture-spec or **:key**

A **sys:gesture-spec** representing the user input.

:character or **character**

A character representing the user input.

:modifier-change

An integer specifying the modifiers as a **logior** of the constants **sys:gesture-spec-shift-bit** etc.

Button with **:nth-press**

An integer which is the number of clicks.

Note: mouse gestures with **`:press`**, **`:second-press`**, **`:third-press`** and **`:nth-press`** actions can each be expected to be followed by a **`:release`** action.

Note: In some circumstances **`:motion`** events can be received even when the **`output-pane`** does not have the input focus. See window style **`:motion-events-without-focus`** under **`interface`** for details.

input-model can be set before the pane is displayed, but changes after that are ignored.

In particular, **`cl:initialize-instance`** is the natural place for subclasses to modify the existing *input-model*, using the **`output-pane`** accessor **`output-pane-input-model`**. Note that since the mappings are processed in order, prepending to an existing *input-model* overrides it when there are clashes, while appending affects only gestures for which the original *input-model* did not have a match.

12.2.2 Commands - aliases

It is possible to define aliases for gestures (called "commands"), which is mapping between a gesture and a command (a unique Lisp object, typically a keyword). The command then can be used as the *gesture* in an *input-model*. That allows changing the actual user gesture to invoke the callbacks that are associated with the command in input models of many panes, without having to change the actual input model specifications.

A command is defined using **`define-command`**, which defines the mapping, and can also specify on which library it is applicable and a translator to change the arguments that are passed to the callback.

Commands that are defined by **`define-command`** can be programmatically invoked (as if the user entered the gesture) by **`invoke-command`** or **`invoke-untranslated-command`**.

12.2.3 Native input method

The input that CAPI sees may be pre-processed by a native input method. Native input methods are part of the underlying GUI system which allow the user to enter characters that do not appear on the keyboard. On GTK+ you can control whether the native input method is used by the **`output-pane`** initarg **`:use-native-input-method`**, and you can specify the default by **`set-default-use-native-input-method`**.

12.2.4 Composition of characters

Composition of characters is done by the underlying window system, which combines several keystrokes to one character (or more rarely, to several characters) , and is used to input characters that are not available on the keyboard. **`output-pane`** has a callback, **`:composition-callback`**, which is called when composition starts and ends, and also if the pane is supposed to display the input, it is called to tell it what to display.

Inside the callback call for starting composition, the function **`set-composition-placement`** where relative to the composition should, which tells the system where to put any window that it popups to interact the user. For example, **`editor-pane`** uses this to set the placement at the position of the cursor.

12.3 Creating graphical objects

A common feature needed by an application is to have a number of objects displayed in a window and to make events affect the object underneath the cursor. The CAPI provides the ability to create graphical objects, to place them into a window at a specified size and position, and to display them as necessary. Also a function is provided to determine which object is under any given point so that events can be dispatched correctly.

These graphical objects are called *pinboard objects*, as they can only be displayed if they are contained within a **`pinboard-layout`**.

Like simple panes, you display a **pinboard-object** by putting it in the *description* of a **layout**, but in the case of a **pinboard-object** the layout must be either a **pinboard-layout** or a layout that is a descendant of a **pinboard-layout** (to any depth). Adding or removing **pinboard-object**s can be done using the standard mechanism of the **:description** initarg and **(setf layout-description)**, but normally it should be done by **manipulate-pinboard**. This is much more efficient and causes much less flickering, which is important when there are many objects.

CAPI provides built-in pinboard object classes for several simple cases including **item-pinboard-object** for displaying text, **line-pinboard-object**, **rectangle**, **ellipse** and **arrow-pinboard-object** for simple shapes, and **image-pinboard-object** for displaying an image. To display more complex drawing, you can use **drawn-pinboard-object**, which takes a *display-callback* which actually does the drawing. For greater control, you can subclass **pinboard-object**, and define the method **draw-pinboard-object** to do the drawing, and if needed also **draw-pinboard-object-highlighted**. You can also subclass any of the specialized **pinboard-object** subclasses if it is useful.

pinboard-objects have geometry like **simple-pane**, that is *x*, *y*, *width* and *height*. These can be specified initially by the initargs **:x** and **:y** and geometry hints (see **6.4 Specifying geometry hints**), and can be read and set later by **static-layout-child-position** and **static-layout-child-size**. They can also be read by using the binding inside **with-geometry**, but setting should be done only by **(setf static-layout-child-position)** and **(setf static-layout-child-size)**.

For **line-pinboard-object** and its subclasses, you would normally specify the start and end points, rather than the rectangle that encloses it (which would require computations taking into account the line width and the position of any label). This is done when making the object using the initargs **:start-x**, **:start-y**, **:end-x** and **:end-y**, and later by the function **move-line**. The function **line-pinboard-object-coordinates** can be used to find the start and end points of an object.

The graphics args that are used to draw the objects in built-in subclasses of **pinboard-object** can be specified by supplying the initarg **:graphics-args**, and modified dynamically by **(setf pinboard-object-graphics-args)** and **(setf pinboard-object-graphics-arg)**. For example, the following code displays a line and after 2 seconds changes its color:

```
(progn
  (setq po
        (capi:contain
         (make-instance 'capi:line-pinboard-object
                        :start-x 50 :end-x 250
                        :start-y 50 :end-y 50
                        :graphics-args
                        '(:thickness 10 :foreground :red))))
  (sleep 2)
  (capi:apply-in-pane-process
   po
   #'(lambda ()
       (setf (capi:pinboard-object-graphics-arg po :foreground)
             :blue))))
```

For pinboard object classes which you define, the drawing functions that you call need to do the drawing using the Graphics Ports drawing functions (see **13.4 Drawing functions**). They take their coordinates with respect to the **pinboard-layout** (not the object), so you need to use the *x* and *y* to compute the arguments for the drawing functions. This is how the specialized classes mentioned above know where to draw. You need to keep the drawing inside the geometry (that is inside the rectangle defined by *x*, *y*, *width* and *height*), because the **pinboard-layout** decides which objects need redrawing using these values.

pinboard-objects can be highlighted. You need to use the functions **highlight-pinboard-object** and **unhighlight-pinboard-object** to switch the highlight state of objects. The function **pinboard-object-highlighted-p** can be used to check whether an object is in the highlighted state. By default, CAPI calls **draw-pinboard-object-highlighted** to add the highlight after drawing the object. In many cases, it is better to do the highlight in the drawing function (either the method of **draw-pinboard-object** or the *display-callback* for

drawn-pinboard-object) rather than separately. Use the initarg **:no-highlight** with value **t** when making the **pinboard-object**, and **pinboard-object-highlighted-p** inside the drawing function to check whether it needs to highlight. These examples both use this technique:

```
(example-edit-file "capi/graphics/circled-graph-nodes")
```

```
(example-edit-file "capi/graphics/tracking-pinboard-layout")
```

It is possible to set an element such that its geometry changes automatically when the **pinboard-layout** is resized, by using either the initarg **:automatic-resize** or calling **set-object-automatic-resize**. See:

```
(example-edit-file "capi/layouts/automatic-resize")
```

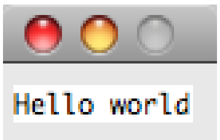
Note: **pinboard-object**s are implemented as graphics on a native window. Compare this with **simple-pane** and its subclasses, where each instance is itself a native window. A consequence of this is that **simple-pane**s do not work well within a **pinboard-layout**, since they always appear above the **pinboard-object**s. For example, to put labels on a pinboard, use **item-pinboard-object** rather than **display-pane** or **title-pane**.

Note: The **pinboard-layout** displays the pinboard objects via its own *display-callback* function **pinboard-layout-display**. If you want do other drawing too, see the entry for **pinboard-layout-display**. It is also possible to draw the pinboard objects of a **pinboard-layout** to another graphics port (for example, a pixmap) using **draw-pinboard-layout-objects**.

Here is an example of the built-in pinboard object class **item-pinboard-object** which displays its text like a **title-pane**. Note that the function **contain** always creates a **pinboard-layout** as part of the wrapper for the object to be contained, and so it is possible to test the display of **pinboard-object**s in just the same way as you can test other classes of CAPI object.

```
(contain
  ;; CONTAIN makes a pinboard-layout if needed, so we don't
  ;; need one explicitly in this example.
  ;; You will need an explicit pinboard-layout if you define
  ;; your own interface class.
  (make-instance
   'item-pinboard-object
   :text "Hello world"))
```

A pinboard object



Here is another example illustrating **item-pinboard-object**:

```
(example-edit-file "capi/graphics/pinboard-object-text-pane")
```

12.3.1 Buffered drawing

Where the display of an **output-pane** is complex you may see flickering on screen on some platforms. Typically this occurs in a **pinboard-layout** with many pinboard objects, or some other characteristic that makes the display complex.

The flickering can be avoided by passing the *draw-with-buffer* initarg which causes the drawing to go to an off-screen pixmap buffer. The screen is then updated from the buffer.

Note: GTK+ and Cocoa always buffer, so the *draw-with-buffer* initarg is ignored on these platforms.

12.3.2 Finding pinboard objects from coordinates

To find the top **pinboard-object** at a supplied position (*x*, *y*), which is typically needed when processing user input, use **pinboard-object-at-position**. To decide whether a pinboard object is at a position, **pinboard-object-at-position** uses the generic function **over-pinboard-object-p**. **over-pinboard-object-p** has a default method that return true when the position is in the rectangle of the object, and a method for line object (subclasses of **line-pinboard-object**) that return true if the position is close to the line. You add methods to **over-pinboard-object-p** for your own classes. For example, if your pinboard object displays a thunder picture, you may want an **over-pinboard-object-p** method that computes whether the position is inside the thunder drawing.

There is also the generic function **pinboard-object-overlap-p**, with a default method that determines whether the rectangle of the object overlaps the rectangle specified by the other arguments.

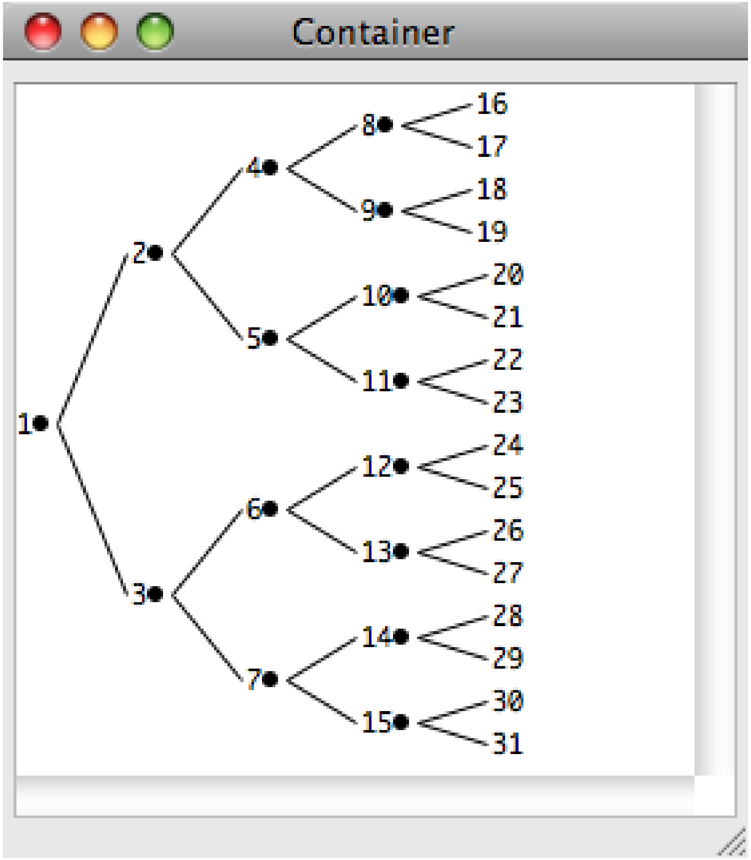
12.3.3 The implementation of graph panes

One of the major uses the CAPI itself makes of pinboard objects is to implement graph panes. The **graph-pane** itself is a **pinboard-layout** and it is built using **pinboard-object**s for the nodes and edges. This is because each node (and sometimes each edge) of the graph needs to react individually to the user. For instance, when an event is received by the **graph-pane**, it is told which pinboard object was under the pointer at the time, and it can then use this information to change the selection.

Create the following **graph-pane** and notice that every node in the graph is made from an **item-pinboard-object** as described in the previous section and that each edge is made from a **line-pinboard-object**.

```
(defun node-children (node)
  (when (< node 16)
    (list (* node 2)
          (1+ (* node 2)))))


(contain
 (make-instance
  'graph-pane
  :roots '(1)
  :children-function 'node-children)
 :best-width 300 :best-height 400)
```

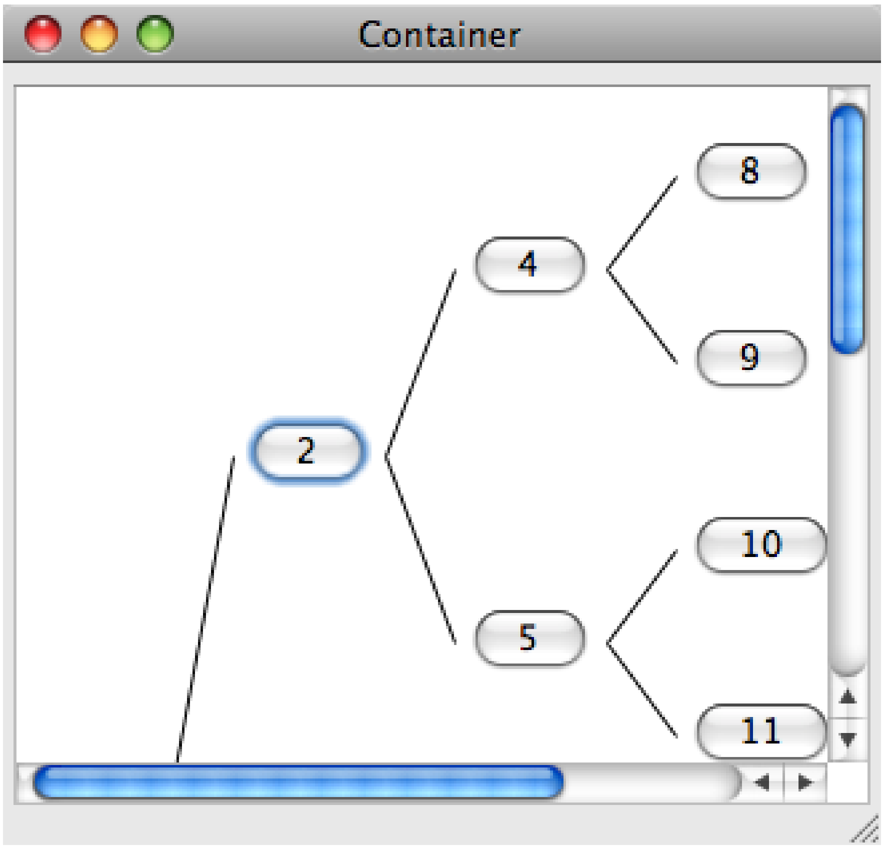
A graph pane with pinboard object nodes



As mentioned before, **pinboard-layout**s can just as easily display ordinary panes inside themselves, and so the **graph-pane** provides the ability to specify the class used to represent the nodes. As an example, here is a **graph-pane** with the nodes made from **push-button**s.

```
(contain
 (make-instance
  'graph-pane
  :roots '(1)
  :children-function 'node-children
  :node-pinboard-class 'push-button)
 :best-width 300 :best-height 400)
```

A graph pane with push-button nodes



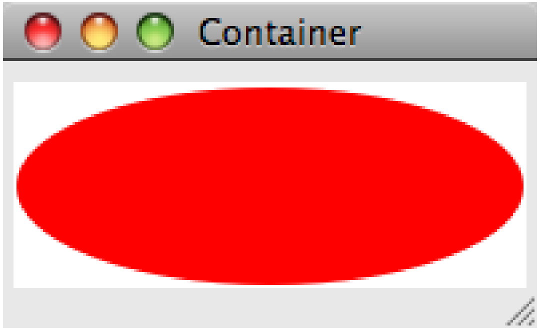
12.3.4 An example pinboard object

To create your own pinboard objects, the class **drawn-pinboard-object** is provided, which is a **pinboard-object** that accepts a *display-callback* to display itself. The following example creates a new subclass of **drawn-pinboard-object** that displays an ellipse.

```
(defun draw-ellipse-pane (gp pane
                             x y
                             width height)
  (with-geometry pane
    (let ((x-radius
           (1- (floor %width% 2)))
          (y-radius
           (1- (floor %height% 2))))
      (gp:draw-ellipse
       gp
       (1+ (+ %x% x-radius))
       (1+ (+ %y% y-radius))
       x-radius y-radius
       :filled t
       :foreground
       (if (> x-radius y-radius)
           :red
         :yellow)))))

(defclass ellipse-pane
     (drawn-pinboard-object)
  ()
  (:default-initargs
   :display-callback 'draw-ellipse-pane
   :visible-min-width 50
```

```
      :visible-min-height 50))

(contain
 (make-instance 'ellipse-pane)
 :best-width 200
 :best-height 100)
```

An ellipse-pane class

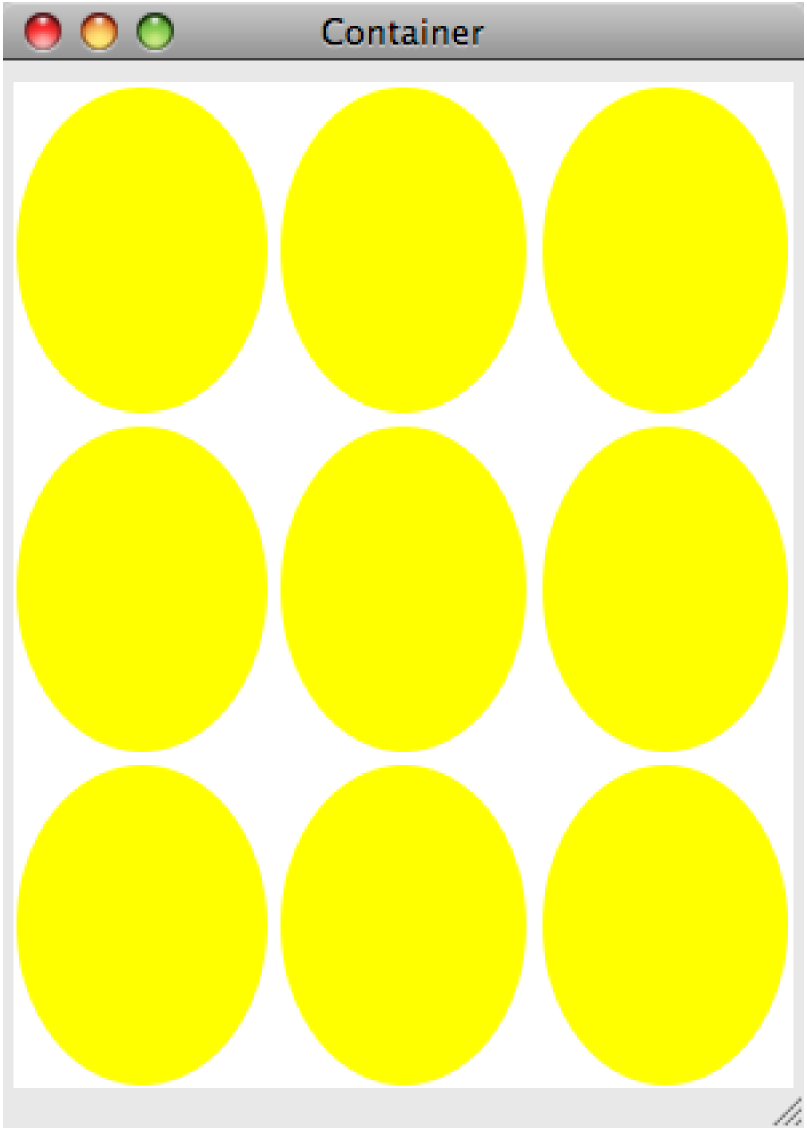


The **with-geometry** macro is used to set the size and position, or geometry, of the ellipse drawn by the **draw-ellipse-pane** function. The fill color depends on the radii of the ellipse - try resizing the window to see this. For more details of see the manual page for **drawn-pinboard-object**.

Now that you have a new ellipse-pane class, you can create instances of them and place them inside layouts. For instance, the example below creates nine ellipse panes and places them in a three by three grid.

```
(contain
 (make-instance
  'grid-layout
  :description
  (loop for i below 9
        collect
        (make-instance 'ellipse-pane))
  :columns 3)
 :best-width 300
 :best-height 400)
```

Nine ellipse-pane instances in a layout



12.3.5 Simple pinboard layout

simple-pinboard-layout is a subclass of **pinboard-layout** with only one child (a pane or a **pinboard-object**). It adopts the size constraints of its child. **simple-pinboard-layout** is useful when you want to arrange **pinboard-object**s using a **layout** pane (or a hierarchy of **layout**s). **pinboard-object**s need a **pinboard-layout** somewhere in the parent hierarchy, but using **pinboard-layout** would mean that the constraints computed by **layout** (top **layout** if it is a hierarchy) would not be automatically propagated to the next level. **simple-pinboard-layout** solves this problem. An example is the **graph-pane**, which is actually a subclass of **simple-pinboard-layout**, and as a child has a **layout** (of internal type) with a special algorithm that lays out the graph and displays it using **pinboard-object**s.

12.3.6 Tracking pinboard layout

tracking-pinboard-layout is a subclass of **pinboard-layout** which tracks the motion of the mouse cursor, by highlighting the object underneath it (if any). Otherwise it behaves the same as **pinboard-layout**. It saves you from implementing the tracking when it is is desired.

```
(example-edit-file "capi/graphics/tracking-pinboard-layout")
```

12.4 output-pane scrolling

An **output-pane** or an instance of any of its subclasses can be made to scroll by passing the **:vertical-scroll** and/or **:horizontal-scroll** initargs which are inherited from **simple-pane**.

12.4.1 Ordinary scrolling

By default, the scrolling is what is called *ordinary scrolling*. In this case you just need to specify that you want scrolling by **:vertical-scroll** and/or **:horizontal-scroll**, and maybe also specify the internal scroll dimension(s) (see below).

In ordinary scrolling, all the interactions are done as if the pane has an "internal canvas" with dimensions (the "internal dimensions") which are different from the visible dimensions on the screen, and typically larger. The coordinates of input gestures and drawing in the pane are all with respect to this internal canvas. Only part of the canvas is displayed at any one time, depending on the position of the scroll slugs. The effect of scrolling is to change what part of the pane is visible, which causes a *display-callback* to draw any newly visible areas. However, the call to the *display-callback* is an ordinary call like any call (for example, like a call as result of part of the window being exposed), and the *display-callback* does not need to know anything about scrolling.

If you need to know when scrolling happened, rather than just display what is needed to display, you can use the **:scroll-callback** initarg to specify a callback that is called before the *display-callback*. However, this is not required for ordinary scrolling to work.

The internal dimensions of the pane can be specified by the initargs **:scroll-height** and **:scroll-width**, and can also be set dynamically set by **set-vertical-scroll-parameters** and **set-horizontal-scroll-parameters**. Some subclasses can compute their internal dimensions, for example **graph-pane** computes its internal dimensions to show all the graph, and **static-layout** and its subclass **pinboard-layout** by default compute the internal dimensions to fit their children (unless *fit-size-to-children* is **nil**).

For example, create an **output-pane** with vertical scroll and internal height of 600 pixels, minimum visible height of 300 pixels, and a *display-callback* that prints the *y* coordinate and the *height* and displays a green square at (0,100) of size 10x10 and a blue square at (0,400) of size 10x10:

```
(defun my-display-callback (pane x y width height)
  (declare (ignore x width))
  (format t " y = ~d,  height =  ~d~%" y height)
  (gp:draw-rectangle pane 0 100 10 10
                     :foreground :green :filled t)
  (gp:draw-rectangle pane 0 400 10 10
                     :foreground :blue :filled t))

(setq output-pane
      (make-instance 'capi:output-pane
                      :vertical-scroll t
                      :scroll-height 600
                      :visible-min-height 300
                      :display-callback 'my-display-callback))
```

Then display it:

```
(capi:contain output-pane)
```

When it appears on the screen its *height* is 300 pixels, the scrollbar is half the height. You receive a display callback with *y* being 0 and *height* 300. You see the green square 100 pixels down from the top. The blue square is invisible, because it is drawn at *y* = 400, which is not inside the visible area.

Now if you scroll to the bottom, you will receive a callback with *y* = 300 and *height* still 300 (possibly after several callbacks with intermediate *y* values). Now you see the blue square 100 pixels from the top, and the green square is invisible.

Note that the display callback knows nothing about the scrolling. It just draws. A real display callback may be made faster by avoiding the drawings which are not going to be visible, for example:

```
(defun my-display-callback-1 (pane x y width height)
  (declare (ignore x width))
  (format t " y = ~d,  height =  ~d~%" y height)
  (unless (or (> y 110) (< (+ Y height) 100) (> x 10))
    (gp:draw-rectangle pane 0 100 10 10
                       :foreground :green :filled t))
  (unless (or (> y 410) (< (+ Y height) 400) (> x 10))
     (gp:draw-rectangle pane 0 400 10 10
                        :foreground :blue :filled t)))
```

but this is just optimization. It does not affect what is shown on the screen.

12.4.2 Internal scrolling

The other type of scrolling is called *internal scrolling* (sometimes "pane scrolling"), and it is set up by passing the **output-pane** initarg **:coordinate-origin** with either **:fixed** or **:fixed-graphics**. In general, internal scrolling is more complex to use, but allows more flexible scrolling.

When using internal scrolling with *coordinate-origin* **:fixed**, drawing coordinates are relative to the visible area, and the coordinates arguments to callbacks are also relative to the visible area. Thus drawing a rectangle at 0,100 as *my-display-callback* above does will always show it at 0,100 on the screen, ignoring any scrolling.

For example, evaluate the following (which requires the definition of my-display-callback):

```
(capi:contain (make-instance
               'capi:output-pane
               :vertical-scroll t
               :scroll-height 600
               :visible-min-height 300
               :display-callback 'my-display-callback
               :coordinate-origin :fixed  ; <<
               )
              :title "With :coordinate-origin :fixed")
```

Scroll it and you will see that it is "fixed": the green rectangle does not move, and the *y* coordinate that is passed to *my-display-callback* is always 0.

When using internal scrolling with *coordinate-origin* **:fixed-graphics**, the drawing coordinate are relative to the visible pane, but CAPI coordinates (that is the arguments to callbacks such as *display-callback*, *scroll-callback* and *input-model* and in calls to **display-popup-menu**) are offset by the scroll position of the pane like in ordinary scrolling. The scroll position can be obtained by calling **get-horizontal-scroll-parameters** and **get-vertical-scroll-parameters** with **:slug-position**, or from **%scroll-x%** and **%scroll-y%** inside **with-geometry**.

For example, evaluate this:

```
(capi:contain (make-instance
               'capi:output-pane
               :vertical-scroll t
               :scroll-height 600
               :visible-min-height 300
               :display-callback 'my-display-callback
               :coordinate-origin :fixed-graphics ;<<
               )
              :title "With :coordinate-origin :fixed-graphics")
```

Scroll it and you will see that the graphics are "fixed" (the green rectangle does not move) but the coordinates "scroll" (the *y* coordinate increases as you scroll). In practice, this means that to get the effect of scrolling, the *display-callback* needs to subtract the scroll position before drawing, or use Graphics Ports transformations, for example:

```
(gp:with-graphics-translation (pane (- scroll-x) (- scroll-y))
   (do-all-the-drawing))
```

If you do not supply *scroll-callback* (inherited from **simple-pane**) in a pane that does internal scrolling, then LispWorks calls **update-internal-scroll-parameters** in response to scrolling gestures to update the internal parameters (that updates the scroll bars themselves if needed), and then calls **invalidate-rectangle**, which will cause the *display-callback* to be called for the whole visible area of the pane. In many cases, that is what you need, but not always.

In some cases, redisplaying the whole of the pane every time it scrolls may not be required or may be too slow, and in other cases you will want to do other things. In these situations, performs the scrolling yourself by supplying a *scroll-callback*. When you supply a *scroll-callback*, your function is responsible for doing anything that needs to be done to make "scrolling" happen (which is not necessarily proper scrolling).

In general, your *scroll-callback* will have to call **update-internal-scroll-parameters** (and maybe **set-vertical-scroll-parameters** or **set-horizontal-scroll-parameters**) to update the scroll parameters, and **get-vertical-scroll-parameters** and **get-horizontal-scroll-parameters** to get the scroll values. Some of these values may be initialized by the **:scroll-...** initargs of **output-pane**. *scroll-callback* may also need to do other computations.

Once the *scroll-callback* has adjusted the internal scrolling state of the application, it needs to ensure that the pane is redisplayed, by calling **invalidate-rectangle** on the area (or on each of multiple areas) that need(s) to be redisplayed. This will then cause the *display-callback* of the **output-pane** to be called on those areas. The *display-callback* needs to know how to draw the pane taking into account the internal scrolling state. It can do that by calling **get-vertical-scroll-parameters** and **get-horizontal-scroll-parameters** (or using the **%scroll-...%** variables inside **with-geometry**), or by using some internal scrolling state that *scroll-callback* has set up.

For examples of internal scrolling that do a little unconventional scrolling see:

```
(example-edit-file "capi/output-panes/coordinate-origin-fixed")
```

For an example of internal scrolling that does something different altogether (rotating) see:

```
(example-edit-file "capi/output-panes/fixed-origin-scrolling")
```

Ordinary scrolling is not only easier to use, but is also normally more efficient, because the underlying window system handles scrolling. In particular, areas that move on the screen are just copied, without a need to redraw what is displayed.

Internal scrolling is useful in situations where what is displayed changes according to the scroll position, other than just scrolling. With ordinary scrolling, the underlying window system calls the *display-callback* when scrolling happens, but only for areas that become visible by the scroll operation. Other areas are normally just copied to their new locations, so the program cannot change them. For example, the display callback below tries to keep a string with a yellow background at a fixed position 100 pixels down from the top left of the pane:

```
(defun a-display-callback (pane x y width height)
  (let* ((scroll-y
          (capi:get-vertical-scroll-parameters pane
              :slug-position)))
    (gp:draw-string pane "A string"  0 (+ scroll-y 100)
                    :background :yellow :block t)))

(capi:contain
 (make-instance 'capi:output-pane
                :vertical-scroll t
                :scroll-height 900
```

```
                    :visible-max-height 600
                    :display-callback 'a-display-callback))
```

However, once you display it and try to scroll, it should be obvious that it does not work because the window system moves the string an the display callback is not called for the area 100 pixels down from the top left of the pane.

One way of working around this kind of issue is add a *scroll-callback* that fixes the display, for example by calling **invalidate-rectangle**, but that can become quite complex. The other way is to use internal scrolling.

Apart from the *display-callback*, the *scroll-callback* and any code that needs to know about scrolling because of the logic of the application, the rest of your code should not need to worry about scrolling. Thus it does not actually add must complexity to your code.

Another situation when you may prefer internal scrolling is when your code precomputes what to display based on the scroll position, and the *display-callback* does minimal computation that is not substantially more expensive than the copying the system would do. That will mean that the *display-callback* does not need to know about scrolling, but all your callbacks will either have to add the scroll position to the their arguments, or work with respect to the precomputed information rather than the whole pane. The latter is what **editor-pane** does.

12.5 Transient display on output-pane and subclasses

It is quite often that you want to transiently add some drawing on top of the permanent drawing of an **output-pane**. Most typically, you want to allow the user to select an area by dragging the mouse while pressing a button, and you want to include some transient graphics to indicate what they are going to select. This could simply be a rectangle, but you may want something more complex.

Ideally, the *display-callback* of the pane would be fast enough to handle this, in which case you simply need to make the *display-callback* draw the transient graphics. For example, in the case of a **pinboard-layout**, it can be done by adding a transient **pinboard-object** above the other objects. This is demonstrated by the "outliner" example:

```
(example-edit-file "capi/graphics/pinboard-test")
```

Note that in this case the outliner's drawing is simple, but it could draw much more complex graphics if required.

However, that solution does not work well if the *display-callback* is not fast enough for these situations. The Cached Display functionality is intended to be used in this case. There are two ways to use the Cached Display interface:

1. Use **output-pane-cache-display** to cache the display, and then **output-pane-draw-from-cached-display** to draw from the cache. In this case you have to ensure that the *display-callback* knows when to use **output-pane-draw-from-cached-display**, either by replacing the *display-callback* for the duration of the Cached Display operation or by keeping a flag that the *display-callback* checks, for example:

```
(if (drawing-by-cached-display-p pane)
    (progn
      (output-pane-draw-from-cached-display pane x y width height)
      (do-some-transient-drawing pane))
  (real-display-callback pane x y width height))
```

2. Use **start-drawing-with-cached-display**, which replaces the *display-callback*, and then use **update-drawing-with-cached-display** or **update-drawing-with-cached-display-from-points** to update the display. This technique is illustrated in:

```
(example-edit-file "capi/output-panes/cached-display")
```

In both cases you finish using the cached display by calling **output-pane-free-cached-display**. The function **output-pane-cached-display-user-info** can be used to hold temporary data during the operation.

13 Drawing - Graphics Ports

13.1 Introduction

Graphics Ports allow you to write source-compatible applications which draw text, lines, shapes and images, for different host window systems. Graphics Ports are the destinations for the drawing primitives. They are implemented with a generic host-independent part and a small host-specific part.

All Graphics Ports symbols are exported from the **`graphics-ports`** package, nicknamed **`gp`**.

Graphics Ports implement a set of drawing functions and a mechanism for specifying the graphics state to be used in each drawing function call. There are four categories of graphics ports:

On-screen ports	These correspond to visible windows. They are instances of **`output-pane`** or a subclass, and are integral part of the CAPI panes system. The functionality of **`output-pane`** (other than drawing) is discussed in **12 Creating Panes with Your Own Drawing and Input**. All drawing to an **`output-pane`** must be done inside its *display-callback*.
Pixmap ports	These are solely for off-screen drawing. Once the drawing is completed they can be copied to another port (typically an on-screen port, with **`copy-area`**), or converted to an image. For the details see **13.1.2 Pixmaps and Metafiles**.
Printer ports	These are used for drawing to a printer. Printing is described in **16 Printing from the CAPI—the Hardcopy API**.
Metafile ports	These are used for recording drawing operations so that the drawing can be realized later or exported to a file that can read by other applications. For the details see **13.1.2 Pixmaps and Metafiles**.

13.1.1 Creating instances

Graphics ports instances are created or temporarily redirected by any of these interfaces:

On-screen ports	**`make-instance`** with **`output-pane`** or any subclass (including **`editor-pane`**, **`pinboard-layout`** and **`graph-pane`**).
Pixmap ports	**`create-pixmap-port`** and **`with-pixmap-graphics-port`**.
Metafile ports	**`with-internal-metafile`** and **`with-external-metafile`**.
Printer ports	**`with-print-job`** and **`simple-print-port`**.

For the details, see the manual pages for the various CAPI and GRAPHICS-PORTS classes listed above.

13.1.2 Pixmaps and Metafiles

Pixmaps are graphics ports for doing off-screen drawing. You create a pixmap with **`with-pixmap-graphics-port`** or **`create-pixmap-port`**, and draw on it using the drawing functions. You draw the contents of the pixmap on another port (any kind of port) by copying it (using **`copy-area`**), or create an image from it using **`make-image-from-port`**. The drawing into and the using of a pixmap can be interleaved (but not in parallel), and each time you use the pixmap you get the result of all the drawing operations on it until this point. If the pixmap is created by **`with-pixmap-graphics-port`** it is destroyed on exiting the scope of **`with-pixmap-graphics-port`**, otherwise you will need to destroy the pixmap when you finish with it (using **`destroy-pixmap-port`**).

Pixmaps are used for efficiency. In general **`copy-area`** would be much faster than doing the drawing operations again for any significant number of drawing operations. It is especially useful for drawing inside the *display-callback* of an **`output-pane`**, which is called whenever part of the output pane needs redrawing, and needs to be fast to look good.

Pixmaps are also useful way of creating your own images for exporting with **`externalize-and-write-image`**.

Examples of using pixmaps:

```
(example-edit-file "capi/graphics/compositing-mode-simple")

(example-edit-file "capi/graphics/compositing-mode")

(example-edit-file "capi/graphics/image-scaling")

(example-edit-file "capi/graphics/images-with-alpha")

(example-edit-file "capi/graphics/pixmap-port")

(example-edit-file "capi/graphics/plot-offline")
```

Metafiles are graphics ports that record drawing operations to them. They are used for two purposes:

- Grouping drawing operations together.

  The operations can then be drawn by one call, and on Cocoa and Windows can also be put in on the clipboard so that another process can access it.

- Exporting the drawing to a file.

  The file is in a format that other applications can also use.

You can group operations by drawing to a metafile inside **`with-internal-metafile`** which returns a metafile object, and later drawing the metafile by using **`draw-metafile`**. You can also convert it directly to an image by **`draw-metafile-to-image`**. Once you have finished with it you need to free the metafile by **`free-metafile`**.

It is possible to perform the same task by drawing the operations to a pixmap and then drawing the pixmap, as described above. However, a metafile gives much better results when it is transformed, because it does the drawing with the transformation, while with a pixmap the transformation transforms the pixels. Metafiles also give better results when the drawing is not completely opaque.

The result of **`with-internal-metafile`** can also be put on the clipboard for other processes, by using **`set-clipboard`** with a **`:plist (list :metafile`** *metafile***`)`**. LispWorks can also read a metafile from the clipboard by passing **`:metafile`** as the *format* to **`clipboard`**.

You can export the drawing to a file by drawing to a metafile inside using **`with-external-metafile`**, which creates the file when it exits.

On Microsoft Windows it creates a Windows enhanced metafile (there are several possible formats). On Cocoa and GTK+ it creates a PDF file.

Compared to exporting images (using **`with-pixmap-graphics-port`**, **`make-image-from-port`**, and **`externalize-and-write-image`**), the exported metafiles (PDF or Windows metafile) behave much better in transformation and combination with other drawings. They are also simpler to use.

LispWorks itself can read the file that was created by **`with-external-metafile`** using the functions that read images (**`load-image`**, **`read-external-image`**).

Metafile functionality is not available on version of GTK+ before 2.8, and on Motif. The function **`can-use-metafile-p`** can be used to check whether the GUI system associated with a screen supports metafile functionality.

Examples of metafiles:

```
(example-edit-file "capi/graphics/metafile")
```

```
(example-edit-file "capi/graphics/metafile-rotation")
```

13.2 Features

The main features of graphics ports are:

1. Each port has a "graphics state" which holds all the information about drawing parameters such as color, line thickness, fill pattern, line-end-style and so on. A graphics state object can also be created independently of any particular graphics port.
2. The graphics state contents can either be enumerated in each drawing function call, bound to values for the entirety of a set of calls, or permanently changed.
3. The graphics state includes a **`transform`** which implements generalized coordinate transformations on the port's coordinates.
4. Off-screen ports can compute the horizontal and vertical bounds of the results of a set of drawing function calls, thus facilitating image or pixmap generation.

13.2.1 The drawing mode and anti-aliasing

Graphics ports has two drawing modes:

`:compatible` Compatible with LispWorks 6.0 and earlier versions.

`:quality` Introduced in LispWorks 6.1, allowing high quality drawing.

The main visible effect is that with *drawing-mode* **`:quality`**, all drawings are transformed properly.

With *drawing-mode* **`:compatible`**, strings and images are not scaled or rotated at all, and ellipses are not rotated correctly. Other shapes are transformed "at the front", that is they are drawn as if the drawing function was called with transformed coordinates. The target of **`copy-pixels`** is also transformed "at the front", that is the rectangle can be translated, but not scaled or rotated.

With *drawing-mode* **`:quality`**, all drawings are fully transformed correctly. Shapes are transformed "at the back", that is they are drawn and then the result of the drawing is transformed. Note that **`clear-rectangle`** and **`pixblt`** are not drawing functions in this sense, and do not take transforms into account.

Another difference is that *drawing-mode* **`:quality`** supports anti-aliasing on Windows, and on GTK+ it adds control over anti-aliasing. See *shape-mode* and *text-mode* on the page for **`graphics-state`**.

With *drawing-mode* **`:quality`** the *operation* value in the **`graphics-state`** is not supported and is ignored. This is because operations do not combine sensibly with anti-aliasing and colors with alpha components. Instead, there is now *compositing-mode*. For more information see the page for **`graphics-state`**.

On Microsoft Windows with *drawing-mode* **`:quality`** only Truetype fonts are supported.

The *drawing-mode* of all graphics ports is **`:quality`** by default, except when a graphics port is made in association with another graphics ports (for example, by **`create-pixmap-port`**), in which case the *drawing-mode* is inherited from the "parent" graphics port.

All the interfaces that create graphics ports, or modify a graphics port to draw to another place, take keyword argument **`:drawing-mode`**. Its value *drawing--mode* can be **`:quality`**, **`:compatible`**, or **`nil`** which is interpreted as use the default (either inherited or the global default **`:quality`**). These interfaces are listed in **13.1.1 Creating instances**.

These examples demonstrate features that are available only with *drawing-mode* **`:quality`**:

Rotating a string:

```
(example-edit-file "capi/graphics/catherine-wheel")
```

Using *compositing-mode*.

```
(example-edit-file "capi/graphics/compositing-mode-simple")
```

Using *compositing-mode*.

```
(example-edit-file "capi/graphics/compositing-mode")
```

Using *compositing-mode*, transforming an image.

```
(example-edit-file "capi/graphics/images-with-alpha")
```

13.3 Graphics state

The **`graphics-state`** object associated with each port holds values for parameters such as *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *mask* and *font* which affect graphics ports drawing to that port.

The full set of parameters is described under **`graphics-state`**.

13.3.1 Setting the graphics state

The graphics state values associated with a drawing function call are set by one of three mechanisms.

1. Enumeration in the drawing function call. For example:

```
(draw-line port 1 1 100 100
           :thickness 10
           :scale-thickness nil
           :foreground :red)
```

2. Bound using macros such as **`with-graphics-state`**. For example:

```
(with-graphics-state (port :thickness 10
                           :scale-thickness nil
                           :foreground :red)
  (draw-line port 1 1 100 100)
  (draw-rectangle port 2 2 40 50 :filled t))
```

For common cases of locally changing the transform in the graphics state, there are specific macros:

- **`with-graphics-transform`** just changes the transform like **`with-graphics-state`** with **`:transform`**.

- **with-graphics-transform-reset** allows you to ignore surrounding transformations.
- **with-graphics-translation**, **with-graphics-post-translation**, **with-graphics-scale** and **with-graphics-rotation** perform commonly-used transformations.
- **with-graphics-mask** affects specifically the masking slots.

3. Set by the **set-graphics-state** function. For example:

```
(set-graphics-state port :thickness 10
                    :scale-thickness nil
                    :foreground :red)
```

The first two mechanisms change the graphics state temporarily. The last one changes it permanently in *port*, effectively altering the "default" state.

13.4 Drawing functions

The section describes the various shapes and so on that you can draw with graphics ports, and lists the relevant drawing functions. The graphics state *foreground* parameter is used for the drawing color.

All drawing functions must be called in the same process as the pane. You will need to arrange for that explicitly in contexts other than callbacks on that pane. To call a function explicitly in the pane's process, use **apply-in-pane-process**, **apply-in-pane-process-if-alive**, **execute-with-interface** or **execute-with-interface-if-alive**.

Note: Unlike images, the *foreground* and *background* colors used when drawing shapes described in this section are not pre-multiplied. Displaying images is described in **13.10 Working with images**.

Note: The full set of graphics state parameters is described under **graphics-state**.

13.4.1 Text

You can draw text with the functions **draw-string** and **draw-character**.

To control the font used, see **13.9 Portable font descriptions**.

13.4.2 Simple lines

You can draw straight lines with the functions **draw-line** and **draw-lines**.

You can draw arcs of an ellipse with the functions **draw-arc** and **draw-arcs**.

13.4.3 Simple shapes

You can draw ellipses and polygons with the functions **draw-ellipse**, **draw-rectangle**, **draw-rectangles**, **draw-polygon** and **draw-polygons**.

You can specify whether a shape is drawn in outline or is filled (with the graphics state *foreground* color) by the argument *filled*.

For example, to clear a rectangular region of an output pane, do:

```
(draw-rectangle pane x y width height
                :filled t
                :foreground color
                :compositing-mode :copy
```

```
                    :shape-mode :plain)
```

`:compositing-mode :copy` is needed only when the color has alpha, and **`:foreground`** *color* is needed only if it is different from the *foreground* in *pane*'s **`graphics-state`**.

13.4.4 Paths

A graphics path is a series of lines, arcs and Bézier curves that together specify one or more disconnected figures to be drawn.

You can draw a path with the function **`draw-path`**.

A path can be drawn in outline or can be filled. A path can also be used as the clipping mask.

13.5 How to draw to an on-screen port

Drawing on an **`output-pane`** should almost always happen only inside its *display-callback*. See **`output-pane`** for more information about this initarg.

If you want to display from outside the *display-callback* then you should call **`invalidate-rectangle`** or **`redisplay-element`**, which will cause the *display-callback* to be called.

13.6 Graphics state transforms

Coordinate systems for windows generally have the origin (0,0) positioned at the upper left corner of the window with X positive to the right and Y positive downwards. This is the "window coordinates" system. Generalized coordinates are implemented using scaling, rotation and translation operations such that any Cartesian coordinates can be used within a window. The Graphics Ports system uses a **`transform`** object to achieve this.

13.6.1 Generalized points

An (x, y) coordinate pair can be transformed to another coordinate system by scaling, rotation and translation. The first two can be implemented using 2 x 2 matrices to hold the coefficients:

If the point P is (x, y) and it is transformed to the point Q (x', y'):

$P \Rightarrow Q$ or $(x, y) \Rightarrow (x', y')$, i.e.

$x' = px + ry$, $y' = qx + sy$.

$$Q = PM, \text{ where } M = \begin{vmatrix} p & q \\ r & s \end{vmatrix}$$

Translation can be included in this if the points P and Q are regarded as 3-vectors instead of 2-vectors, with the 3rd element being unity:

$$\begin{aligned} Q &= PM \\ &= (x \;\; y \;\; 1) \begin{vmatrix} p & q & 0 \\ r & s & 0 \\ u & v & 1 \end{vmatrix} \end{aligned}$$

The coefficients u and v specify the translation.

So, the six elements (*p*, *q*, *r*, *s*, *u*, and *v*) of the 3 x 3 matrix contain all the transformation information. These elements are stored in a list (of type **`transform`**) in the **`graphics-state`** slot *transform*.

Transforms can be combined by matrix multiplication to effect successions of translation, scaling and rotation operations.

Functions are provided in Graphics Ports which apply translation, scaling and rotation to a transform, combine transforms by pre- or post-multiplication, invert a transform, perform some operations while ignoring an established transform, and so on. The macros **`with-graphics-rotation`**, **`with-graphics-scale`** and **`with-graphics-translation`** pre-multiply a supplied transform while a body of code is executed.

13.6.2 Drawing on screen

Drawing functions such as **`draw-line`** and **`draw-ellipse`** modify pixels, but you cannot assume that they have exactly the same effect on all platforms. Some platforms might put pixels below and to the right of integer coordinates (x y) while others may center the pixel at (x y).

This applies to all the drawing functions which are documented in **22 GRAPHICS-PORTS Reference Entries -** see the entries for functions with names beginning **`draw-`**.

13.7 Combining source and target pixels

This section describes how new drawings are combined with the existing pixel values in the target of the drawing to generate the result, according to graphics state parameters *compositing-mode* or *operation*.

Note: The full set of graphics state parameters is described under **`graphics-state`**.

13.7.1 Combining pixels with :compatible drawing

When the port's *drawing-mode* is **`:compatible`** the graphics state parameter *operation* determines how the colors are combined, and *compositing-mode* is ignored.

The allowed values of *operation* are the values of the Common Lisp constants **`boole-1`**, **`boole-and`** and so on. These are the allowed values of the first argument to the Common Lisp function **`boole`**. See the specification of **`boole`** in the ANSI Common Lisp standard for the full list of operations.

The color combination corresponds to the logical operation defined there, as if by calling:

```
(boole operation new-pixel screen-pixel)
```

For example, passing **`:operation`** **`boole-andc2`** provides a **`graphics-state`** where graphics ports drawing functions draw with the bitwise AND of the *foreground* color and the complement of the existing color of each pixel.

Note: Graphics State *operation* is not supported by Cocoa/Core Graphics so this parameter is ignored on Cocoa.

13.7.2 Combining pixels with :quality drawing

When the port's *drawing-mode* is **`:quality`** the graphics state parameter *compositing-mode* determines how the colors are combined, and *operation* is ignored.

compositing-mode **`:over`** means draw over the existing values, blending alpha values if they exist.

compositing-mode **`:copy`** means that the source is written to the destination ignoring the existing values. If the source has alpha and the target does not, that has the effect of converting semi-transparent source to solid. **`:copy`** is especially useful for creating transparent and semi-transparent pixmap ports, which can be displayed directly or converted to images by **`make-image-from-port`**.

Further *compositing-mode* values are supported on later versions of Cocoa and GTK+.

13.8 Pixmap graphics ports

Pixmap graphics ports are drawing destinations which exist only as pixel arrays whose contents are not directly accessible. They can be drawn to using the **draw-***thing* functions (for example **draw-string**), they can be used as the port for loading images using **load-image**, and their contents can be copied onto other graphics ports. However this copying can be meaningless unless the conversion of colors uses the same color device on both ports. Because color devices are associated with regular graphics ports (windows) rather than pixmap graphics ports, you have to connect a pixmap graphics port to a regular graphics port for color conversion. This is the main role of the *port* argument of **with-pixmap-graphics-port** and **create-pixmap-port**. The conversion of colors to color representations is done in the same way as for regular graphics ports, but the pixmap graphics port's owner is used to find a color device. You can draw to pixmap graphics ports using pre-converted colors to avoid color conversion altogether, in which case a null color owner is OK for a pixmap graphics port.

13.8.1 Relative drawing in pixmap graphics ports

Many of the drawing functions have a *relative* argument. If non-nil, it specifies that when drawing functions draw to the pixmap, the extremes of the pixel coordinates reached are accumulated. If the drawing strays beyond any edge of the pixmap port (into negative coordinates or beyond its width or height), then the drawing origin is shifted so that it all fits on the port. If the drawing extremes exceed the total size available, some are inevitably lost. If *relative* is **nil**, any part of the drawing which extends beyond the edges of the pixmap is lost. If *relative* is **nil** and *collect* non-nil, the drawing bounds are collected for later reading, but no relative shifting of the drawing is performed. The collected bounds are useful when you need to know the graphics motion a series of drawing calls causes. The *rest* args are host-dependent. They usually include a **:width** and **:height** pair.

13.9 Portable font descriptions

Portable font descriptions are designed to solve the following problems:

- Specify enough information to uniquely determine a real font.
- Query which real fonts match a partial specification.
- Allow font specification to be recorded and reused in a later run.

All the functions described below are exported from the **gp** package.

You can obtain the names of all the fonts which are available for a given pane by calling **list-all-font-names**, which returns a list of partially-specified font descriptions.

Portable font descriptions are used only for lookup of real fonts and for storing the parameters to specify when doing a font lookup operation. To draw text in a specified font using the Graphics Ports drawing functions, supply in the graphics state a font object as returned by **find-matching-fonts** and **find-best-font**.

13.9.1 Font attributes and font descriptions

Font attributes are properties of a font, which can be combined to uniquely specify a font on a given platform. There are some portable attributes which can be used on all platforms; other attributes are platform-specific and will be ignored or signal errors when used on the wrong platform.

Font descriptions are externalizable objects which contain a set of font attributes. When using a font description in a font lookup operation, missing attributes are treated as wildcards (as are those with value **:wild**) and invalid attributes signal errors. The result of a font lookup contains all the attributes needed to uniquely specify a font on that platform.

The **:stock** font attribute is special: it can be used to reliably look up a system font on all platforms.

Font descriptions can be manipulated using the functions **merge-font-descriptions** and **augment-font-description**.

These are the current set of portable font attributes and their portable types:

Set of portable font attributes

Attribute	Possible values	Comments
:family	string	Values are not portable.
:weight	**(member :normal :bold)**	
:slant	**(member :roman :italic)**	
:size	**(or (eql :any) (integer 0 *))**	**:any** means a scalable font
:stock	**(member :system-font :system-fixed-font)**	Stock fonts are guaranteed to exist.
:charset	keyword	

13.9.2 Fonts

Fonts are the objects which are actually used in drawing operations. They are made by a font lookup operation on a pane, using a font description as a pattern.

Examples of font lookup operations are **find-best-font** and **find-matching-fonts**.

Once a font object is resolved you can read its properties such as height, width and average width. The functions **get-font-height**, **get-font-width** and **get-font-average-width** and so on need a pane that has been created. In general, you need to call these functions within **interface-display**, or a *display-callback* or possibly a *create-callback*. See the manual page for **interface** for more information about these initargs.

13.9.3 Font aliases

You can define font aliases, which map a keyword symbol to some font or font description, using **define-font-alias**. You can then use this the keyword as the *font* for CAPI panes.

13.10 Working with images

Graphics Ports supports drawing images, and also reading/writing them from/to file via your code. A wide range of image types is supported. Also, several CAPI classes support the same image types.

To draw an image with Graphics Ports, you need an **image** object which is associated with an instance of **output-pane** (or a subclass of this). You can create an **image** object from:

- A file of recognized image type.
- A registered image identifier (see **13.10.4 Registering images**).
- An **external-image** object.
- A graphics port.

Draw the image to the pane by calling **draw-image**. Certain images ("Plain Images") can be manipulated via the Image Access API. The image should be freed by calling **free-image** when you are done with it.

The CAPI classes **image-pinboard-object**, **button**, **list-panel**, **list-view**, **tree-view**, **toolbar**, **toolbar-button** and **toolbar-component** all support images. There is also limited support for images in **menu**. These classes handle the drawing and freeing for you.

13.10.1 Image formats supported for reading from disk and drawing

This table lists the formats supported at the time of writing:

Operating system and supported image types

OS	Supported Image Types
Microsoft Windows	BMP, DIB, GIF, JPEG, PNG, TIFF, EMF, ICO
macOS	BMP, DIB, GIF, JPEG, TIFF, PICT and many others. Also EPS, PDF
GTK+	BMP, DIB, GIF, JPEG, PNG, TIFF and many others.
X11/Motif	BMP, DIB, GIF, JPEG, PNG, TIFF, XPM, PGM, PPM

Functions which load images from a file attempt to identify the image type from the file type.

Call the function **list-known-image-formats** to list the formats that the current platform supports for reading and drawing.

Note: On X11/Motif, LispWorks uses the freeware **imlib2** library on Linux, FreeBSD and macOS, and **imlib** on Solaris.

Note: On Microsoft Windows, ICO images are supported for certain situations such as buttons and drawing images. See **button** and **draw-image** for details.

Note: On Microsoft Windows, LispWorks additionally supports Windows Icon files with scaling - see **load-icon-image** for details.

Note: On Microsoft Windows, only bitmaps with maximum 24 bits per pixel are supported.

Note: LispWorks 4.3 and previous versions supported only Bitmap images.

13.10.2 Image formats supported for writing to disk

Graphic images can be written to files in several formats, using **externalize-and-write-image**.

All platforms can write at least BMP, JPG, PNG and TIFF files. Call the function **list-known-image-formats** with optional argument *for-writing-too* **t** to list the formats that the current platform supports for writing.

On Microsoft Windows and Cocoa you can also write GIF files, while on GTK+ you can also write ICO and CUR (cursor) files. The cursor files that are written with GTK+ can be used on Windows and Cocoa, although on Cocoa it does not recognize the hot-spot in a CUR file.

There is a simple example of writing a PNG image here:

```
(example-edit-file "capi/graphics/images-with-alpha")
```

13.10.3 External images

An External Image is an intermediate object. It is a representation of a graphic but is not associated with a port and cannot be used directly for drawing. It is a Lisp object which can be loaded into Lisp and saved in a LispWorks image created by **save-image** or **deliver**.

An object of type **external-image** is created by reading an image from a file, or by externalizing an **image** object, or by copying an existing **external-image**. Or, if you have the image bitmap data, you can create one directly as in this example:

```
(example-edit-file "capi/buttons/buttons")
```

The **external-image** contains the bitmap data, potentially compressed. You can copy **external-image** objects, or write them to file, or compress the data.

You cannot query the size of the image in an **external-image** object directly. To get the dimensions without actually drawing it on screen see **13.8 Pixmap graphics ports**.

An **external-image** can be written to a file using **write-external-image**. If you create an **image** and want to externalize it to write it to file, follow this example:

```
(let ((image (gp:make-image-from-port pane 10 10 200 200)))
  (unwind-protect
      (gp:externalize-and-write-image pane image filename)
(gp:free-image pane image)))
```

13.10.3.1 Converting an external image

Convert an **external-image** to an object of type **image** ready for drawing to a port in several ways as described in **13.10.5 Making an image that is suitable for drawing**. Such conversions are cached but you can remove the caches by **clear-external-image-conversions**.

You can also convert an **image** to an **external-image** by calling **externalize-image**.

13.10.3.2 Transparency and the alpha channel

Graphics ports images support an alpha channel, as long as the image format does.

An External Image representing an image in a format with a color table but with no alpha channel (such as 8-bit BMP) can simulate transparency by specifying an index to represent the transparent color. When converted this color is replaced by the *background* color of the port (which is documented in **simple-pane**).

You can specify the transparent color by:

```
(gp:read-external-image file :transparent-color-index 42)
```

or by:

```
(setf
 (gp:external-image-transparent-color-index
  external-image) 42)
```

You can use an image tool such as Gimp (**www.gimp.org**) to figure out the transparent color index.

On platforms other than Motif you can actually make the background of such an image format truly transparent when displayed. To do this, supply *transparent-color-index* as a cons **(***index* **. :transparent)**.

Note: *transparent-color-index* works only for images with a color map - those with 256 colors or less.

13.10.4 Registering images

One way to load an **image** is via a registered image identifier.

Registering an external image is the way to pre-load images while building an application. To do this, establish a registered image identifier by calling **register-image-translation** at build time:

```
(gp:register-image-translation
 'info-image
  (gp:read-external-image "info.bmp"
                          :transparent-color-index 7))
```

Then at run time obtain the **image** object by:

```
(gp:load-image port 'info-image)
```

13.10.5 Making an image that is suitable for drawing

To create an **image** object suitable for drawing on a given pane, use one of **convert-external-image**, **read-and-convert-external-image**, **load-image**, **make-image-from-port**, **make-sub-image**, **make-scaled-sub-image** or (on Microsoft Windows) **load-icon-image**.

Images need to be freed after use. When the pane that an image was created for is destroyed, the image is freed automatically. However if you want to remove the image before the pane is destroyed, you must make an explicit call **free-image**. If the image is not freed, then a memory leak will occur.

Another way to create an **image** object is to supply a registered image identifier in a CAPI class that supports images. For example you can specify an *image* in an **image-pinboard-object**. Then, an **image** object is created implicitly when the pinboard object is displayed and freed implicitly when the pinboard object is destroyed.

In all cases, the functions that create the **image** object require the pane to be already created. So if you are displaying the image when first displaying your window, take care to create the **image** object late enough, for example in the **:before** method of **interface-display** on the window's interface class, or in the first **:display-callback** of the pane.

13.10.6 Querying image dimensions

To obtain the pixel dimensions of an image, load the image using **load-image** and then use the readers **image-width** and **image-height**. The first argument to **load-image** must be a pane in a displayed interface.

To query the dimensions before displaying anything you can create and "display" an interface made with the **:display-state :hidden** initarg. Call **load-image** with this hidden interface and your **external-image** object, and then use the readers **image-width** and **image-height**.

13.10.7 Drawing images

The function to draw an image is **draw-image**.

As with the other drawing functions, this must be called in the same process as the pane, as outlined in **13.4 Drawing functions**.

13.10.8 Image access

You can read and write pixel values in an **image** via an Image Access object, but only if the image is a Plain Image. You can ensure you have a Plain Image by using the result of:

```
(load-image pane image :force-plain t)
```

To read and/or write pixel values, follow these steps:

1. Start with a Graphics Port (for example an **output-pane**) and an **image** object associated with it, which is a Plain Image. See above for how to create an **image** object.
2. Construct an Image Access object by calling **make-image-access**.
3. To read pixels from the image, first call **image-access-transfer-from-image** on the Image Access object. This notionally transfers all the pixel data from the window system into the access object. It might do nothing if the window system allows fast access to the pixel data directly. Then call **image-access-pixel** with the coordinates of each pixel (or use **image-access-pixels-to-bgra**). The values are color representations like those returned from **convert-color** and can be converted to RGB using **unconvert-color** if required.
4. To write pixels to the image, you must have already called **image-access-transfer-from-image**. Then call **(setf image-access-pixel)** with the coordinates of each pixel (or use **image-access-pixels-from-bgra**) to write pre-multiplied pixel RGB values and then call **image-access-transfer-to-image** on the Image Access object. This notionally transfers all the pixel data back to the window system from the access object. It might do nothing if the window system allows fast access to the pixel data directly.
5. Free the image access object by calling **free-image-access** on it.

It is also possible to get all the pixels into a single vector, where each color is represented by four elements, using **image-access-pixels-from-bgra**, and to change all the pixels in the image to values from a vector using **image-access-pixels-to-bgra**. When accessing many pixels, using these functions and accessing the vector is much faster than using the single pixel access.

There is an example that demonstrates the uses of Image Access objects in:

```
(example-edit-file "capi/graphics/image-access")
```

This further example demonstrates the uses of Image Access objects with colors that have an alpha component:

```
(example-edit-file "capi/graphics/image-access-alpha")
```

13.10.8.1 Pre-multiplied pixel values in images

The color values that are received and set using Image Access are *premultiplied*, which means that the value of each of the three components (Red, Green and Blue) are already multiplied by the value of the alpha. This is different from the way colors are represented elsewhere. The functions **color-to-premultiplied** and **color-from-premultiplied** can be used the convert between premultiplied colors and ordinary colors, although they lose some precision in the process.

For example, the form below creates an image from a pixmap filled with a color that has alpha 0.5. When accessing the image using Image Access, the values in the color that it returned are half of the values in the original color.

```
(let* ((initial-color (color:make-rgb 0.8 0.6 0.4 0.5))
       (image-pixel
        (let ((pane (capi:editor-pane
                     (capi:find-interface 'lw-tools:listener))))
          ;; Make a temporary pixmap filled with the
          ;;  initial-color and create a gp:image from it
```

```
              (let ((image (gp:with-pixmap-graphics-port
                                 (pixmap pane 10 10
                                         :background initial-color
                                         :clear t)
                               (gp:make-image-from-port pixmap))))
                ;; Create a gp:image-access, read
                ;;  a pixel and unconvert it
                (let ((image-access (gp:make-image-access
                                     pane image)))
                  (gp:image-access-transfer-from-image
                   image-access)
                  (let ((pixel (color:unconvert-color
                                pane
                                (gp:image-access-pixel
                                 image-access 0 0))))
                    (gp:free-image-access image-access)
                    (gp:free-image pane image)
                    pixel))))))
      (flet ((output-color (string color)
               (format t
                       "~%~a~28t: Red ~4,2f, Green ~4,2f, Blue ~4,2f"
                       string
                       (color:color-red color)
                       (color:color-green color)
                       (color:color-blue color))))
        (output-color "Initial-color"
                      initial-color)
        (output-color "premultiplied"
                      (color:color-to-premultiplied initial-color))
        (output-color "In the image"
                      image-pixel)
        (output-color "Pixel un-premultiplied"
                      (color:color-from-premultiplied image-pixel))))
```

13.10.9 Creating external images from Graphics Ports operations

To create an **external-image** object from graphics ports operations, use **with-pixmap-graphics-port**, and in the scope of it do the drawing and then use **make-image-from-port** to create an **image** object. You can then use **externalize-image** or **externalize-and-write-image** to externalize the image.

```
(defun record-picture (output-pane)
  (gp:with-pixmap-graphics-port
      (port output-pane
            400 400
            :clear t
            :background :red)
    (gp:draw-rectangle port 0 0 200 200
                       :filled t
                       :foreground :blue)
    (let ((image (gp:make-image-from-port port)))
      (gp:externalize-image port image))))
```

Here *output-pane* must be a displayed instance of **output-pane** (or a subclass). The code does not affect the displayed pane.

If you do not already display a suitable output pane, you can create an invisible one like this:

```
(defun record-picture-1 ()
  (let* ((pl (make-instance 'capi:pinboard-layout))
         (win (capi:display
               (make-instance 'capi:interface
                              :display-state :hidden
                              :layout pl))))
```

```
        (prog1 (record-picture pl)
          (capi:destroy win))))
```

Note: There is no reason to create and destroy the invisible interface each time a new picture is recorded, so for efficiency you could cache the interface object and use it repeatedly.

14 Graphic Tools drawing objects

The drawing objects of Graphic Tools add a mechanism to creates a hierarchy of drawing, when a "drawing" is (typically) a simple Graphics Ports drawing operation. The hierarchy specifies the geometry of each node in the hierarchy, so the whole group of drawings can be manipulated as a single object.

The lower level interface allows you to create drawing objects and manipulate them. The higher level interface allows you to generate graphs of functions or bar charts, where "generate" means create a hierarchy of drawing objects. The higher level functions are useful on their own, but they also give examples of how to create high-level objects from drawing objects. You can look at their output to get a better idea how to write your own Graphic Tools code.

The Graphic Tools interface is defined in the package LW-GT. To use it, you need to load the "graphic-tools" module:

```
(require "graphic-tools")
```

14.1 Lower level - drawing objects and objects displayers

The drawing objects are instances of subclasses of **`drawing-object`**. The term "drawing-object-spec" refers to either a **`drawing-object`** or a list of "drawing-object-specs". The drawing objects hierarchy is made of "drawing-object-specs".

The leaf nodes in the hierarchy are **`drawing-object`**s which actually do the drawing, typically by calling a Graphics Ports drawing function (for example **`draw-line`**). You generate such a **`drawing-object`** by using any of the **`lw-gt:make-draw-…`** functions, for example **`make-draw-line`**. You can also have a **`drawing-object`** that calls an arbitrary function by using **`make-a-drawing-call`**.

The non-leaf nodes in the hierarchy are made by instances of **`compound-drawing-object`**. **`compound-drawing-object`** has a *sub-object* slot, which contains a "drawing-object-spec" (either a list of "drawing-object-specs" or a **`drawing-object`**). Since the elements in lists are themselves "drawing-object-specs", that is can also be lists, part of the hierarchy can be done in lists of lists.

The main function of **`compound-drawing-object`** is to define the geometry of the drawing. The actual objects are instances of **`geometry-drawing-object`** which is a subclass of **`compound-drawing-object`**. These objects define the geometry, by rebinding the Graphics Ports transform, and then drawing their *sub-object* in this context. The width and height of the **`compound-drawing-object`** are also passed down, so **`geometry-drawing-object`**s inside the *sub-object* can use it when computing their own geometry.

You create a **`geometry-drawing-object`** by using one of:

`position-object` Defines the rectangle for drawing the *sub-object*.

`fit-object` Scales its *sub-object*.

`position-and-fit-object`

Both positions and scales.

`rotate-object` Rotates its *sub-object*.

`make-absolute-drawing*` and **`make-absolute-drawing`**

Draw their *sub-object* in the translated position, but without scaling or rotation.

Lists just draw their elements in the same geometry as their "parent".

To actually be drawn, the root of the hierarchy must be stored in the *drawing-object* slot of an "objects displayer", which is either an **objects-displayer** (subclass of **pinboard-layout**), or **pinboard-objects-displayer** (subclass of **pinboard-object**). The **objects-displayer** or **pinboard-objects-displayer** displays the hierarchy starting from the object in their *drawing-object* slot, passing its own geometry. The object in the *drawing-object* slot will typically be a list (which then draws its elements) or a **compound-drawing-object** (which then draws its *sub-object* with modified geometry). This process recurses and draws the entire hierarchy.

By default, both **objects-displayer** and **pinboard-objects-displayer** use an internal metafile as a way to cache the drawing and also to improve resizing.

drawing-objects do not have a permanent notion of "parent", and can appear concurrently as "children" of many "parents", and the same applies to a list in the hierarchy. The objects do not have any specific thread information and drawing does not modify anything in the objects. Therefore "drawing-object-specs" can appear concurrently in many places, whether inside the same hierarchy or in different hierarchies.

For example, the following **do-object** function takes an object, and positions it at the bottom (with no positioning), middle and top. It then groups these three occurrences in a list ("drawing-object-spec"). It then uses "drawing-object-spec" twice, once inside **pinboard-objects-displayer**, and once in an **objects-displayer** that also displays the **pinboard-objects-displayer**. Thus the object is displayed six times: bottom, middle and top of the **pinboard-objects-displayer**, and bottom, middle and top of **objects-displayer**.

```
(defun do-object (the-object height)
  (let* ((bottom-one the-object)
         (middle-one
          (lw-gt:position-object the-object
                                 :bottom-ratio 0.5
                                 :bottom-margin (/ height -2)))
         (top-one
          (lw-gt:position-object the-object
                                 :bottom-ratio 1
                                 :bottom-margin (- height)))
         (drawing-object-spec
          (list bottom-one middle-one top-one))
         (pinboard-object
          (lw-gt:make-pinboard-objects-displayer
           drawing-object-spec
           :x 80
           :y 40
           :width 100
           :height 200 )))
    (capi:contain
     (make-instance 'lw-gt:objects-displayer
                    :description (list pinboard-object)
                    :drawing-object drawing-object-spec))))
```

We then use **do-object** to display a red rectangle:

```
(do-object
 (lw-gt:make-draw-rectangle 0 0 40 20 :filled t :foreground :red)
 20)
```

You see that there are six rectangles. When you resize the pane, the three rectangles on the left, which are the rectangles in the *drawing-object* slot of the **objects-displayer**, resize too. That is because the metafile of the **objects-displayer** resizes. The three rectangles of the **pinboard-objects-displayer** do not resize, because the **pinboard-objects-displayer** does not change its size.

The function can be used for more complex objects:

```
(do-object
 (list
  (lw-gt:make-draw-rectangle 0 0 40 20
                             :filled t :foreground :red)
  (lw-gt:make-draw-ellipse 20 10 20 10
                           :filled t :foreground :blue)
  (lw-gt:make-draw-line 0 10 40 10
                        :filled t :foreground :green))
 20)
```

The next example uses **`rotate-object`**. This first shifts the object to the right and down by using **`position-object`**, rotates the objects six times, rotating pi/3 each time, around a point which is in the middle of the height of the object, and distance of height to its left. Note that consequently the actual position of the copies is quite different from where **`position-object`** put them, which is a slightly counter-intuitive feature of **`rotate-object`** when using a rotating point which is not the center of the object:

```
(defun do-rotating (the-object height)
  (let ((shifted
         (lw-gt:position-object the-object
                                :left-margin height
                                :bottom-margin (- (/ height 2)))))
    (let* ((rotated-copies
            (loop repeat 6
                  for angle from 0 by (/ pi 3)
                  collect (lw-gt:rotate-object shifted angle)))
           ;; position the result in the middle of the pane
           (positioned-drawing
            (lw-gt:position-object rotated-copies
                                   :bottom-ratio 0.5
                                   :left-ratio 0.5)))
      (capi:contain
       (make-instance 'lw-gt:objects-displayer
                      :drawing-object positioned-drawing)))))
```

and rotate the same object that we used above:

```
(do-rotating
 (list (lw-gt:make-draw-rectangle 0 0 40 20
                                  :filled t :foreground :red)
       (lw-gt:make-draw-ellipse 20 10 20 10
                                :filled t :foreground :blue)
       (lw-gt:make-draw-line 0 10 40 10
                             :filled t :foreground :green))
 20)
```

A sub-hierarchy inside a hierarchy can be modified destructively by setting the *sub-object* slot of **`compound-drawing-object`**s in the hierarchy. For example, we use the function **`do-object`** above to display rectangles, and then make it switch between rectangles and ellipses:

```
(let ((rect
       (lw-gt:make-draw-rectangle 0 0 40 20
                                  :filled t :foreground :red))
      (ellipse
       (lw-gt:make-draw-ellipse 20 10 20 10
                                :filled t :foreground :blue)))
  (let ((my-object
         ;; Use lw-gt:position-object to create a
         ;;  compound-drawing-object, without actual positioning
         (lw-gt:position-object rect)))
    (let ((the-pane (do-object my-object 20)))
      (dotimes (x 20)
        (sleep 0.5)
```

```
        ;; modify the hierarchy
        (setf (lw-gt:compound-drawing-object-sub-object my-object)
              (if (evenp x) ellipse rect))
        ;; make it redraw
        (lw-gt:force-objects-redraw the-pane)))))
```

In principle you can also modify the hierarchy by setting the **cl:car** of a cons in a list inside the hierarchy, though that will make your code less clear. Do not set the **cl:cdr** of conses in these lists.

As the example above shows, you do not need to do modifications in the pane thread (in contrast to operations on CAPI objects). If you modify the hierarchy while it is being drawn, the drawing in this drawing operation may be mixed up. However, normally you will want to force it to redraw using **force-objects-redraw**, which will draw correctly.

To make it easier to modify objects in the hierarchy, the functions that generate **compound-drawing-object**s all take keyword arguments *data* and *function*, which then are used to update the object automatically by calls to **compute-drawing-object-from-data** or **recurse-compute-drawing-object**. For example, the switch example above can be written using this mechanism, without having to remember *my-object*:

```
(defun my-updating-function (data)
  (car data))

(let ((data (list nil)))
  (let ((rect
          (lw-gt:make-draw-rectangle 0 0 40 20 :filled t :foreground :red))
         (ellipse
          (lw-gt:make-draw-ellipse 20 10 20 10 :filled t :foreground :blue)))
    (let ((my-object
            ;; Use position-object to create a compound-drawing-object,
            ;;  without actual positioning, but with updating information
            (lw-gt:position-object rect
                                   :function 'my-updating-function
                                   :data data)))
      (let ((the-pane (do-object my-object 20)))
        (dotimes (x 20)
          (sleep 0.5)
          (setf (car data) (if (evenp x) ellipse rect))
          (lw-gt:recurse-compute-drawing-object the-pane))))))
```

Because **drawing-object**s do not actually know which hierarchy they are in, they cannot tell their containing pane to redraw. We used **force-objects-redraw** in the first example above, and in the last example above we rely the fact that **recurse-compute-drawing-object**, when called on a pane, does this itself. In general, to actually get the pane redrawn, you will have to have a call of some function (**force-objects-redraw** or a function that calls it) on either the pane or on a **pinboard-objects-displayer**.

Note that just invalidating the pane (by **invalidate-rectangle**) does not cause redrawing of the **drawing-object**s when a metafile is used (the default case). That is intentional, to make exposure and resize fast.

Modifying the hierarchy is thread-safe, in that threads modifying the hierarchy in parallel, and even parallel to it being drawn, will not cause a problem on its own. However there is no guard against different threads making conflicting changes. For example, if thread A sets the *sub-object* of a **compound-drawing-object**, and at the same time thread B sets something inside the *sub-object*, then the change that thread B made will not be visible in the hierarchy. You will have to guard against such conflicts.

The **drawing-object** code cannot cope with a circular hierarchy.

14.2 Higher level - drawing graphs and bar charts

The higher level Graphic Tools functions all generate a "drawing-object-spec" (a **drawing-object** or a list) which can then be displayed by inclusion in the hierarchy under an **objects-displayer** (potentially via a **pinboard-objects-displayer**).

The functions are geared towards producing graphs of (mathematical) functions and bar charts. The function **generate-grid-lines** is used to generate grid of lines. The function **generate-labels** is used to generate labels, with the intention that these labels will match the grid lines.

The functions **generate-graph-from-pairs** and **generate-graph-from-graph-spec** are used to generate the actual graph. The graph is actually a sequence of straight lines connecting consecutive points (neighbouring points in the x dimension), but by giving it enough points the graph can be made to look smooth. Currently there is no smoothing option.

generate-graph-from-pairs receives the points as a list of lists (x y). **generate-graph-from-graph-spec** takes a **basic-graph-spec** which you make by calling **make-basic-graph-spec**. The graph spec contains a function which computes the y value corresponding to the supplied x value, and information (start, step and range) which specifies the x values to use. The **basic-graph-spec** is intended to simplify writing code that repeatedly draws graphs with similar attributes.

generate-bar-chart generates the bars of a bar chart, with an optional title for each bar.

To show something useful, you will normally combine the results of **generate-grid-lines**, **generate-labels** and one of **generate-graph-from-pairs**, **generate-graph-from-graph-spec** or **generate-bar-chart** (typically by just using **cl:list**), and then position and scale the result using the geometry functions (**position-object**, **fit-object**, **position-and-fit-object**), and the result of this will be put into a hierarchy under an **objects-displayer** or **pinboard-objects-displayer**.

Note that when you scale (using **fit-object** or **position-and-fit-object**), you effectively change the units of drawing inside the scaled object. You can therefore generate the graph in its natural coordinates, and then put in the correct dimensions on the screen. The example below generates a graph with size of 18x9, and then uses **fit-object** with the same width and height, which scales the graph to fit the full area that it is supplied. We also give it some margin using **position-object**.

We then use the result (fitted-graph-with-margin) both as the *drawing-object* of a **pinboard-objects-displayer** and the *drawing-object* of an **objects-displayer** which also contains the **pinboard-objects-displayer**. In the **pinboard-objects-displayer** we also add a red rectangle to show the area of the **pinboard-objects-displayer**. The result is that the the same graph is displayed twice: once inside **pinboard-objects-displayer** and once inside the whole **objects-displayer**. If you resize the window, you see that the outer graph resizes, while the inner graph stays the same (because the **pinboard-objects-displayer** does not change size).

```
(let* ((graph
        (lw-gt:generate-grid-lines :horizontal-count 18
                                   :vertical-count 9
                                   :right-thickness 3
                                   :major-x-step 4
                                   :major-y-step 3
                                   :thickness 1
                                   :major-thickness 2
                                   :major-color :blue
                                   :color :green))
       (fitted-graph (lw-gt:fit-object graph 18 9))
       (fitted-graph-with-margin
        (lw-gt:position-object fitted-graph
                               :left-margin 10
                               :right-margin 10
                               :top-margin 10
                               :bottom-margin 10))
       (red-rectangle
        (lw-gt:fit-object
```

```
            (lw-gt:make-draw-rectangle 0 0 1 1
                                       :foreground :red
                                       :thickness 2
                                       :scale-thickness nil)
                             1 1))
          (pinboard-object (lw-gt:make-pinboard-objects-displayer
                            (list red-rectangle fitted-graph-with-margin)
                            :x 45 :y 45 :width 400 :height 400)))
     (setq *pane* (capi:contain (make-instance 'lw-gt::objects-displayer
                                  :description (list pinboard-object)
                                  :drawing-object fitted-graph-with-margin
                                  )
                   :best-width 500 :best-height 500)))
```

For the **pinboard-object** to resize, you need to resize it explicitly.

The following function moves the first pinboard object:

```
(defun move-first-pinboard-object (pane x y width height)
  (capi:apply-in-pane-process
   pane
   #'(lambda (pane x y width height)
       (let ((po (car (capi:layout-description pane))))
         (setf (capi:static-layout-child-geometry po)
               (values x y width height))))
                              pane x y width height))
```

Now this moves the pinboard object, and resizes the grid inside it (as well as the red rectangle):

```
(move-first-pinboard-object *pane* 20 60 420 300)
```

More extended are examples are in:

```
(example-edit-file "graphic-tools/bar-chart-example")
```

```
(example-edit-file "graphic-tools/graph-example")
```

15 The Color System

The LispWorks Color System allows you to manipulate colors, which are used as the color values in Graphics Ports and CAPI functions. For example, to draw a string in red, you call:

```
(gp:draw-string pane string x y :foreground :red)
```

The value of **`:foreground`** (**`:red`** above) must be a color specification that is recognized by the Color System (**`:red`** is recognized because it is part of the color database that is pre-loaded)..

In the LispWorks Color System, colors can be represented in two ways:

1. A color spec, which specifies a color model (for example RGB) and the values of the parameters in this model (for example the parameters in RGB would be the values of the red, green and blue components, and optionally the alpha value).
2. A symbol, normally a keyword. For a symbol to be used a color, it must be associated with a color spec, either directly or via another symbol. Symbols that are used as colors are looked up in a color database. The LispWorks image is supplied with a large color database already loaded (approximately 660 entries), and you can add your own entries using **`define-color-alias`** or by loading your own color database.

The LispWorks Color System allows you to:

- Make your own color specs in RGB, HSV or GRAY color models, and access components of color specs. See **15.1 Color specs**.
- Define new association between symbols and colors, query which association exist, and find the color spec associated with a symbol. See **15.2 Color aliases**.
- Convert color specs between color models. See **15.3 Color models**.
- Load a color database from a file of color descriptions. See **15.4 Loading the color database**.
- Define new color models. See **15.5 Defining new color models**.

The Color System symbols are exported from the COLOR package, and all symbols mentioned in this chapter are assumed to be external to this package unless otherwise stated.

15.1 Color specs

A color spec is an object which numerically defines a color in some color-model. For example the object returned by the call:

```
(color:make-rgb 0.0 1.0 0.0) =>
#(:RGB 0.0 1.0 0.0)
```

defines the color green in the RGB color model. Generally short-floats are used; this results in the most efficient color conversion process. However, any float type can be used.

To find out what color-spec is associated with a color name, use the function **`get-color-spec`**. It returns the color-spec associated with a symbol. If there is no color-spec associated with *color-name*, this function returns **`nil`**. If *color-name* is the name of a color alias, the color alias is dereferenced until a color-spec is found.

Color-specs are made using standard functions **make-rgb**, **make-hsv** and **make-gray**. For example:

```
(make-rgb 0.0s0 1.0s0 0.0s0)
(make-hsv 1.2s0 0.5s0 0.9s0)
(make-gray 0.66667s0)
```

To create a color spec with an alpha component using the above constructors, pass an extra optional argument. For example this specifies green with 40% transparency:

```
(make-rgb 0.0s0 1.0s0 0.0s0 0.6s0)
```

You can also make a transparent color using **color-with-alpha**:

(color-with-alpha *color-spec* **0.8s0)**

Note that the alpha component is not supported on Motif.

The function **color-model** returns the model in which a color-spec object has been defined.

The components of color specs can be accessed using the following functions:

RGB model	**color-red**, **color-green**, **color-blue**.
HSV model	**color-hue**, **color-saturation**, **color-value**.
Gray model	**color-level**.

When these readers are supplied a color spec of their model, they just return the corresponding component. If they are supplied a color spec of another model, they compute the component.

The function **color-alpha** can be used to access the alpha value of a color (its opacity). If the color does not have an alpha, **color-alpha** returns 1.0.

15.2 Color aliases

You can enter a color alias in the color database using the function **define-color-alias**. You can remove an entry in the color database using **delete-color-translation**.

define-color-alias makes an entry in the color database under a name, which should be a symbol. LispWorks by convention uses keyword symbols. The name points to either a color-spec or another color name (symbol):

```
(define-color-alias :wire-color :darkslategray)
```

Attempting to replace an existing color-spec in the color database results in an error. By default, replacement of existing aliases is allowed but there is an option to control this (see the manual page for **define-color-alias**).

delete-color-translation removes an entry from the color-database. Both original entries and aliases can be removed:

```
(delete-color-translation :wire-color)
```

As described in **15.1 Color specs**, the function **get-color-spec** returns the color-spec associated with a color alias. The function **get-color-alias-translation** returns the ultimate color name for an alias:

```
(define-color-alias :lispworks-blue
            (make-rgb 0.70s0 0.90s0 0.99s0))
(define-color-alias :color-background
                   :lispworks-blue)
```

```
(define-color-alias :listener-background
                    :color-background)
```

```
(get-color-alias-translation :listener-background)
         => :lispworks-blue
(get-color-alias-translation :color-background)
         => :lispworks-blue
```

There is a system-defined color alias **`:transparent`** which is useful when specified as the *background* of a pane. It is currently supported only on Cocoa. For example:

```
(capi:popup-confirmer
 (make-instance 'capi:display-pane
                :text
                (format nil "The background of this pane~%is transparent")
                :background :transparent)
 "")
```

To find out what colors are defined in the color database, use the function **`apropos-color-names`**. For example:

```
(apropos-color-names "RED") =>
 (:ORANGERED3 :ORANGERED1 :INDIANRED3 :INDIANRED1
  :PALEVIOLETRED :RED :INDIANRED :INDIANRED2
  :INDIANRED4 :ORANGERED :MEDIUMVIOLETRED
  :VIOLETRED :ORANGERED2 :ORANGERED4 :RED1 :RED2 :RED3
  :RED4 :PALEVIOLETRED1 :PALEVIOLETRED2 :PALEVIOLETRED3
  :PALEVIOLETRED4 :VIOLETRED3 :VIOLETRED1 :VIOLETRED2
  :VIOLETRED4)
```

For information about only aliases or only original entries, use **`apropos-color-alias-names`** or **`apropos-color-spec-names`** respectively.

To get a list of all color names in the color database, call **`get-all-color-names`**.

15.3 Color models

Three color models are defined by default: RGB, HSV and GRAY. RGB and HSV allow specification of any color within conventional color space using three orthogonal coordinate axes, while gray restricts colors to one hue between white and black. All color models contain an optional alpha component, though this is used only on Cocoa and Windows.

Color models defined by default

Model	Name	Component: Range
RGB	Red Green Blue	RED (0.0 to 1.0) GREEN (0.0 to 1.0) BLUE (0.0 to 1.0) ALPHA (0.0 to 1.0)
HSV	Hue Saturation Value	HUE (0.0 to 5.99999) SATURATION (0.0 to 1.0) VALUE (0.0 to 1.0) ALPHA (0.0 to 1.0)
GRAY	Gray	GRAY (0.0 to 1.0) ALPHA (0.0 to 1.0)

The Hue value in HSV is mathematically in the open interval [0.0 6.0). All values must be specified in floating point values.

You can convert color-specs between models using the available **ensure-<***model***>** functions. For example:

```
(setf green (make-rgb 0.0 1.0 0.0))
    => #(:RGB 0.0 1.0 0.0)
(eq green (ensure-rgb green)) => T


(ensure-hsv green) => #(:HSV 2.0 0.0 1.0)
(eq green (ensure-hsv green)) => NIL


(ensure-rgb (ensure-hsv green)) => #(:RGB 0.0 1.0 0.0)
(eq green (ensure-rgb (ensure-hsv green))) => NIL
```

Of course, information can be lost when converting to GRAY:

```
(make-rgb 0.3 0.4 0.5) => #(:RGB 0.3 0.4 0.5)
(ensure-gray (make-rgb 0.3 0.4 0.5))
    => #(:GRAY 0.39999965)
(ensure-rgb (ensure-gray
             (make-rgb 0.3 0.4 0.5)))
       => #(:RGB 0.39999965 0.39999965 0.39999965)
```

There is also **ensure-color** which takes two color-spec arguments. It converts if necessary the first argument to the same model as the second. For example:

```
(ensure-color (make-gray 0.3) green)
     => #(:RGB 0.3 0.3 0.3)
```

ensure-model-color takes a model as the second argument. For example:

```
(ensure-model-color (make-gray 0.3) :hsv)
     => #(:HSV 0 1.0 0.3)
```

The function **colors=** compares two color-spec objects for color equality.

The function **color-level** returns the gray level of a color-spec, and the functions **color-blue**, **color-green**, **color-red**, **color-hue**, **color-saturation** and **color-value** return the associated components.

The color models above represent the color in a portable (and externalizable) way. To actually use it, the system needs to convert to the representation used by the underlying display system. The user can do the conversion using **convert-color**. The result is called a "converted color" or "color representation" or "color-rep", and is more efficient to use in drawing functions, because it saves the system from doing the conversion each time it uses the color.

15.4 Loading the color database

You can load new color definitions into the color database using **read-color-db** and **load-color-database**.

Given a color definition file **my-colors.db** of lines like these:

```
#(:RGB 1.0s0 0.980391s0 0.980391s0)     snow
#(:RGB 0.972548s0 0.972548s0 1.0s0)     GhostWhite
```

call:

```
(load-color-database (read-color-db "my-colors.db"))
```

The color database is stored in the variable **`*color-database*`**. To clear the database use the form:

```
(setf *color-database* (make-color-db))
```

Note: You should do this before starting the LispWorks IDE (that is, before **`env:start-environment`** is called) or before your application's GUI starts. Be sure to load new color definitions for all the colors used in the GUI. The initial colors were obtained from the **`config/colors.db`** file.

You can remove a color database entry with **`delete-color-translation`**.

15.5 Defining new color models

Before using the definition described here, you should evaluate the form:

```
(require "color-defmodel")
```

The macro **`define-color-models`** can be used to define new color models for use in the color system.

The default color models are defined by the following form:

```
(define-color-models ((:rgb (red 0.0 1.0)
                            (green 0.0 1.0)
                            (blue 0.0 1.0))
                      (:hsv (hue 0.0 5.99999)
                            (saturation 0.0 1.0)
                            (value 0.0 1.0))
                      (:gray (level 0.0 1.0))))
```

For example, to define a new color model YMC and keep the existing RGB, HSV and GRAY models:

```
(define-color-models ((:rgb (red 0.0 1.0)
                            (green 0.0 1.0)
                            (blue 0.0 1.0))
                      (:hsv (hue 0.0 5.99999)
                            (saturation 0.0 1.0)
                            (value 0.0 1.0))
                      (:gray (level 0.0 1.0))
                      (:ymc (yellow 0.0 1.0)
                            (magenta 0.0 1.0)
                            (cyan 0.0 1.0))))
```

You must then define some functions to convert YMC color-specs to other color-specs. In this example, those functions are named:

```
make-ymc-from-rgb
make-ymc-from-hsv
make-ymc-from-gray
```

and:

```
make-rgb-from-ymc
make-hsv-from-ymc
make-gray-from-ymc
```

You can make this easier, of course, by defining the functions:

```
make-ymc-from-hsv
make-ymc-from-gray
```

```
make-hsv-from-ymc
make-gray-from-ymc
```

in terms of **make-ymc-from-rgb** and **make-rgb-from-ymc**.

If you never convert between YMC and any other model, you need only define the function **make-rgb-from-ymc**.

16 Printing from the CAPI—the Hardcopy API

The CAPI hardcopy API is a mechanism for printing a Graphics Port (and hence a CAPI **`output-pane`**) to a printer. It is arranged in a hierarchy of concepts: printers, print jobs, pagination and outputting.

Printers correspond to the hardware accessible to the OS. Print jobs control connection to a printer and any printer-specific initialization. Pagination controls the number of pages and which output appears on which page. Outputting is the operation of drawing to a page. This is accomplished using the standard Graphics Ports drawing functions discussed in **13 Drawing - Graphics Ports**.

Printing is done by using the macro **`with-print-job`** to define a job. Inside its *body* you specify pages to print by either **`with-document-pages`** ("page on demand printing") or **`with-page`** ("page sequential printing"). Inside the *body* of **`with-document-pages`** or **`with-page`** you use normal drawing functions on the variable bound by **`with-print-job`** to draw the page. You normally also use **`with-page-transform`** to specify the transformation to the page area. There are also several functions for simple printing jobs.

16.1 Printers

You can obtain the current printer, or ask the user to select one, by using **`current-printer`**. You can ask the user about configuration by using the functions **`page-setup-dialog`** and **`print-dialog`** which display the standard Page Setup and Print dialogs.

You can pass the printer object (as returned by **`current-printer`** or **`print-dialog`**) to APIs with a *printer* argument, such as **`with-print-job`**, **`page-setup-dialog`** and **`print-dialog`**. The printer object itself is opaque but you can modify the configuration programmatically using **`set-printer-options`**.

16.1.1 Standard shortcut keys in printer dialogs

On Cocoa by default the standard shortcuts **`Command+P`** and **`Command+Shift+P`** invoke **Print...** and **Page Setup...** menu commands respectively.

In Microsoft Windows editor emulation by default the standard shortcut **`Ctrl+P`** invokes a **Print...** menu command.

16.2 Print jobs

A Print job is contained within a use of the macro **`with-print-job`**, which handles connection to the printer and sets up a graphics port for drawing to the printer.

16.3 Handling pages—page on demand printing

In *Page on Demand Printing*, the application provides code to output an arbitrary page. The application should be prepared to print pages in any order. This is the preferred means of implementing printing. Page on Demand printing uses the **`with-document-pages`** macro, which executes the code for each page to be printed, in an unspecified order.

16.4 Handling pages—page sequential printing

Page Sequential Printing may be used when it is inconvenient for the application to implement Page on Demand printing. In Page Sequential Printing, the application outputs each page of the document in order. Page Sequential printing is done by using the **with-page** macro, with each invocation of **with-page** contributing a new page to the document.

Note: **with-page** does not work on Cocoa.

16.5 Printing a page

In either mode of printing, the way in which a page is printed is the same. A suitable transformation must be established between the coordinate system of the **output-pane** or **printer-port** object and the physical page being printed. The page is then drawn using normal Graphics Ports operations, which are described in **13 Drawing - Graphics Ports**.

16.5.1 Establishing a page transform

The **with-page-transform** macro can be used to establish a page transform which controls scaling by mapping a rectangular region of the document to the printable area of the page. The scale matches the screen by default. By specifying a large rectangle, you can get finer granularity in the drawing. Any number of invocations of **with-page-transform** may occur during the printing of a page. For instance, it may be convenient to use a different page transform when printing headers and footers to the page from that used when printing the main body of the page.

A helper function, **get-page-area**, is provided to simplify the calculation of suitable rectangles for use with **with-page-transform**. It calculates the width and height of the rectangle in the user's coordinate space that correspond to one printable page, based on the logical resolution of the user's coordinate space in dpi.

For more specific control over the page transform, the printer metrics can be queried using **get-printer-metrics** and the various **printer-metrics** accessors such as **printer-metrics-height**.

Margins and the printable area can be set using **set-printer-metrics**.

There is an example in:

```
(example-edit-file "capi/printing/fit-to-page")
```

16.6 Other printing functions

To add, remove and configure printers on platforms other than Motif use the system configuration utility. On Microsoft Windows this is the Printer Control Panel. On Cocoa printers are configured via the System Preferences.

A simple printing API is available via **simple-print-port**, which prints the contents of an **output-pane** to a printer.

The Hardcopy API also allows you to print plain text to a printer. To do this, use the functions **print-text**, **print-file** and **print-editor-buffer**, and the macro **with-output-to-printer**.

16.7 Printing on Motif

This section applies only to X11/Motif, where the hardcopy API uses Postscript rather than native printing.

16.7.1 Printer definition files

On Motif, CAPI uses its own printer definition files to keep information about printers. These files contain a few configuration settings, and the name of the PPD file if applicable (see **16.7.2 PPD files** for information about PPD files). When a user saves a printer configuration, the system writes such a file. Note that because the printer definition file contains the name of the PPD file, it must only be moved between machines with care: the PPD file must exist in the same path.

Printer definition files are loaded from directories in the value of **`*printer-search-path*`**.

16.7.2 PPD files

To fully use the functionality of a Postscript printer on Motif, the system needs a Postscript Printer Description (PPD) file, which is a file in a standard format defined by Adobe. It describes the options the printer has and how to control them.

When a print dialog is presented to the user (either by an explicit call to **`print-dialog`**, or by printing), the system uses the PPD file to find what additional options to present, and how to communicate them to the printer.

A PPD file should be supplied by the manufacturer with the printer itself. Otherwise, it is normally possible to obtain the PPD file from the website of the manufacturer. The name of a PPD file should be *printername*.ppd.

When the user configures a new printer, the first thing the system does is to show the user all the PPD files that it can find under the **`*ppd-directory*`** (directly, or one level of directories below it). The application should set this variable to the appropriate directory.

If the value of **`*ppd-directory*`** is **`nil`**, the system looks at the directory obtained by evaluating **`(sys:lispworks-dir "postscript/ppd")`**.

If the printer does not have a PPD file, the user can still use it by selecting the default button in the print dialog. This means that the system will let the user change only the basic properties of the printer, without using its more complex features.

16.7.3 Adding and removing printers

On Motif, printers can be added, removed and configured interactively via **`printer-configuration-dialog`**. Printers can be added and removed programmatically with **`install-postscript-printer`** and **`uninstall-postscript-printer`**.

17 Drag and Drop

This chapter discusses how to implement drag and drop functionality in your CAPI application. The example code in this chapter forms a complete example allowing the user to drag an item from a **`tree-view`** to a **`list-panel`**.

17.1 Overview of drag and drop

A drag and drop operation occurs when the user clicks and holds the mouse button in a pane supporting dragging, then drags to a pane supporting dropping, and releases the mouse button.

Visual feedback may be provided indicating that dragging is happening, whether a drop operation is possible at the current mouse position, and what operation will occur when the user drops. Usually the operation is the transfer of data.

You need to decide which CAPI pane(s) and interfaces will support dragging and then implement it for each, and similarly for dropping. You will implement drag and drop for one or more specified data formats.

17.1.1 Drag and drop with other applications

Certain predefined data formats can be dragged from a CAPI application to another application such as the Windows Explorer or the macOS Finder, and vice versa.

17.1.2 Drag and drop within a CAPI application

When both the drag and the drop phases are within the same CAPI image, you can specify private data formats, in addition to the predefined data formats.

17.2 Dragging

First you should decide which CAPI pane(s) and interfaces will support dragging, and which data formats they will support. Data formats are arbitrary keywords that must be interpreted by the pane where the user can drop.

17.2.1 Dragging values from a choice

To implement dragging in **`list-panel`** or **`tree-view`** supply the **`:drag-callback`** initarg. When the user drags, *drag-callback* receives a list of indices of the choice items being dragged.

The *drag-callback* should return a property list whose keys are the data formats (such as **`:string`** or **`:image`**) to be dragged, along with the values associated with each format.

17.2.1.1 Example: dragging from a tree

This example returns string data for a **`tree-view`** defined below:

```
(defun tree-drag-callback (pane indices)
  (list :string
        (string (elt (capi:collection-items pane)
                     (first indices)))))
```

```
(defun fruits (x)
  (case x
    (:fruits (list :apple :orange))
    (:apple (list :cox :bramley))
    (:orange (list :blood-orange :seville))
    (t nil)))

(capi:contain
 (make-instance 'capi:tree-view
                :title "Fruit tree"
                :roots '(:fruits)
                :children-function 'fruits
                :drag-callback 'tree-drag-callback))
```

There is a further example showing dragging from **`list-panel`**s in:

```
(example-edit-file "capi/choice/drag-and-drop")
```

17.2.2 Dragging within an output-pane

To implement dragging items around within a single **`output-pane`**, include suitable callbacks on these gestures in its *input-model*:

```
(:button-1 :press)
```

```
(:button-1 :motion)
```

In this case it is not necessary to call **`drag-pane-object`** and you can implement dropping in the same pane by a suitable callback for:

```
(:button-1 :release)
```

See this example:

```
(example-edit-file "capi/applications/balloons")
```

17.2.3 Dragging values from an output-pane

To implement dragging from an **`output-pane`** include an appropriate callback on the **`(:button-1 :press)`** gesture in the pane's *input-model*. This callback should call **`drag-pane-object`** with arguments which provide the data formats and values associated with each format. You will also specify *drop-callback* in the destination pane(s), as described in **17.3 Dropping**.

See the example file in:

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

17.2.3.1 Dragging editor-pane text

To implement dragging of text in an **`editor-pane`**, use EDITOR functions such as **`editor:points-to-string`** to obtain the value for the **`:string`** format.

17.2.4 Data formats

`:string` Receives a string, potentially from another application. Is also understood by some other panes that expect text.

`:image` Receives an image on Cocoa and GTK+. The value passed should be an **`image`** object. See **13.10 Working with images** for more information about images.

When supplying an image for dragging (that is, including **`:image`** *image* in the plist argument of **`drag-pane-object`** or in the plist that is returned from the *drop-callback*), the dragging mechanism frees the image (as by **`free-image`**) when it finishes with it (which will be at some indeterminate time later). If you need to pass an image which you want to use later, you should make a copy of it by **`make-sub-image`**.

When receiving an image (by calling **`drop-object-get-object`** with **`:image`**), the received image should also be freed when you finish with it. However, it will be freed automatically when the pane supplied to **`drop-object-get-object`** is destroyed, so you do not need to free it explicitly if freeing can wait (which is probably true in most cases).

See this example:

```
(example-edit-file "capi/choice/list-panel-drag-images")
```

`:filename-list` Receives a list of files. Is understood by other applications such as the macOS Finder and Windows Explorer.

You can also use private formats, named by arbitrary keywords, which will work only in the same Lisp image.

17.2.5 Dragging a Cocoa title bar image

On Cocoa, if there is a drag image in an **`interface`** title bar, then dragging this image will by default return a list containing the **`interface`** *pathname* as **`:filename-list`** data. You could override this by providing a *drag-callback* for the interface.

17.3 Dropping

First you should decide which CAPI pane(s) and interfaces will support dropping, where exactly dropping should be allowed, and what should occur on dropping for each data format that is made available.

17.3.1 The drop callback

To implement dropping in **`list-panel`** or **`tree-view`** or **`output-pane`**, supply the **`:drop-callback`** initarg.

You can also supply **`:drop-callback`** for an **`interface`**. When the user drags an object over a window, the system first tries to call the *drop-callback* of any pane under the mouse and otherwise calls the *drop-callback* of the top-level interface, if supplied.

The *drop-callback* receives as arguments a *drop-object* which is used to communicate information about the dropping operation and *stage* which is a keyword. The *drop-callback* is called at several stages: when the pane is displayed; when the user drags over the pane; and when the user drops over the pane. Various functions are provided which you can use to query the *drop-object* and set attributes appropriately.

You will use **`set-drop-object-supported-formats`** to specify the data formats that it wants to receive. The **`:string`** format can be used to receive a string from another application and the **`:filename-list`** format can be used to receive a list of filenames from another application such as the Macintosh Finder or the Windows Explorer. Any other keyword in formats is assumed to be a private format that can only be used to receive objects from within the same Lisp image.

You can use **drop-object-provides-format** to query whether a given data format is actually available, and then you can call **(setf drop-object-drop-effect)** to modify the effect of the dropping operation .

Finally, at the **:drop** stage, you will use **drop-object-get-object** to retrieve (for each data format) the object which was returned by the *drag-callback*, and then do something with this object, typically copying or moving it to the pane in some way.

17.3.2 Dropping in a choice

Additionally within the *drop-callback* of a **list-panel** or **tree-view** you can use **drop-object-collection-index** (or **drop-object-collection-item**) to query the index (or item) where the object would currently be dropped.

17.3.2.1 Example: dropping in a list

This *drop-callback* simply appends the dropped string at the end of the list:

```
(defun list-drop-callback (pane drop-object stage)
  (format t "list drop callback ~S ~S ~S" pane  drop-object stage)
  (case stage
    (:formats
     (set-drop-object-supported-formats drop-object
                                        (list :string)))
    ((:enter :drag)
     (when (and (drop-object-provides-format drop-object
                                             :string)
                (drop-object-allows-drop-effect-p drop-object
                                                  :copy))
       (setf (drop-object-drop-effect drop-object) :copy)))
    (:drop
     (when (and (drop-object-provides-format drop-object
                                             :string)
                (drop-object-allows-drop-effect-p drop-object
                                                  :copy))
       (setf (drop-object-drop-effect drop-object) :copy)
       (add-list-item pane drop-object)))))

(defun add-list-item (pane drop-object)
  (append-items
        pane
        (list (string-capitalize
               (drop-object-get-object drop-object
                                       pane :string)))))

(contain
 (make-instance 'list-panel
                :title "Shopping list"
                :items (list "Tea" "Bread")
                :drop-callback 'list-drop-callback))
```

Try dragging an item from the **tree-view** created in **17.2.1.1 Example: dragging from a tree**.

Below is a more sophisticated version of **add-list-item** which inserts the item at the expected position within the list. This position is obtained using **drop-object-collection-index**:

```
(defun add-list-item (pane drop-object)
  (multiple-value-bind (index placement)
      (drop-object-collection-index drop-object)
    (list-panel-add-item pane
                         (string-capitalize
                          (drop-object-get-object
                           drop-object pane :string))
```

```
                              index placement)))

    (defun list-panel-add-item (pane item index placement)
      (let ((item-count (count-collection-items pane)))
        (let ((adjusted-index (if (eq placement :above)
                                   index
                                 (1+ index)))
              (current-items (collection-items pane)))
          (setf (collection-items pane)
                (concatenate 'simple-vector
                             (subseq current-items 0 adjusted-index)
                             (vector item)
                             (subseq current-items adjusted-index
                                     item-count))))))
```

17.3.3 Dropping text in an editor-pane

Supply the special *drop-callback* **`:default`** to implement dropping text in an **`editor-pane`**.

17.3.4 Dropping in an output-pane

Additionally within the *drop-callback* of an **`output-pane`**, you can use **`drop-object-pane-x`** and **`drop-object-pane-y`** to query the coordinates in the pane that the object is being dropped over.

17.4 Limitations of CAPI drag and drop

`:image` format currently works fully only on Cocoa and GTK+. On Microsoft Windows the **`:image`** format works only when dragging between panes in the same process.

Drag and drop is not implemented in CAPI on Motif.

Not all pane classes support drag and drop.

18 Miscellaneous functionality

This chapter discusses miscellaneous functionality available for use during development and in your CAPI application.

18.1 Development functions

The following functions are intended as aids during development. In general they are not suitable for use in real applications, though they are fully supported.

The function **contain** takes an element argument and displays it. The element can be any pane, menu or a part of a menu, or a pinboard-object. Since displaying always requires an interface, **contain** creates an interface (unless the element is an **interface** itself). **contain** takes various keyword arguments that tell it how to display, and can also display the element as a dialog.

To create the interface, **contain** uses **make-container**, which can also be called directly.

18.2 Sounds

18.2.1 Sound API

This section applies to Cocoa and Microsoft Windows only.

On Cocoa and Microsoft Windows, CAPI provides a simple interface to play sound from sound files. The host system determines which formats of sound files it can play.

Use **load-sound** to create a sound object from either a file or the result of **read-sound-file**, then **play-sound** to play it, and **stop-sound** to stop playing. **free-sound** can be used to free it.

read-sound-file can be used to load a sound file as data into the Lisp image, which then can be used by **load-sound** without accessing a file. This is useful in delivered applications.

18.2.2 Beep

The function **beep-pane** tries to make a beep sound.

18.3 Modifier keys state

You can query the state of the modifier keys (**Control**, **Shift**, **Meta**, **Command** (**Hyper**) and **Caps Lock**) by calling **pane-modifiers-state**.

18.4 Restoring display while debugging

Some error handlers may disable display of a pane if there is an error during the display. You can check if a pane is in this state by calling **pane-can-restore-display-p**, and if so you can use **pane-restore-display** to restore the display. That assumes that the code was fixed, so is useful only while debugging.

The Window Browser tool in the LispWorks IDE allows you to restore the display interactively using these functions.

18.5 Object properties and name

All CAPI elements (panes and **`pinboard-object`**) inherit from **`capi-object`**. This includes a plist, which can be accessed by **`capi-object-property`**, **`(setf capi-object-property)`** and **`remove-capi-object-property`**. There is also the accessor **`capi-object-plist`**.

CAPI object property is a very convenient mechanism to add slot-like behavior without having to define your own class. For example, it is used for caching the images in:

```
(example-edit-file "capi/choice/list-panel-drag-images")
```

A **`capi-object`** also has a *name*, which can be used to give it a unique identifier. You can set *name* by the initarg **`:name`**, and access it by **`capi-object-name`**.

18.6 Clipboard

You can access the system clipboard, which allows passing and receiving values from other processes, by the functions **`clipboard`** and **`set-clipboard`**. These can deal with strings and images, and metafiles on Cocoa and Microsoft Windows. When used inside the same Lisp process, they can also be used to pass Lisp values. Use **`clipboard-empty`** to check if there is anything in the clipboard. See also **7.6 Edit actions on the active element**.

Similarly, the primary selection of the GUI system can accessed by the function **`selection`**, **`set-selection`** and **`selection-empty`**.

18.7 Handles

The function **`simple-pane-handle`** can be used to retrieve the "handle" of a displayed pane. Similarly **`current-dialog-handle`** returns the handle of the current dialog, if there is one.

The handle is the representation in the underlying GUI system, and may be useful in some situations for performing operations for which there is no CAPI interface.

18.8 Setting the font and colors for specific panes in specific interfaces.

The functions **`set-interface-pane-name-appearance`** and **`set-interface-pane-type-appearance`** can be used to tell LispWorks to set some attributes (font, foreground, background) in specific panes (specified by name or type) inside specific interfaces (specified by type). They can be used to customize the appearance of the panes without changing the code that created them. For example, it can be used to customize the LispWorks IDE.

19 Host Window System-specific issues

This chapter describes how the host window system affects the appearance and behavior of CAPI windows, and how to configure this.

19.1 Microsoft Windows-specific issues

19.1.1 Using Windows themes

On Microsoft Windows Vista, Windows 7, Windows 8 and Windows 10 LispWorks is *themed*. That is, it uses the current theme of the desktop.

It is possible to switch this off by calling the function **`win32:set-application-themed`** with argument **`nil`**.

`win32:set-application-themed` affects only windows that are created after it was called. Normally, it should be called before any window is created, so that all LispWorks windows will have a consistent appearance.

19.1.2 The break gesture

If a CAPI/Windows window is busy and unresponsive you can use the break gesture **`Ctrl+Break`** to regain control.

19.2 Cocoa-specific issues

19.2.1 The break gesture

If a CAPI/Cocoa window is busy and unresponsive you can use the break gesture **`Command+Ctrl+,`** (comma) to regain control.

19.2.2 The Cocoa application interface

You can use **`set-application-interface`** on an instance of a subclass of **`cocoa-default-application-interface`** to get the following functionality:

- Define the application menu (leftmost menu in the menu bar).
- Define the menu bar items that are displayed when no interface is on the screen.
- Define the Dock context menu, which is raised from the Dock icon.
- Control and callbacks about the lifecycle of the interface.

A proper Cocoa application is likely to use this mechanism. Note that the call to **`set-application-interface`** needs to happen before any display or attempt to access the screen. See **`cocoa-default-application-interface`** for more details.

19.3 GTK+-specific issues

19.3.1 The break gesture

If a CAPI/GTK+ window is busy and unresponsive you can use the break gesture **Meta+Ctrl+C** to regain control.

On GTK+ you can use the function **set-interactive-break-gestures** both to find and to set the keys that are used interactively as break gestures. When the system detects a break gesture it tries to interrupt any running process, to allow the user to deal with runaway processes.

19.3.2 Matching resources for GTK+

You can configure the LispWorks IDE and your application to use resources on GTK+. The applicable resources determine the default fonts, colors and certain other properties used in CAPI elements.

The **element** initarg **:widget-name** is used to match resources. CAPI gives a name for the main widget that it creates for each element that has a representation in the library. This name is then included in the "path" that GTK+ uses to match resources for each widget.

19.3.2.1 Resources on GTK+

By default, the name of the widget is the name of the class of the element, downcased (except top level interfaces, see next paragraph). You can override the name by either passing *widget-name* when making the element, or by setting the **element-widget-name** before displaying the element.

To make it easier to define resources specific to the application, the CAPI GTK+ library, when using the default name, prepends the *application-class* (see **convert-to-screen**) followed by a dot. So for an interface of class **my-interface** which is displayed in a screen with *application-class* **"my-application"**, the default *widget-name* is:

```
my-application.my-interface
```

Example GTK+ resource files are in your LispWorks installation directory under **examples/gtk/**:

gtkrc-break-gestures

gtkrc-font

gtkrc-parameters

gtkrc-styles

19.3.2.2 Resources for CAPI/GTK+ applications

Delivered applications which need fallback resources should pass the **:application-class** and **:fallback-resources** keys described in the manual page for **convert-to-screen**.

This example shows how to make a CAPI GUI configurable by GTK+ resources:

```
(example-edit-file "capi/elements/gtk-resources")
```

To construct custom resources for your CAPI/GTK+ application, see the example resource files in your LispWorks installation directory under **examples/gtk/**.

19.3.2.3 X resources for in-place completion windows

The special window described in **10.6 In-place completion** has interface with name **`"non-focus-list-prompter"`**. This name can be used to define resources specific to the in-place completion window. The completion list is a **`list-panel`** and the filter is a **`text-input-pane`**.

19.4 Motif-specific issues

19.4.1 Using Motif

The Motif backend is deprecated and the GTK+ backend is preferred.

This section describes how to use the Motif window system on supported platforms.

19.4.1.1 Using Motif on Linux, FreeBSD and x86/x64 Solaris

Use of Motif with LispWorks is deprecated on these platforms, but you can still use it.

LispWorks uses GTK+ as the default window system for CAPI and the LispWorks IDE on Linux, FreeBSD and x86/x64 Solaris.

To use Motif instead you need to load it explicitly, by:

```
(require "capi-motif")
```

Requiring the **`"capi-motif"`** module makes CAPI use Motif as its default library.

You can override the default library by specifying the appropriate CAPI screen (see **19.5 CAPI communication with host window system - libraries** and the *screen* argument to **`display`** and **`convert-to-screen`**).

19.4.1.2 Using Motif on Macintosh

Use of Motif with LispWorks is deprecated on the Macintosh, but you can still use it.

LispWorks is supplied as two images. One uses Cocoa as the default window system for CAPI and the LispWorks IDE, the other uses GTK+ as its default window system. Only this latter image can use the alternative Motif window system.

To use Motif you need to load it into the GTK+ LispWorks image, by:

```
(require "capi-motif")
```

Requiring the **`"capi-motif"`** module makes CAPI use Motif as its default library.

You can override the default library by specifying the appropriate CAPI screen (see **19.5 CAPI communication with host window system - libraries** and the *screen* argument to **`display`** and **`convert-to-screen`**).

Note: you cannot load Motif into the Cocoa image.

Note: the GTK+ LispWorks image is installed on Macintosh when you select the X11 GUI option at install time. See the *Release Notes and Installation Guide* for further information on installing this option.

19.4.2 The break gesture

If a CAPI/Motif window is busy and unresponsive you can use the break gesture **Meta+Ctrl+C** to regain control.

On Motif you can use the function **set-interactive-break-gestures** both to find and to set the keys that are used interactively as break gestures. When the system detects a break gesture it tries to interrupt any running process, to allow the user to deal with runaway processes.

19.4.3 Matching resources for X11/Motif

On Motif, you can configure the LispWorks IDE and your application to use resources similarly to GTK+ (see **19.3.2 Matching resources for GTK+**).

19.4.3.1 Resources on X11/Motif

widget-name is used as described for GTK+ in **19.3.2.1 Resources on GTK+**, except that the default *widget-name* for a top level interface does include the prepended *application-class*.

The file **app-defaults/Lispworks**, supplied in the LispWorks library for relevant platforms, contains the application fallback resources for LispWorks 8.0 and illustrates resources you may wish to change.

The file **app-defaults/GcMonitor** contains the application fallback resources for the Lisp Monitor window.

The files **app-defaults/*-classic** contain the fallback resources that were supplied with LispWorks 4.4.

For further information about X resources, consult documentation for the X Window system.

19.4.3.2 Resources for CAPI/Motif applications

To construct custom X resources for your CAPI/Motif application, consult **app-defaults/Lispworks** which illustrates resources you may wish to change in your application.

19.5 CAPI communication with host window system - libraries

CAPI communicates with the host window system via backends called *libraries*. In most cases you need not worry about the library, and just use generic CAPI.

Currently there are four libraries, named by keywords as follows:

:win32	The only library for Microsoft Windows.
:cocoa	The default library for macOS.
:gtk	The default library for Linux, FreeBSD and x86/x64 Solaris, also available on macOS.
:motif	Deprecated but available on non-Windows platforms.

The function **default-library** returns the default library for the current platform.

Note: On platforms that support GTK+ and Motif, **default-library** normally returns **:gtk**, but after loading Motif using **(require "capi-motif")** it returns **:motif**.

A library name is a valid argument to **convert-to-screen**, and can be used in places when a screen specification is required, most importantly as argument to **display**. Normally, however, you will be using the default screen of the default library, so you will not have to worry about it.

`default-library` is used when a program that is designed to run on various platforms wants to do different things in different GUI systems. Note that **`default-library`** is available before displaying anything, and can be used at load-time.

The functions **`installed-libraries`** returns a list of the installed libraries in the current image. Normally it is just a list of the default library, but loading Motif adds it into the list.

20 Self-contained examples

This chapter enumerates the set of CAPI examples in the LispWorks library. Each example contains complete, self-contained code and detailed comments, which include one or more entry points near the start of the file which you can run to start the program.

To run the example code:

1. Open the file in the Editor tool in the LispWorks IDE. Evaluating the call to **`example-edit-file`** shown below will achieve this.
2. Compile the example code, by **`Ctrl+Shift+B`**.
3. Place the cursor at the end of the entry point form and press **`Ctrl+X Ctrl+E`** to run it.
4. Read the comment at the top of the file, which may contain further instructions on how to interact with the example.

20.1 Output pane examples

This section lists the example files illustrating input, drawing, scrolling, tooltips, dragging and images in an **`output-pane`**. These are also applicable to **`static-layout`** and **`pinboard-layout`**.

Processing input with the *input-model*:

```
(example-edit-file "capi/output-panes/input-model1")
```

```
(example-edit-file "capi/output-panes/input-model")
```

```
(example-edit-file "capi/output-panes/drawing")
```

```
(example-edit-file "capi/output-panes/spirograph")
```

```
(example-edit-file "capi/output-panes/input-model-touch")
```

```
(example-edit-file "capi/output-panes/modifier-change")
```

Defining a command (that is, an alias to an input gesture):

```
(example-edit-file "capi/output-panes/commands")
```

Drawing to an output pane:

See the following section **20.2 Graphics examples**.

Temporary drawing on top of the normal drawing, for example when the user drags:

```
(example-edit-file "capi/output-panes/cached-display")
```

```
(example-edit-file "capi/graphics/pinboard-test")
```

```
(example-edit-file "capi/graphics/pixmap-port")
```

Simple scrolling without a scroll bar:

```
(example-edit-file "capi/output-panes/scrolling-without-bar")
```

Using *scroll-callback*:

```
(example-edit-file "capi/graphics/scrolling-test")
```

Using fixed *coordinate-origin* scrolling:

```
(example-edit-file "capi/output-panes/coordinate-origin-fixed")
```

```
(example-edit-file "capi/output-panes/fixed-origin-scrolling")
```

Displaying tooltips:

```
(example-edit-file "capi/graphics/pinboard-help")
```

Dragging from/to an output pane:

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

Copying and pasting images in an output pane:

```
(example-edit-file "capi/output-panes/drawing")
```

Indicate selection of objects in response to mouse movement:

```
(example-edit-file "capi/graphics/highlight-rectangle")
```

20.2 Graphics examples

This section lists the example files illustrating graphics transforms, transparency in images and pixmaps ports, combining existing and new pixels when drawing, drawings dependent on dynamic computations, editing an image, scaling an image, metafiles and paths.

Drawing an image read from a file:

```
(example-edit-file "capi/graphics/images")
```

Transforms and **`apply-rotation-around-point`**:

```
(example-edit-file "capi/graphics/rotation-around-point")
```

```
(example-edit-file "capi/output-panes/cached-display")
```

Creating transparent and semi-transparent areas in a pixmap:

```
(example-edit-file "capi/graphics/compositing-mode-simple")
```

Simple example of *compositing-mode*:

```
(example-edit-file "capi/graphics/compositing-mode-simple")
```

Complex example of *compositing-mode*:

```
(example-edit-file "capi/graphics/compositing-mode")
```

Simple example of scaling an image:

```
(example-edit-file "capi/graphics/image-scaling")
```

Draw something that is computed dynamically and slowly without hanging the GUI:

```
(example-edit-file "capi/graphics/plot-offline")
```

Using an Image Access object:

```
(example-edit-file "capi/graphics/image-access")
```

Pixel-by-pixel editing of an image:

```
(example-edit-file "capi/graphics/image-access-alpha")
```

Obtaining BGRA color data from an image:

```
(example-edit-file "capi/graphics/image-access-bgra")
```

Handling the alpha channel (transparency) of images:

```
(example-edit-file "capi/graphics/images-with-alpha")
```

Creating and using a metafile:

```
(example-edit-file "capi/graphics/metafile-rotation")
```

Clipboard access with a metafile:

```
(example-edit-file "capi/graphics/metafile")
```

Drawing paths using **`draw-path`**:

```
(example-edit-file "capi/graphics/paths")
```

Drawing a chart of prices:

```
(example-edit-file "capi/applications/price-charting")
```

Effects of *drawing-mode*:

```
(example-edit-file "capi/graphics/catherine-wheel")
```

20.3 Pinboard examples

Simple manipulation of **pinboard-object**s:

```
(example-edit-file "capi/graphics/pinboard-movement")
```

```
(example-edit-file "capi/graphics/pinboard-test")
```

```
(example-edit-file "capi/layouts/wrapping-layout")
```

Simple manipulation with animation:

```
(example-edit-file "capi/applications/balloons")
```

Laying out objects inside **pinboard-layout** using child layouts:

```
(example-edit-file "capi/graphics/pinboard-object-text-pane")
```

Specialized drawing using **drawn-pinboard-object**:

```
(example-edit-file "capi/graphics/pinboard-test")
```

```
(example-edit-file "capi/applications/othello")
```

Specialized drawing using your own pinboard objects:

```
(example-edit-file "capi/applications/balloons")
```

Automatic resizing of pinboard objects:

```
(example-edit-file "capi/layouts/automatic-resize")
```

Indicate selection of pinboard objects in response to mouse movement:

```
(example-edit-file "capi/graphics/highlight-rectangle-pinboard")
```

20.4 Examples using timers to implement "animation"

```
(example-edit-file "capi/graphics/rotation-around-point")
```

```
(example-edit-file "capi/graphics/metafile-rotation")
```

```
(example-edit-file "capi/applications/balloons")
```

```
(example-edit-file "capi/applications/pong")
```

20.5 Drag and Drop examples

From and to output panes:

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

From and to list panels:

```
(example-edit-file "capi/choice/drag-and-drop")
```

Images from and to list panels:

```
(example-edit-file "capi/choice/list-panel-drag-images")
```

GTK+ specific:

```
(example-edit-file "capi/elements/gtk-filename-list-and-uris")
```

Minimal drag-and-drop code:

```
(example-edit-file "capi/elements/simple-dragndrop")
```

20.6 Graph examples

Simple examples:

```
(example-edit-file "capi/graphics/graph-pane")
```

```
(example-edit-file "capi/choice/simple-graph-pane")
```

Customizing **graph-pane**:

```
(example-edit-file "capi/graphics/circled-graph-nodes")
```

```
(example-edit-file "capi/graphics/labelled-graph-edges")
```

```
(example-edit-file "capi/graphics/wiggly-line-graph")
```

```
(example-edit-file "capi/choice/simple-graph-pane")
```

Changing the appearance of edges:

```
(example-edit-file "capi/graphics/graph-color-edges")
```

20.7 Cocoa-specific examples

Control over the macOS application menu:

```
(example-edit-file "capi/applications/cocoa-application-single-window")
```

```
(example-edit-file "capi/applications/cocoa-application")
```

20.8 Examples of complete CAPI applications

Simple applications:

```
(example-edit-file "capi/applications/hangman")
```

```
(example-edit-file "capi/applications/maze")
```

```
(example-edit-file "capi/applications/maze-multi")
```

```
(example-edit-file "capi/applications/othello")
```

```
(example-edit-file "capi/applications/simple-othello")
```

```
(example-edit-file "capi/applications/pong")
```

```
(example-edit-file "capi/applications/rich-text-editor")
```

Complete interface, including toolbar, option pane, and multi-column list panel:

```
(example-edit-file "capi/applications/simple-symbol-browser")
```

Incorporating CPU-intensive work with responsive GUI:

```
(example-edit-file "capi/applications/multi-threading")
```

20.9 Choice examples

Different kinds of interaction:

```
(example-edit-file "capi/choice/double-list-panels")
```

```
(example-edit-file "capi/choice/list-panels")
```

Using *print-function* and *data-function*:

```
(example-edit-file "capi/choice/list-panels")
```

Using **(setf capi:collection-items)** and *print-function* in a list panel:

```
(example-edit-file "capi/choice/expanding-list")
```

Adding images:

```
(example-edit-file "capi/choice/double-list-panels")
```

Drag and drop in a list panel:

```
(example-edit-file "capi/choice/drag-and-drop")
```

```
(example-edit-file "capi/choice/list-panel-drag-images")
```

Simple **tree-view** with images:

```
(example-edit-file "capi/choice/tree-view")
```

```
(example-edit-file "capi/choice/extended-selection-tree-view")
```

Tree-view images and checkboxes:

```
(example-edit-file "capi/choice/extended-selection-tree-view")
```

tree-view combined with an XML parser to display an RSS file:

```
(example-edit-file "capi/applications/rss-reader")
```

An example of using **stacked-tree**:

```
(example-edit-file "capi/choice/stacked-tree")
```

Interaction between context menu and selection:

```
(example-edit-file "capi/choice/list-panel-pane-menu")
```

Multi column list panel:

```
(example-edit-file "capi/choice/multi-column-list-panels")
```

Sorting a **list-panel** for a specific column:

```
(example-edit-file "capi/choice/multi-column-list-panels")
```

Using *keyboard-search-callback* in a **list-panel**:

```
(example-edit-file "capi/choice/list-panel-keyboard-search")
```

Adding images to **option-pane**:

```
(example-edit-file "capi/choice/option-pane-with-images")
```

Disabling items in **option-pane**:

```
(example-edit-file "capi/choice/option-pane-with-images")
```

```
(example-edit-file "capi/choice/option-pane")
```

Alternative action callback (that is, a callback when modifier key is pressed):

```
(example-edit-file "capi/choice/alternative-action-callback")
```

20.10 Examples of dialogs and prompts

Simple dialog:

```
(example-edit-file "capi/dialogs/simple-dialog")
```

```
(example-edit-file "capi/dialogs/mutating-dialog")
```

Customizing **prompt-with-list**:

```
(example-edit-file "capi/choice/prompt-with-buttons")
```

20.11 editor-pane examples

Simple editor pane:

```
(example-edit-file "capi/editor/editor-pane")
```

change-callback, text property and editor face:

```
(example-edit-file "capi/editor/change-callback")
```

Callbacks before and after input:

```
(example-edit-file "capi/editor/input-callback")
```

20.12 Menu examples

Adding images to menus:

```
(example-edit-file "capi/elements/menu-with-images")
```

Defining accelerator keys:

```
(example-edit-file "capi/elements/accelerators")
```

Dynamically defining the items in the context menu:

```
(example-edit-file "capi/elements/pane-popup-menu-items")
```

Button with a drop-down menu:

```
(example-edit-file "capi/elements/popup-menu-button")
```

Menus with a *popup-callback*:

```
(example-edit-file "capi/elements/popup-menu-button")
```

20.13 Miscellaneous examples

A prototype grid implementation, and an example using it:

```
(example-edit-file "capi/elements/grid")
```

```
(example-edit-file "capi/elements/grid-impl")
```

Converting coordinates between a pane and its ancestors or the screen:

```
(example-edit-file "capi/elements/convert-relative-position")
```

Changing the mouse cursor:

```
(example-edit-file "capi/elements/cursor")
```

Passing initargs to a pane inside an interface using **:make-instance-extra-apply-args**:

```
(example-edit-file "capi/applications/argument-passing")
```

Server and client for a simple line-based textual chat program:

```
(example-edit-file "capi/applications/chat")
```

```
(example-edit-file "capi/applications/chat-client")
```

Server and client for a simple textual remote debugger:

```
(example-edit-file "capi/applications/remote-debugger")
```

```
(example-edit-file "capi/applications/remote-debugger-client")
```

20.14 GTK+ specific examples

Defining and using GTK+ resources:

```
(example-edit-file "capi/elements/gtk-resources")
```

Dragging URIs:

```
(example-edit-file "capi/elements/gtk-filename-list-and-uris")
```

20.15 Motif specific examples

Defining and using Motif resources:

```
(example-edit-file "capi/elements/widget-name")
```

20.16 Layout examples

Simple **grid-layout**:

```
(example-edit-file "capi/layouts/titles-in-grid")
```

Extending cells in **grid-layout**:

```
(example-edit-file "capi/layouts/extend")
```

Dynamic resizing of layouts:

```
(example-edit-file "capi/layouts/resize-layout")
```

Define a layout which aligns its children top/bottom and also displays oversized children nicely:

```
(example-edit-file "capi/layouts/buffer-layout")
```

A **graph-pane** with a custom layout:

```
(example-edit-file "capi/graphics/simple-layout-definition")
```

20.17 Tooltip examples

General tooltips:

```
(example-edit-file "capi/elements/help")
```

Displaying tooltips in an **output-pane**:

```
(example-edit-file "capi/graphics/pinboard-help")
```

20.18 Examples illustrating other pane classes

Simple standalone scroll bar:

```
(example-edit-file "capi/elements/scroll-bar")
```

Non-linear integer values in a **slider**:

```
(example-edit-file "capi/elements/slider-print-function")
```

Simple use of progress bars:

```
(example-edit-file "capi/elements/progress-bar")
```

Updating a progress bar from another thread:

```
(example-edit-file "capi/elements/progress-bar-from-background-thread")
```

text-input-choice basic functionality:

```
(example-edit-file "capi/elements/text-input-choice")
```

`text-input-pane` basic functionality:

```
(example-edit-file "capi/elements/text-input-pane")
```

`text-input-range` basic functionality:

```
(example-edit-file "capi/elements/text-input-range")
```

Toolbar examples:

```
(example-edit-file "capi/elements/toolbar")
```

Docking layout:

```
(example-edit-file "capi/layouts/docking-layout")
```

Switchable layout:

```
(example-edit-file "capi/layouts/switchable")
```

Rich Text pane:

```
(example-edit-file "capi/applications/rich-text-editor")
```

Various buttons:

```
(example-edit-file "capi/buttons/buttons")
```

Simple layout in button panel:

```
(example-edit-file "capi/buttons/button-panel-layout")
```

`tracking-pinboard-layout` example:

```
(example-edit-file "capi/graphics/tracking-pinboard-layout")
```

`simple-network-pane` example with labeling of graph edges:

```
(example-edit-file "capi/graphics/network")
```

20.19 Printing examples

Simple printing:

```
(example-edit-file "capi/printing/simple-print-port")
```

Fitting drawing to a page:

```
(example-edit-file "capi/printing/fit-to-page")
```

Printing a drawing on multiple pages:

```
(example-edit-file "capi/printing/multi-page")
```

```
(example-edit-file "capi/printing/page-on-demand")
```

20.20 Graphic Tools examples

Using the higher level Graphic Tools to draw bar charts and graphs:

```
(example-edit-file "graphic-tools/bar-chart-example")
```

```
(example-edit-file "graphic-tools/graph-example")
```

Drawing a chart of prices:

```
(example-edit-file "capi/applications/price-charting-gt")
```

21 CAPI Reference Entries

The following chapter documents symbols exported from the **capi** package.

abort-callback *Function*

Summary

Aborts out of the context of the current callback.

Package

capi

Signature

abort-callback &optional *always-abort*

Arguments

always-abort⇓	A generalized boolean.

Description

The function **abort-callback** aborts out of the context of the current callback, returning **nil** when it is relevant (for example in an **interface** *confirm-destroy-callback*).

If called outside the context of a callback, if *always-abort* is **t** then **abort-callback** calls **(abort)**, otherwise it just returns.

The default value of *always-abort* is **t**.

See also

callbacks
interface
3.4 Callbacks

abort-dialog *Function*

Summary

Aborts the current dialog.

Package

capi

Signature

abort-dialog &rest *ignored-args*

Arguments

ignored-args⇓ Lisp objects.

Description

The function **abort-dialog** aborts the current dialog. For example, it can be made a selection callback from a **Cancel** button so that pressing the button aborts the dialog. In a similar manner the complementary function **exit-dialog** can be used as a callback for an **OK** button.

The arguments in *ignored-args* are all ignored.

If there is no current dialog then **abort-dialog** does nothing and returns **nil**. If there is a current dialog then **abort-dialog** either returns non-nil or does a non-local exit. Therefore code that depends on **abort-dialog** returning must be written carefully. Constructs like this can be useful:

```
(unless (capi:abort-dialog)
  (foo))
```

Above, *foo* will be called only if there is no current dialog.

It is not useful to do either:

```
(when (capi:abort-dialog)
  (foo))
```

or:

```
(progn
  (capi:abort-dialog)
  (foo))
```

as in both cases it is not well-defined whether *foo* will be called if there is a current dialog.

Examples

```
(capi:display-dialog
  (capi:make-container
    (make-instance 'capi:push-button
                   :text "Cancel"
                   :callback 'capi:abort-dialog)
                   :title "Test Dialog"))
```

Also see these examples:

```
(example-edit-file "capi/dialogs/")
```

See also

exit-dialog
display-dialog
popup-confirmer

interface
10 Dialogs: Prompting for Input

abort-exit-confirmer *Function*

Summary

Aborts the exiting of a dialog.

Package

capi

Signature

abort-exit-confirmer

Description

The function **abort-exit-confirmer** can be used to abort the exiting of a confirmer. It can be used in the *ok-function* of a confirmer, to abort the exit and return to the dialog.

If **abort-exit-confirmer** is called outside the exiting of a confirmer, it does nothing.

Examples

This example asks the user for a string. If the string is longer than 20 characters, it confirms with the user that they really want such a long string, and if they do not it returns to the dialog.

```
(capi:popup-confirmer
 (make-instance 'capi:text-input-pane)
 "New Name"
 :value-function 'capi:text-input-pane-text
 :ok-function
 #'(lambda (value)
     (when (and (> (length value) 20)
                (not (capi:prompt-for-confirmation
                      "Name is very long. Use it?")))
       (capi:abort-exit-confirmer))
     value))
```

See also

popup-confirmer

accepts-focus-p *Generic Function*

Summary

Determines if an element accepts the focus.

Package

capi

Signature

accepts-focus-p *element* **=>** *result*

Arguments

element⇓ A CAPI element.

Values

result A boolean.

Description

The generic function **accepts-focus-p** determines if the element *element* accepts the focus for user input, and controls tabstops.

The method on **element** uses the value of the *accepts-focus-p* slot, but methods on some subclasses override this.

accepts-focus-p also influences whether a pane is a tabstop. On Microsoft Windows a pane acts as a tabstop if and only if the function **accepts-focus-p** returns true and the **element** *accepts-focus-p* initarg value is **:force**. On Motif and Cocoa, a pane acts as a tabstop if and only if the function **accepts-focus-p** returns true.

See also

element
pane-has-focus-p
set-pane-focus
3.1.5 Focus

activate-pane *Function*

Summary

Gives a pane the input focus and raises the window containing it.

Package

capi

Signature

activate-pane *pane*

Arguments

pane⇓ An **element** or a **pinboard-object** or a **toolbar-object**.

Description

The function **`activate-pane`** gives the focus to the pane *pane* and brings the window containing *pane* to the front.

If *pane* cannot accept the focus then **`activate-pane`** chooses a sensible alternative inside the same interface.

Examples

This example demonstrates how to swap the focus from one window to another.

```
(setq text-input-pane
      (capi:contain (make-instance
                     'capi:text-input-pane)))


(setq button
      (capi:contain (make-instance
                     'capi:push-button
                     :text "Press Me")))


(capi:activate-pane text-input-pane)


(capi:activate-pane button)
```

See also

`hide-interface`
`raise-interface`
`set-pane-focus`
`show-interface`
`quit-interface`
`simple-pane`
7.7 Manipulating top-level windows

active-pane-copy
active-pane-copy-p
active-pane-cut
active-pane-cut-p
active-pane-deselect-all
active-pane-deselect-all-p
active-pane-paste
active-pane-paste-p
active-pane-select-all
active-pane-select-all-p
active-pane-undo
active-pane-undo-p

Functions

Summary

Perform, or check applicability of, an "edit/select operation" on the active pane.

Package

`capi`

Signatures

`active-pane-copy &optional` *pane*

`active-pane-copy-p &optional` *pane*

`active-pane-cut &optional` *pane*

`active-pane-cut-p &optional` *pane*

`active-pane-deselect-all &optional` *pane*

`active-pane-deselect-all-p &optional` *pane*

`active-pane-paste &optional` *pane*

`active-pane-paste-p &optional` *pane*

`active-pane-select-all &optional` *pane*

`active-pane-select-all-p &optional` *pane*

`active-pane-undo &optional` *pane*

`active-pane-undo-p &optional` *pane*

Arguments

pane⇓ A **`simple-pane`**.

Description

These functions perform an "edit/select operation" on the active pane, or check if this operation is currently applicable.

The active pane will be the one on the same screen as *pane* if *pane* is non-nil, or otherwise the same screen as the default interface.

These functions find the active pane, that is the pane where keyboard input currently goes. Note that this is not necessarily a pane that is recognized by CAPI. The predicates (those with names ending **`-p`**) return true if the operation is currently applicable. The other functions tell the active pane to do the operation.

The edit/select operations are implemented by the **`pane-interface-*`** generic functions such as **`pane-interface-copy-object`**.

It is not an error to do the operation even if the predicate returns false. It will just do nothing useful.

Examples

```
(example-edit-file "capi/applications/rich-text-editor")
```

See also

pane-interface-copy-object
7.6 Edit actions on the active element

append-items *Generic Function*

Summary

Adds to the items in a collection.

Package

capi

Signature

append-items *collection new-items*

Arguments

collection⇓	A collection.
new-items⇓	A sequence.

Description

The generic function **append-items** adds the items in *new-items* to the **collection** *collection*.

This is logically equivalent to recalculating the collection items and calling **(setf collection-items)**. However, **append-items** is more efficient and causes less flickering on screen.

append-items can only be used when the **collection** has the default *items-get-function* **svref**.

Notes

append-items cannot be used a **graph-pane** or a **tree-view**.

See also

collection
remove-items
replace-items
5 Choices - panes with items

apply-in-pane-process *Function*

Summary

Applies a function in the process associated with a pane.

Package

capi

Signature

apply-in-pane-process *pane* *function* **&rest** *args* **=> nil**

Arguments

pane⇓	An **element** or a **pinboard-object** or a **toolbar-object**.
function⇓	A function designator.
args⇓	Lisp objects.

Description

The function **apply-in-pane-process** applies *function* to *args* in the process that is associated with *pane*. This is required when *function* modifies *pane* or changes how it is displayed. If *pane* has not been displayed yet, then *function* is called immediately.

Notes

1. All accesses (reads as well as writes) on a pane should be performed in the pane's process. Within a callback on the pane's interface this happens automatically, but **apply-in-pane-process** is a useful utility in other circumstances.
2. **apply-in-pane-process** calls *function* on the current process if the pane's interface does not have a process.
3. If the pane's process is no longer active then **apply-in-pane-process** applies *function* directly.
4. **apply-in-pane-process-if-alive** is another way to call *function* in the CAPI process appropriate for *pane*. However it only does this if *pane* is alive so in particular, if *pane* does not have a process, it does not call *function*.

Examples

Editor commands must be called in the correct process:

```
(setq editor
      (capi:contain
       (make-instance 'capi:editor-pane
                      :text "Once upon a time...")))


(capi:apply-in-pane-process
 editor 'capi:call-editor editor "End Of Buffer")


(capi:apply-in-pane-process
 editor 'capi:call-editor editor "Beginning Of Buffer")
```

See also

apply-in-pane-process-if-alive
execute-with-interface
4.1 The correct thread for CAPI operations
7 Programming with CAPI Windows

apply-in-pane-process-if-alive
apply-in-pane-process-wait-single
apply-in-pane-process-wait-multiple *Functions*

Summary

Applies a function in the process associated with a pane, and optionally waits for and returns its values.

Package

capi

Signatures

apply-in-pane-process-if-alive *pane function* **&rest** *args* **=>** *alivep*

apply-in-pane-process-wait-single *pane timeout function* **&rest** *args* **=>** *result*, *status*

apply-in-pane-process-wait-multiple *pane timeout function* **&rest** *args* **=>** *results*, *status*

Arguments

pane⇓	A CAPI element or pinboard object.
function⇓	A function or an fbound symbol.
args⇓	Any Lisp objects.
timeout⇓	A non-negative real (number of seconds) or **nil**.

Values

alivep⇓	A boolean.
result⇓	Any Lisp object.
status⇓	**nil**, **t** or **:timeout**.
results⇓	A list of Lisp objects.

Description

The function **apply-in-pane-process-if-alive** applies *function* to *args* in the process that is associated with *pane* , if *pane* is "alive". This is like **apply-in-pane-process** except that *function* is called only if *pane* is alive. The meaning of "alive" and the value of *alivep* are as defined for **execute-with-interface-if-alive**.

If *pane* does not have a process, then *function* is not called.

The return value of **apply-in-pane-process-if-alive**, *alivep*, is true if the pane is "alive" and false otherwise.

apply-in-pane-process-wait-single applies *function* to *args* like **apply-in-pane-process-if-alive**, and then waits for *function* to return. If the call returns successfully, *result* is the first return value of the call to *function*, and *status* is **t**. If *pane* is not "alive", *result* and *status* are **nil**. If *timeout* is non-nil and the call did not return within *timeout* seconds, then *result* is **nil** and *status* is **:timeout**.

apply-in-pane-process-wait-multiple is the same as **apply-in-pane-process-wait-single** except for the returned values. If the call to *function* returns successfully, *results* is a list of the values that *function* returned and *status* is **t**. If *pane* is not "alive", *result* and *status* are **nil**. If *timeout* is non-nil and the call did not return within *timeout* seconds, then *result* is **nil** and *status* is **:timeout**.

Notes

Even if **apply-in-pane-process-if-alive** returns true for *alivep*, *function* is not guaranteed to be called. For example, the process of *pane* might be killed or hang.

After *timeout* has expired in **apply-in-pane-process-wait-multiple** or **apply-in-pane-process-wait-single**, *function* may or may not have been called.

apply-in-pane-process-wait-multiple and **apply-in-pane-process-wait-single** work by creating a **mp:mailbox**, applying (in the same way that **apply-in-pane-process-if-alive** does) a lambda that puts the result(s) of *function* in the mailbox, and then wait for the mailbox. It is quite easy to write your own version of this if you need additional features (for example, error handling).

See also

apply-in-pane-process
execute-with-interface-if-alive
4.1 The correct thread for CAPI operations
7 Programming with CAPI Windows

arrow-pinboard-object *Class*

Summary

A **pinboard-object** that draws itself as an arrow.

Package

capi

Superclasses

line-pinboard-object

Subclasses

double-headed-arrow-pinboard-object
labelled-arrow-pinboard-object

Initargs

:head A keyword specifying the position of the arrowhead on the line.

:head-direction A keyword specifying the direction of the arrowhead.

:head-length The length of the arrowhead.

:head-breadth The breadth of the arrowhead, or **nil**.

:head-graphics-args

A graphics args plist.

Description

An instance of the class **`arrow-pinboard-object`** is a **`pinboard-object`** that draws itself as an arrow.

head must be **`:end`**, **`:middle`** or **`:start`**. The default is **`:end`**.

head-direction must be **`:forwards`**, **`:backwards`** or **`:both`**. The default is **`:forwards`**.

head-length is the length of the arrowhead in pixels. It defaults to 12.

head-breadth is the breadth of the arrowhead in pixels, or **`nil`** which means that the breadth is half of *head-length*. The default is **`nil`**.

head-graphics-args is a plist of graphics state parameters and values used when drawing the arrow head. For information about the graphics state, see **`graphics-state`**.

Examples

```
(capi:contain
 (make-instance
  'capi:pinboard-layout
  :description
  (list
   (make-instance 'capi:arrow-pinboard-object
                  :start-x 5 :start-y 10
                  :end-x 105 :end-y 60 )
   (make-instance 'capi:arrow-pinboard-object
                  :start-x 5 :start-y 110
                  :end-x 105 :end-y 160
                  :head :middle)
   (make-instance 'capi:arrow-pinboard-object
                  :start-x 5 :start-y 210
                  :end-x 105 :end-y 260
                  :head-direction :both )
   (make-instance 'capi:arrow-pinboard-object
                  :start-x 5 :start-y 310
                  :end-x 105 :end-y 360
                  :head-graphics-args
                  '(:foreground :pink)
                  :head-length 30)
   (make-instance 'capi:arrow-pinboard-object
                  :start-x 5 :start-y 410
                  :end-x 105 :end-y 460
                  :head-length 30 :head-breadth 5)
   (make-instance 'capi:arrow-pinboard-object
                  :start-x 5 :start-y 510
                  :end-x 105 :end-y 560
                  :head-breadth 10
                  :head-direction :backwards))
  :visible-min-width 120
  :visible-min-height 620))
```

See also

`graphics-state`
12.3 Creating graphical objects

attach-interface-for-callback *Function*

Summary

Changes the interface that is passed when a callback is made.

Package

capi

Signature

attach-interface-for-callback *element interface*

Arguments

element⇓ An **element**.

interface⇓ An **interface**.

Description

The function **attach-interface-for-callback** changes the interface that is passed when a callback is made. Callbacks for *element* get passed *interface* instead of the parent interface of *element*.

See also

callbacks
element
element-interface-for-callback
interface
3.4 Callbacks

attach-simple-sink *Function*

Summary

Attaches a sink to the active component in an **ole-control-pane**.

Package

capi

Signature

attach-simple-sink *invoke-callback pane interface-name* **&key** *sink-class* **=>** *sink*

Arguments

invoke-callback⇓ A function designator.

pane⇓ An **ole-control-pane**.

interface-name⇓ A refguid or the symbol **:default**.

sink-class⇓ A symbol naming a class.

Values

sink The sink object.

Description

The function **attach-simple-sink** make a sink object and attaches it to the active component in *pane*.

When an event callback is triggered for the source interface named by *interface-name*, the sink object will call *invoke-callback* with four arguments: *pane* (see *sink-class* below), the source method name as a string, the source method type (either **:method**, **:get** or **:put**) and a vector of the remaining callback arguments.

interface-name is either a string naming a source interface that the component in *pane* supports or **:default** to connect to the default source interface.

sink-class can be used to control the class of the sink object. This defaults to **ole-control-pane-simple-sink**, but can be a subclass of this class to allow the first argument of *invoke-callback* to be chosen by a method on the generic function **com:simple-i-dispatch-callback-object**.

Attached sinks are automatically disconnected when the object is closed or can be manually disconnected by calling **detach-simple-sink**.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

detach-simple-sink
ole-control-pane
ole-control-pane-simple-sink

attach-sink *Function*

Summary

Attaches a sink to the active component in an **ole-control-pane**.

Package

capi

Signature

attach-sink *sink pane interface-name*

Arguments

sink⇓ A class instance.

pane⇓ An **ole-control-pane**.

interface-name⇓ A refguid or the symbol **:default**.

Description

The function **attach-sink** attaches a sink to the active component in the the **ole-control-pane** *pane*.

sink is an instance of a class that implements the source interface *interface-name*.

pane is an **ole-control-pane** which is the pane where the component is.

interface-name is either a string naming a source interface that the component in *pane* supports or **:default** to connect to the default source interface.

Attached sinks are automatically disconnected when the object is closed or can be manually disconnected by calling **detach-sink**.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

attach-simple-sink
detach-sink
ole-control-pane

beep-pane *Function*

Summary

Sounds a beep.

Package

capi

Signature

beep-pane &optional *pane*

Arguments

pane⇓ A **simple-pane**.

Description

The function **beep-pane** sounds a beep on the screen associated with *pane* or on the current screen if *pane* is **nil**.

Examples

```
(capi:beep-pane)
```

See also

simple-pane
screen
18.2 Sounds

browser-pane *Class*

Summary

Embeds a pane that can display HTML. Implemented only on Microsoft Windows and Cocoa.

Package

capi

Superclasses

simple-pane

Initargs

:before-navigate-callback
A function that is called before navigating, or **nil**.

:navigate-complete-callback
A function that is called when navigation completes, or **nil**.

:new-window-callback
A function that is called before opening a new window, or **nil**.

:status-text-change-callback
A function that is called when there is a new status text or **nil**.

:document-complete-callback
A function that is called when a document is complete, or **nil**.

:title-change-callback
A function that is called when the title changes, or **nil**.

:update-commands-callback
A function that is called when the enabled status of commands related to the pane may need to change, or **nil**.

:internet-explorer-callback
Microsoft Windows specific: A function that is whenever there is an event from the underlying IWebBrowser2, or **nil**.

:navigate-error-callback
A function that is called when the pane fails to navigate, or **nil**.

:progress-callback

:debug A boolean specifying whether debugging mode is on or not.

:url A string specifying the initial URL.

Accessors

browser-pane-navigate-complete-callback
browser-pane-new-window-callback
browser-pane-status-text-change-callback
browser-pane-document-complete-callback
browser-pane-title-change-callback
browser-pane-update-commands-callback
browser-pane-internet-explorer-callback
browser-pane-before-navigate-callback
browser-pane-navigate-error-callback
browser-pane-debug

Readers

browser-pane-url
browser-pane-successful-p
browser-pane-title

Description

An instance of the class **browser-pane** is a pane that embeds a pane that can display HTML. Navigation in the pane happens either by the user clicking on hyperlinks, or by the application using **browser-pane-navigate**. The various callbacks gives the program information on what happens in the window and can be used to control (for example, to block or redirect pages).

browser-pane is implemented only on Microsoft Windows (where it embeds an IWebBrowser2) and Cocoa (where it uses WebKit).

The initarg **:url** specifies the initial URL. After being created, the pane automatically navigates to this URL.

When *before-navigate-callback* is non-nil, it is called before any navigation (whether programmatic or by the user), and gives the application control over whether to perform the navigation. The callback must have this signature:

before-navigate-callback *pane url* **&key** *sub-frame-p frame-name* **&allow-other-keys =>** *do-it*

pane is the pane that navigates, and *url* is a string to which it wants to navigate. *sub-frame-p* is true when the navigation is for a sub-frame inside the current URL, otherwise *sub-frame-p* is **nil**. *frame-name* is either **nil** or the name of a sub-frame when the navigation is to a sub-frame.

If *before-navigate-callback* returns **nil**, the navigation is canceled.

Note: To perform a redirection, just call **browser-pane-navigate** to the required URL, and return **nil** from *before-navigate-callback*.

If *new-window-callback* is non-nil, it is called before the pane tries to open a new window. It must have this signature:

new-window-callback *pane url* **&key** *context flags* **&allow-other-keys =>** *do-it-p*

pane is the pane that wants to open a new window, and *url* is a string containing the URL that the new window will navigate to. *context* is a string containing the URL of the page from which the request comes.

flags is implementation-specific flags. On Cocoa *flags* is always 0. On Microsoft Windows *flags* contains bits from the NWMF enumeration.

If *new-window-callback* returns **nil**, the opening of the new window is canceled. If *new-window-callback* returns **t** or is not supplied, it launches a browser using the OS settings.

On Microsoft Windows, *new-window-callback* is invoked from the "NewWindow3" event (or "NewWindow2" for old versions) of the sink of the underlying IWebBrowser2. If not canceled, the pane opens a new normal Internet Explorer window.

If *document-complete-callback* is non-nil, it is called when the new document in the pane is complete. It must be a function with signature:

`document-complete-callback` *pane url title* **`=>`**

url is the loaded URL, and may be **`nil`** in the case of failure. *title* is a string that is associated with the URL *url* (or the previous URL if the latest call failed).

document-complete-callback is called when, as far as the system is concerned, all the data for the URL has been loaded and is displayed in the pane. There is only one call to *document-complete-callback* for each navigation of the pane.

If *navigate-complete-callback* is non-nil, it is called whenever a navigation completes. *navigate-complete-callback* can be called several times for each navigation of the pane. It must be a function with the signature:

`navigate-complete-callback` *pane url sub-frame-p* **`=>`**

pane is the pane that is navigated. *url* is a string to which it navigated, unless the navigation failed, in which case *url* is **`nil`**. *sub-frame-p* is true when the navigation was in a sub-frame.

Notes: For most purposes the *document-complete-callback* is more useful than *navigate-complete-callback*. When *navigate-complete-callback* gets a **`nil`** *url*, the value of the URL in the pane (that is, what the accessor **`browser-pane-url`** returns) is still set to the actual URL. The success flag (which you can read with **`browser-pane-successful-p`**) is set to **`nil`**.

url can be non-nil even if there was an error in the navigation, if the server supplied another URL. In this case, on Microsoft Windows only, the success flag is set to **`:redirected`**. You can read it with **`browser-pane-successful-p`**.

If *navigate-error-callback* is non-nil, it is called when navigation fails for some reason. It should have this signature:

`navigate-error-callback` *pane url* **`&key`** *http-code error-symbol implementation-error-code message frame-name sub-frame-p fatal* **`&allow-other-keys =>`** *cancel*

pane is the navigating pane, and *url* is the URL that got the error.

If the failure is server-side failure, then *http-code* contains the http-code in the response of the server, otherwise (that is, when it failed to connect to a server) it is **`nil`**.

error-symbol is a keyword uniquely identifying the error. For an http error it is of the form **`:HTTP_STATUS*`**, and for requests with bad syntax *error-symbol* is **`:bad-request`**.

On Microsoft Windows *implementation-error-code* is the code in the "NavigateError" event. If *http-code* is non-nil then *implementation-error-code* and *http-code* will be the same. On Cocoa *implementation-error-code* will be the same as *http-code* in the case of server-side failure, otherwise it is one of the **`NSURLError*`** constants.

fatal is a boolean. A true value means that nothing is going to be displayed in the pane to tell the user about the error.

message is a message saying what the error is. *sub-frame-p* is **`t`** when the navigation is for a sub-frame, otherwise **`nil`**. *frame-name* is the name of the frame.

The return value *cancel* of *navigate-error-callback* should be one of **`nil`**, **`t`**, or **`:stop`**, with these interpretations:

`nil` On Microsoft Windows this means displaying either the substitution page from the server if there is one, or displaying automatically generated (by the underlying IWebBrowser2) error page.

t Cancel. On Microsoft Windows this means not displaying the automatically generated error page, but displaying server substitution if there is any.

:stop Stop the navigation immediately.

Note that the effect of the returned value *cancel* is only on the specific navigation, so it possible for a sub-frame to be stopped, while the main page and maybe other sub-frames complete.

On Cocoa there is no automatically generated error page, so the return value of *cancel* **nil** means the same as **t**, and both display whatever the server returned.

Note: To redirect on error, *navigate-error-callback* should just call **browser-pane-navigate** with the new page and return **:stop**.

If *title-change-callback* is non-nil, it is called when the title of the pane should change. It should have this signature:

title-change-callback *pane new-title*

new-title is a string, which the application should use as the title of the pane.

Note: In most cases, using the *title* argument of the *document-complete-callback* is more useful.

If *status-text-change-callback* is non-nil, it is called when the status text of the pane should change. It has this signature:

status-text-change-callback *pane new-status-text*

new-status-text is a string, which the application should use as the status text for the pane.

If *update-commands-callback* is non-nil, it is called when other panes (typically buttons or menu items) that are used to perform commands on the pane need to update. The callback has this signature:

update-commands-callback *pane what enabled-p*

Currently *what* can be one of:

:forward Other panes that are used to go forward in the pane should be enabled or disabled.

:backward Other panes that are used to go backward in the pane should be enabled or disabled.

Additionally on Microsoft Windows only, *what* can be:

t Other panes that may try to anything with the pane may need updating. Note that this callback is called quite often with *what* = **t**, so make sure it usually does not do much work in this case.

enabled-p specifies whether the other panes should be enabled or disabled.

On Windows only, if *internet-explorer-callback* is non-nil, it is called for each event for the pane. It has the signature:

internet-explorer-callback *pane event-name args*

event-name is a string specifying the event. *args* is a vector containing the arguments in order. The callback is called before any code that is used to implement the callbacks, which is called afterwards with the same argument vector. That means that the callback should not set anything in the vector, except when debugging.

internet-explorer-callback is intended to add functionality that is not given by the callbacks, and for debugging (but see also **:debug**). If you need more control, you probably want to define your pane directly: for the basics see:

```
(example-edit-file "com/ole/html-viewer")
```

debug specifies that the pane should be in debugging mode. Currently, on Microsoft Windows this means that it prints each event and the arguments that it receives. Whenever an event is sent to the sink associated with the embedded browser, the method name (which is the same as the event name in this case) and the argument are printed to **`mp:*background-standard-output*`**. On Cocoa it prints some diagnostics to **`mp:*background-standard-output*`**.

`browse-pane-url` returns the current *url* of the pane. Initially the value is the keyword **`:url`**, but once the browser completed navigation to some URL it is changed to this. Note that the *url* changes even if the navigation was not successful, as long as it was not stopped or canceled and there was no substitution page.

`browse-pane-title` returns the title of the current document. Note that during navigation **`browse-pane-title`** and **`browse-pane-url`** may not be synchronized. They are synchronized when *document-complete-callback* is called, until the next *before-navigate-callback* call.

`browser-pane-successful-p` tests whether the navigation to the current URL completed successfully, returning **`nil`** for failure and **`t`** for success. On Microsoft Windows only it can also return **`:substituted`**, which means that the server returned an error but also supplied a substitution page. On Cocoa, **`browser-pane-successful-p`** returns only **`t`** or **`nil`**.

Notes

`browser-pane` and related APIs are implemented on Microsoft Windows and Cocoa only. You can test whether it is available by **`browser-pane-available-p`**.

See also

`browser-pane-available-p`
`browser-pane-busy`
`browser-pane-go-forward`
`browser-pane-go-back`
`browser-pane-navigate`
`browser-pane-refresh`
`browser-pane-set-content`
`browser-pane-stop`
3.6 Displaying rich text

browser-pane-available-p *Function*

Summary

The predicate for whether **`browser-pane`** can be used on a specified screen.

Package

`capi`

Signature

`browser-pane-available-p &optional` *screen-spec* **`=>`** *result*

Arguments

screen-spec⇓ A CAPI object, a plist, or **`nil`**,.

Values

result A boolean.

Description

The function **`browser-pane-available-p`** returns true if there is a **`browser-pane`** implementation for the library associated with *screen-spec*.

If *screen-spec* is not supplied, the default library is used.

If *screen-spec* is supplied, it must be a valid argument to **`convert-to-screen`**.

See also

`browser-pane`
`convert-to-screen`

browser-pane-navigate
browser-pane-busy
browser-pane-go-back
browser-pane-go-forward
browser-pane-set-content
browser-pane-stop
browser-pane-refresh

Functions

Summary

Controls a **`browser-pane`**.

Package

`capi`

Signatures

`browser-pane-navigate` *pane url* **`=>`** *result*

`browser-pane-busy` *pane* **`=>`** *result*

`browser-pane-go-back` *pane*

`browser-pane-go-forward` *pane*

`browser-pane-set-content` *pane string*

`browser-pane-stop` *pane*

`browser-pane-refresh` *pane* **`&optional`** *level*

Arguments

pane⇓ A **`browser-pane`**.

url⇓ A string.

string⇓ A string.

level⇓ One of the keywords **:normal** and **:completely**.

Values

result⇓ A boolean.

Description

These functions are used to control an instance of **browser-pane**.

browser-pane-navigate navigates to *url*, that is it gets and displays the contents of *url*. Note that if there is any redirection, it is the redirected URL that is displayed.

browser-pane-navigate does the navigation asynchronously, so when the function returns the navigation has just started. If *result* is true then the navigation started, and if *result* is **nil** then some error in the URL has already been detected. If the pane has an error callback, it already has been called in this case.

If **browser-pane-navigate** is called while *pane* is not displayed, it sets the initial URL of it.

Note: **browser-pane-navigate** can be used to effect a redirection from inside the error before navigation and new-window callbacks.

browser-pane-busy tests whether the browser is currently navigating, returning true if it is.

browser-pane-go-forward and **browser-pane-go-back** navigate forward and back in the history, like the buttons on most web browsers.

browser-pane-set-content sets the contents of *pane* to *string*. It has same effect as if *pane* navigated to a URL whose contents is *string*. **browser-pane-set-content** creates a temporary file containing *string* and uses the pathname as the URL for *pane*. The file is deleted when *pane* is destroyed.

browser-pane-stop stops the current navigation.

browser-pane-refresh refreshes the pane, which means re-reading the URL. *level* can be one of:

:normal Asks the server for the contents again. This is the default value of *level*.

:completely Asks the server for the contents again without looking at any cache.

Notes

browser-pane and related APIs are implemented on Microsoft Windows and Cocoa only.

Compatibility note

In LispWorks 6.1 these functions were documented as generic functions, however it is not intended that you should define methods.

See also

browser-pane

browser-pane-property-get
browser-pane-property-put

Generic Functions

Summary

Get or set value of a specified Windows property of the underlying browser.

Package

`capi`

Signatures

`browser-pane-property-get` *pane property-name* **`=>`** *value*

`browser-pane-property-put` *pane property-name value*

Arguments

pane⇓	A **`browser-pane`**.
property-name⇓	A string.
value⇓	A Lisp value of appropriate type for the property *property-name*.

Values

value	A Lisp value of appropriate type for the property *property-name*.

Description

The functions **`browser-pane-property-get`** and **`browser-pane-property-put`** get or set the value of a specified Windows property of the underlying browser of *pane*.

property-name has to be one of the properties listed in the Properties section of the documentation of IWebBrowser2 in the MSDN and *value* should be of the appropriate type for that property when setting it.

Notes

1. **`browser-pane-property-get`** and **`browser-pane-property-put`** are implemented on Microsoft Windows only.
2. **`browser-pane-property-get`** and **`browser-pane-property-put`** do not correspond to the methods "GetProperty" and "PutProperty" of IWebBrowser2.

See also

`browser-pane`

button *Class*

Summary

A class of pane that displays either a piece of text or an image, and that performs an action when pressed. Certain types of buttons can also be selected and deselected.

Package

`capi`

Superclasses

`simple-pane`
`item`

Subclasses

`push-button`
`radio-button`
`check-button`

Initargs

`:interaction`	The interaction style for the button.
`:selected`	For radio button and check button styles, if **`selected`** is set to **`t`**, the button is initially selected.
`:callback`	Specifies the callback to use when the button is selected.
`:image`	An image for the button (or **`nil`**).
`:selected-image`	The image used when the button is selected.
`:enabled`	If **`nil`** the button cannot be selected.
`:cancel-p`	If true the button is the "Cancel" button, that is, the button selected by the **`Escape`** key.
`:default-p`	If true the button is the default button, that is, the button selected by the **`Return`** key.
`:disabled-image`	The image for the button when disabled (or **`nil`**), only implemented on Motif and Microsoft Windows.
`:selected-disabled-image`	The image used when the button is selected and disabled, only implemented on Motif and Microsoft Windows.
`:armed-image`	The image used when the button is pressed and *interaction* is **`:no-selection`**, only implemented on GTK+ and Motif and Microsoft Windows.
`:mnemonic`	A character, integer or symbol specifying a mnemonic for the button, only implemented on Microsoft Windows and GTK+.
`:mnemonic-text`	A string specifying the text and a mnemonic, only implemented on Microsoft Windows and GTK+.
`:mnemonic-escape`	A character specifying the mnemonic escape. The default value is **`#\&`**, only implemented on Microsoft Windows and GTK+.

Accessors

`button-selected`
`button-image`
`button-armed-image`
`button-selected-image`
`button-disabled-image`
`button-selected-disabled-image`
`button-enabled`
`button-cancel-p`
`button-default-p`

Description

The class **`button`** is the class that **`push-button`**, **`radio-button`**, and **`check-button`** are built on. It can be displayed either with text or an image, and a callback is called when the button is clicked. It inherits all of its textual behavior from **`item`**, including the slot *text* which is the text that appears in the button.

Rather than creating direct instances of **`button`**, you usually create instances of its subclasses, each of which has a specific interaction style. Occasionally it may be easier to instantiate **`button`** directly with the appropriate value of *interaction* (for instance, when the interaction style is only known at run-time) but you may not use such a button as an item in a **`button-panel`**.

The values allowed for *interaction* are as follows:

`:no-selection` A push button.

`:single-selection` A radio button.

`:multiple-selection`

A check button.

Both radio buttons and check buttons can have a selection which can be set using the initarg **`:selected`** and the accessor **`button-selected`**.

The button's callback gets called when the user clicks on the button, and by default gets passed the data in the button and the interface. This can be changed by specifying a callback type as described in the description of callbacks. The following callbacks are accepted by buttons:

`:selection-callback`

Called when the button is selected.

`:callback` For buttons this is a synonym of **`:selection-callback`**.

`:retract-callback` Called when the button is deselected.

By default, *image* and *disabled-image* are **`nil`**, meaning that the button is a text button, but if *image* is provided then the button displays an image instead of the text. The image can be an **`external-image`** or any object accepted by **`load-image`**, including a .ico file on Microsoft Windows. The disabled image is the image that is shown when the button is disabled (or **`nil`**, meaning that it is left for the window system to decide how to display the image as disabled). On some platforms the system computes the disabled image and so *disabled-image* is ignored.

The button's actions can be enabled and disabled with the *enabled* slot, and its associated accessor **`button-enabled`**. This means that when the button is disabled, pressing on it does not call any callbacks or change its selection.

Note that the class **`button-panel`** provides functionality to group buttons together, and should normally be used in preference to creating individual buttons yourself. For instance, a **`radio-button-panel`** makes a number of radio buttons and also controls them such that only one button is ever selected at a time.

A mnemonic is an underlined character within the button *text* or the printed representation of the button *data* which can be entered to select the button. The value *mnemonic* is interpreted as described for **menu**.

An alternative way to specify a mnemonic is to pass *mnemonic-text*. This is a string which provides the text for the button and also specifies the mnemonic character. *mnemonic-text* and *mnemonic-escape* are interpreted in just the same way as the *mnemonic-title* and *mnemonic-escape* of **menu**.

Notes

1. The **simple-pane** initarg *foreground* is not supported for buttons on Windows and Cocoa.
2. The *disabled-image*, *armed-image* and *selected-disabled-image* will work on Microsoft Windows provided you are running with the themed look-and-feel (which is the default). See **19.1.1 Using Windows themes**.

Examples

In the following example a button is created. Using the **button-enabled** accessor the button is then enabled and disabled.

```
(setq button
      (capi:contain (make-instance
                     'capi:push-button
                     :text "Press Me")))

(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) nil button)

(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) t button)
```

In the next example a button with an image instead of text is created.

```
(setq button
      (capi:contain
       (make-instance
        'capi:push-button
        :image
        (example-file
         "capi/applications/images/info.bmp"))))
```

The following examples illustrate mnemonics:

```
(defun egg (&rest ignore)
  (declare (ignore ignore))
  (capi:display-message "Egg"))

(capi:contain
 (make-instance 'capi:push-button
                :selection-callback 'egg
                :mnemonic-text "Chicken && Rice"))

(capi:contain
 (make-instance 'capi:push-button
                :data "Chicken"
                :selection-callback 'egg
                :mnemonic #\k))
```

Compare this with the previous example: the **#\k** does not appear and the **#\e** becomes the mnemonic:

```
(capi:contain
 (make-instance 'capi:push-button
                :selection-callback 'egg
                :mnemonic-escape  #\k
                :mnemonic-text "Chicken"))
```

Also see these examples:

```
(example-edit-file "capi/buttons/")
```

See also

button-panel
callbacks
3.10 Button elements
13.10 Working with images

button-panel *Class*

Summary

The class **button-panel** is a pane containing a number of buttons that are laid out in a particular style, and that have group behavior.

Package

capi

Superclasses

choice
titled-object
simple-pane

Subclasses

push-button-panel
radio-button-panel
check-button-panel

Initargs

:layout-class	The type of layout for the buttons.
:layout-args	Initialization arguments for the layout.
:callbacks	The selection callbacks for each button.
:button-class	The class of the buttons.
:images	A list.
:disabled-images	A list.
:armed-images	A list.
:selected-images	A list.

:selected-disabled-images

	A list.
:help-keys	A list.
:default-button	Specifies the default button.
:cancel-button	Specifies the cancel button.
:mnemonics	A list specifying mnemonics for the buttons, only implemented on Microsoft Windows.
:mnemonic-items	A list of strings, each specifying the text and a mnemonics, only implemented on Microsoft Windows.
:mnemonic-escape	A character specifying the mnemonic escape. The default value is **#\&**s, only implemented on Microsoft Windows.
:mnemonic-title	A string specifying the title and a mnemonics, only implemented on Microsoft Windows.

Accessors

pane-layout

Description

The class **button-panel** inherits most of its behavior from **choice**, which is an abstract class providing support for handling items and selections. By default, a button panel has single selection interaction style (meaning that only one of the buttons can be selected at any one time), but this can be changed by specifying an *interaction*.

The subclasses **push-button-panel**, **radio-button-panel** and **check-button-panel** are provided as convenience classes, but they are just button panels with different interactions (**:no-selection**, **:single-selection** and **:multiple-selection** respectively).

The layout of the buttons is controlled by a layout of class *layout-class* (which defaults to **row-layout**) but this can be changed to be any other CAPI layout. When the layout is created, the list of initargs *layout-args* is passed to **make-instance**.

Each button uses the callbacks specified for the button panel itself, unless the argument *callbacks* is specified. *callbacks* should be a list (one element per button). Each element of *callbacks*, if non-nil, will be used as the selection callback of the corresponding button.

button-class, if supplied, determines the class used for each of the buttons. This should be the class appropriate for the *interaction*, or a subclass of it. The default behavior is to create buttons of the class appropriate for the *interaction*.

Each of *images*, *disabled-images*, *armed-images*, *selected-images*, *selected-disabled-images* and *help-keys*, if supplied, should be a list of the same length as *items*. The values are passed to the corresponding item, and interpreted as described for **button**. The **button-panel** *images* values map to **button** *image* arguments, and so on.

For **button-panel** and its subclasses, the *items* supplied to the **:items** initarg and **(setf collection-items)** function can contain button objects. In this case, the button is used directly in the button panel rather than a button being created by the CAPI.

This allows button size and spacing to be controlled explicitly. Note that the button must be of the appropriate type for the subclass of **button-panel** being used, as shown in the following table:

Button and panel classes

Button panel class	Button class
push-button-panel	**push-button**
radio-button-panel	**radio-button**
check-button-panel	**check-button**

For example:

```
(let ((button1 (make-instance 'capi:push-button
                              :text "button1"
                              :internal-border 20
                              :visible-min-width 200))
      (button2 (make-instance 'capi:push-button
                              :text "button2"
                              :internal-border 20
                              :visible-min-width 200)))
  (capi:contain (make-instance 'capi:push-button-panel
                           :items (list button1 button2)
                           :layout-args '(:x-gap 30))))
```

default-button specifies which button is the default (selected by pressing **Return**). It should be equal to a member of *items* when compared by *test-function*. If the items are non-immediate objects such as strings or **button** objects, you must ensure either that the same (**eq**) object is passed in *items* as in *default-button*, or that a suitable *test-function* is supplied.

cancel-button specifies which button is selected by pressing **Escape**. The comparison with members of items is as for *default-button*.

mnemonics is a list of the same length as *items*. Each element is a character, integer or symbol specifying the mnemonic for the corresponding button in the same way as described for **menu**.

mnemonic-items is an alternate way to specify the mnemonics in a button panel. It is a list of the same length as *items*. Each element is a string which is interpreted for the corresponding **button** as its *mnemonic-text* initarg.

mnemonic-title and *mnemonic-escape* are interpreted as for **menu**. *mnemonic-escape* specifies the escape character for mnemonics both in the buttons and in the pane's title.

Compatibility note

Button panels now default to having a maximum size constrained to their minimum size as this is useful when attempting to layout button panels into arbitrary spaces without them changing size. To get the old behavior, specify **:visible-max-width nil** in the **make-instance**.

Examples

```
(capi:contain (make-instance
               'capi:button-panel
               :items '(:red :green :blue)
               :print-function 'string-capitalize))
```

```
(setq buttons
      (capi:contain
       (make-instance
        'capi:button-panel
        :items '(:red :green :blue)
        :print-function 'string-capitalize
        :interaction :multiple-selection)))
```

```
(capi:apply-in-pane-process
 buttons #'(setf capi:choice-selected-items)
 '(:red :green) buttons)
```

```
(capi:contain (make-instance
               'capi:button-panel
               :items '(1 2 3 4 5 6 7 8 9)
               :layout-class 'capi:grid-layout
               :layout-args '(:columns 3)))
```

This example illustrates use of *default-button* and *test-function*:

```
(capi:contain
 (make-instance 'capi:push-button-panel
                :items '("one" "two" "three")
                :default-button "two"
                :test-function 'equalp
                :selection-callback
                'capi:display-message))
```

Also see these example files:

```
(example-edit-file "capi/buttons/buttons")
```

```
(example-edit-file "capi/buttons/button-panel-layout")
```

See also

radio-button
check-button
push-button
set-button-panel-enabled-items
5 Choices - panes with items

calculate-constraints *Generic Function*

Summary

Calculates the internal constraints of a pane.

Package

capi

Signature

calculate-constraints *pane*

Arguments

pane⇓ A CAPI pane or layout.

Description

The generic function **`calculate-constraints`** calculates the internal constraints for *pane* according to the sizes of its children, and sets these values into its geometry cache. It can also store other information about the constraints for later use by **`calculate-layout`**.

When the pane does not scroll in the relevant dimension, all the geometry hints (**`:external-min-width`**, **`:visible-max-height`** and so on) override the values that are computed by **`calculate-constraints`**.

When the pane does scroll in the relevant dimension, **`:internal-min-width`** and **`:internal-min-height`** override the values that are computed by **`calculate-constraints`**. (**`:internal-max-width`** and **`:internal-max-height`** are ignored when scrolling.)

See **6.4.1 Width and height hints** for a description of internal and external constraints.

The CAPI calls **`calculate-constraints`** for each pane and layout that it displays.

When creating your own layout, you should define a method for **`calculate-constraints`** that sets the values of the following geometry slots based on the constraints of its children.

`%min-width%`	The minimum width of *pane*.
`%max-width%`	The maximum width of *pane*.
`%min-height%`	The minimum height of *pane*.
`%max-height%`	The maximum height of *pane*.

See **`with-geometry`** for more details of these slots.

The constraints of any CAPI element can be found by calling **`get-constraints`**.

Note: Unless your layout is a direct subclass of **`layout`**, you must ensure that the **`calculate-constraints`** methods from the superclasses are called. You can do this by calling **`call-next-method`** or defining your **`calculate-constraints`** method as an **`:after`** method.

See also

`calculate-layout`
`define-layout`
`get-constraints`
`element`
`layout`
`with-geometry`
7 Programming with CAPI Windows

calculate-layout *Generic Function*

Summary

Provides a method for laying out the children of a new layout.

Package

`capi`

Signature

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

Arguments

layout⇓ A **`layout`**.

x⇓, *y*⇓, *width*⇓, *height*⇓

Integers.

Description

The generic function **`calculate-layout`** is called by the CAPI to set the position and size of the children of *layout*.

x, *y*, *width* and *height* are the position and size of a rectangle that should contain the children.

When defining a new subclass of **`layout`** using **`define-layout`**, a **`calculate-layout`** method must be provided that sets the position and size of each of the layout's children. This method must try to obey the constraints specified by its children (its minimum and maximum size) and should only break them when it becomes impossible to fit the constraints of all of the children. Use *x*, *y*, *width* and *height* to calculate a suitable position and size for each of the children and set them using the macro **`with-geometry`**, which works in a similar way to **`with-slots`**.

Examples

```
(example-edit-file "capi/layouts/buffer-layout")
```

```
(example-edit-file "capi/layouts/wrapping-layout")
```

See also

`get-constraints`
`with-geometry`
`interpret-description`
6 Laying Out CAPI Panes

callbacks *Class*

Summary

The class **`callbacks`** is used as a mixin by classes that provide callbacks.

Package

`capi`

Superclasses

`capi-object`

Subclasses

`collection`

item
menu-object

Initargs

:callback-type The type of arguments for the callbacks.

:selection-callback
The callback for selecting an item.

:extend-callback The callback for extending the selection.

:retract-callback The callback for deselecting an item.

:action-callback The callback for an action.

:alternative-action-callback
The callback for an alternative action in **choice** and its subclasses.

Accessors

callbacks-callback-type
callbacks-selection-callback
callbacks-extend-callback
callbacks-retract-callback
callbacks-action-callback

Description

Each callback function can be one of the following:

function Call the function.

list Apply the head of the list to the tail.

:redisplay-interface

Call **redisplay-interface** on the top-level interface.

:redisplay-menu-bar

Call **redisplay-menu-bar** on the top-level interface.

The slot value *callback-type* determines which arguments get passed to each of the callbacks. It can be any of the following values, and passes the corresponding data to the callback function:

:collection-data **(***collection data***)**

:data **(***item-data***)**

:data-element **(***item-data element***)**

:data-interface **(***item-data interface***)**

:element **(***element***)**

:element-data **(***element item-data***)**

:element-item **(***element item***)**

:interface-data **(***interface item-data***)**

:item **(***item***)**

:item-element	**(***item element***)**
:item-interface	**(***item interface***)**
:interface-item	**(***interface item***)**
:interface	**(***interface***)**
:full	**(***item-data item interface***)**
:focus	The pane with the current input focus.
:none	**()**
nil	**()**

callback-type can also be a list containing any of **:focus**, **:data**, **:element**, **:interface**, **:collection**, **:item**.

The *item-data* variable is the item's data if the item is of type **item**, otherwise it is the item itself, as for *item*. The *item* variable means the item itself. The *interface* is the **element-interface** of the element. *collection* is the element's **collection**, if there is one. The *element* variable means the element containing the callback itself.

In a **choice**, the *alternative-action-callback* is invoked by a gesture which is the *action-callback* gesture modified by the **Shift** key on Microsoft Windows and GTK+, and modified by the **Command** key on Cocoa.

alternative-action-callback is applicable only to **choice** and its subclasses.

Apart from being invoked with a different gesture, the *alternative-action-callback* has exactly the same semantics as *action-callback*.

Examples

```
(example-edit-file "capi/choice/alternative-action-callback")
```

See also

abort-callback
choice
attach-interface-for-callback
3.4 Callbacks
5.10.3 Callbacks in choices
8 Creating Menus

call-editor *Generic Function*

Summary

Executes an editor command in an **editor-pane**.

Package

capi

Signature

call-editor *editor-pane command*

Arguments

editor-pane⇓ An **editor-pane**.

command⇓ A string.

Description

The generic function **call-editor** executes the editor command *command* in the current buffer in *editor-pane*.

It can be used directly in a callback for an interface that contains *editor-pane*. See **11.4 Connecting an interface to an application**. In other cases, take care to modify displayed CAPI interfaces only in their own process: **execute-with-interface** and **apply-in-pane-process** are useful for this.

The *before-input-callback* and *after-input-callback* of the **editor-pane** are called when **call-editor** is called.

Examples

```
(setq editor (capi:contain
              (make-instance 'capi:editor-pane
                             :text "abc")))


(capi:apply-in-pane-process
 editor 'capi:call-editor editor "End Of Buffer")
```

Also see this example:

```
(example-edit-file "capi/editor/editor-pane")
```

See also

apply-in-pane-process
editor-pane
execute-with-interface
10.6 In-place completion

can-use-metafile-p *Function*

Summary

Queries whether metafiles can be used.

Package

capi

Signature

can-use-metafile-p &optional *screen* **=>** *result*

Arguments

screen⇓ An object accepted by the function **convert-to-screen**.

Values

result A boolean.

Description

The function **can-use-metafile-p** is the predicate for whether the default library (if no argument is passed) or a specified **screen** (if an argument is passed) can use metafiles.

If the argument *screen* is supplied, it is converted to a screen by **convert-to-screen**.

Examples

```
(example-edit-file "capi/graphics/metafile")
```

See also

convert-to-screen
default-library

capi-object *Class*

Summary

The class **capi-object** is the superclass of all CAPI classes.

Package

capi

Superclasses

standard-class

Subclasses

item
callbacks
element
interface
pinboard-object

Initargs

:name	The name of the object.
:plist	A property list for storing miscellaneous information.

Accessors

capi-object-name
capi-object-plist

Description

The class **capi-object** provides a name and a property list for general purposes, along with the accessors **capi-object-name** and **capi-object-plist** respectively. The name of a **capi-object** is defaulted by **define-interface** to be the name of the slot into which the object is put.

Examples

```
(setq object (make-instance 'capi:capi-object
                            :name 'test))
```

```
(capi:capi-object-name object)
```

```
(setf (capi:capi-object-plist object)
      '(:red 1 :green 2 :blue 3))
```

```
(capi:capi-object-property object :green)
```

See also

capi-object-property
18.5 Object properties and name

capi-object-property *Accessor*

Summary

Accesses properties in the property list of a **capi-object**.

Package

capi

Signature

capi-object-property *object property* **=>** *value*

setf (capi-object-property *object property***)** *value* **=>** *value*

Arguments

object⇓	A **capi-object**.
property⇓	A Lisp object.
value⇓	A Lisp object.

Values

value⇓	A Lisp object.

Description

The accessor **capi-object-property** gets and sets the property named *property* in the property list of *object*. *value* can be any Lisp object.

All CAPI objects contain a property list, similar to the plist of a symbol. The recommended ways of accessing properties are **capi-object-property** and **(setf capi-object-property)**. To remove a property, use the function **remove-capi-object-property**.

Examples

In this example a list panel is created, and a test property is set and examined using **capi-object-property**.

```
(setq pane (make-instance 'capi:list-panel
                          :items '(1 2 3)))

(capi:capi-object-property pane 'test-property)

(setf (capi:capi-object-property pane 'test-property)
      "Test")
(capi:capi-object-property pane 'test-property)

(capi:remove-capi-object-property pane 'test-property)
(capi:capi-object-property pane 'test-property)
```

See also

capi-object
remove-capi-object-property
18.5 Object properties and name

check-button *Class*

Summary

A check button is a button that can be either selected or deselected, and its selection is independent of the selections of any other buttons.

Package

capi

Superclasses

button
titled-object

Description

The class **check-button** inherits most of its behavior from the class **button**. Note that it is normally best to use a **check-button-panel** rather than make the individual buttons yourself, as the button panel provides functionality for handling groups of buttons. However, **check-button** can be used if you need to have more control over the button's behavior.

Examples

The following code creates a check button.

```
(setq button (capi:contain
              (make-instance 'capi:check-button
                             :text "Press Me")))
```

The button can be selected and deselected using this code.

```
(capi:apply-in-pane-process
 button #'(setf capi:button-selected) t button)

(capi:apply-in-pane-process
 button #'(setf capi:button-selected) nil button)
```

The following code disables and enables the button.

```
(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) nil button)

(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) t button)
```

See also

push-button
radio-button
button-panel
3.10 Button elements

check-button-panel *Class*

Summary

A class of panes containing a group of buttons each of which can be selected or deselected.

Package

capi

Superclasses

button-panel

Description

The class **check-button-panel** inherits all of its behavior from **button-panel**, which itself inherits most of its behavior from **choice**. Thus, the **check-button-panel** can accept *items, callbacks*, and so on.

Examples

```
(capi:contain (make-instance
               'capi:check-button-panel
               :title "Select some packages"
```

```
                 :items '("CAPI" "LISPWORKS" "CL-USER")))

(setq buttons (capi:contain
               (make-instance
                'capi:check-button-panel
                :title "Select some packages"
                :items '("CAPI" "LISPWORKS" "CL-USER")
                :layout-class 'capi:column-layout)))

(capi:choice-selected-items buttons)
```

Also see this example:

```
(example-edit-file "capi/buttons/buttons")
```

See also

check-button
push-button-panel
radio-button-panel
5 Choices - panes with items

choice *Class*

Summary

An abstract class that collects together a group of items, and provides functionality for displaying and selecting them.

Package

capi

Superclasses

collection

Subclasses

button-panel
double-list-panel
extended-selection-tree-view
graph-pane
list-panel
menu-component
option-pane
toolbar-component
tree-view

Initargs

:interaction	The interaction style of the choice.
:selection	The indexes of the choice's selected items.
:selected-item	The selected item for a single selection choice.

:selected-items A list of the selected items.

:keep-selection-p If **t**, retains any selection when the items change.

:initial-focus-item

If supplied, this should be an item in the choice.

Accessors

choice-selection

Readers

choice-interaction
choice-initial-focus-item

Description

The class **choice** inherits most of its behavior from **collection**, and then provides the selection facilities itself. The classes **list-panel**, **button-panel**, **option-pane**, **menu-component** and **graph-pane** inherit from it, and so it plays a key role in CAPI applications.

A **choice** can have one of four different interaction styles, and these control how it behaves when an item is selected by the user. *interaction* can be one of:

:no-selection The choice behaves just as a collection.

:single-selection The choice can have only one selected item.

:multiple-selection

The choice can have multiple selected items, except on macOS.

:extended-selection

An alternative to **multiple-selection**.

With *interaction* **:no-selection**, the choice cannot have a selection, and so behaves just as a collection would.

With *interaction* **:single-selection**, the choice can only have one item selected at a time. When a new selection is made, the old selection is cleared and its *selection-callback* is called. The *selection-callback* is also called when the user invokes the selection gesture on the selected item.

With *interaction* **:multiple-selection**, the choice can have any number of items selected, and selecting an item toggles its selection status. The *selection-callback* is called when an item becomes selected, and the *retract-callback* is called when an item is deselected. **:multiple-selection** is not supported for lists on macOS.

With *interaction* **:extended-selection**, the choice can have any number of items selected as with **:multiple-selection** interaction, but the usual selection gesture removes the old selection. However, there is a window system-specific means of extending the selection. When an item is selected the *selection-callback* is called, when the selection is extended the *extend-callback* is called, and when an item is deselected the *retract-callback* is called.

On macOS, the selection gesture is mouse (left button) click. Deselection and discontinuous selections are made by **Command+Click**, and a continuous selection is made by **Shift+Click**, regardless of whether if *interaction* is **:multiple-selection** or **:extended-selection**.

The choice's selection stores the indices of the currently selected item, and is a single number for single selection choices and a list for all other interactions. Therefore when calling **(setf choice-selection)** you must pass an integer or **nil** if *interaction* is **:single-selection**, and you must pass a list of integers if *interaction* is **:multiple-selection** or

:extended-selection.The functions **choice-selected-item** and **choice-selected-items** treat the selection in terms of the items themselves as opposed to their indices.

Usually when a choice's items are changed using **(setf collection-items)** the selection is lost.

However, if the choice was created with **:keep-selection-p t**, then the selection is preserved over the change.

initial-focus-item, if supplied, specifies the item which has the input focus when the choice is first displayed.

Notes

When calling **(setf choice-selection)** you must pass an integer or **nil** when *interaction* is **:single-selection**. You must pass a list for other values of *interaction*.

Compatibility note

In LispWorks 5.0 and earlier versions, for interaction **:single-selection** the *selection-callback* is called only after a new selection is made.

Examples

The following example defines a choice with three possible selections.

```
(setq choice (make-instance 'capi:choice
                            :items '("One" "Two" "Three")
                            :selection 0))
```

```
(capi:display-message "Selection: ~S"
                      (capi:choice-selection choice))
```

```
(capi:choice-selected-item choice)
```

The selection is changed using the following code.

```
(setf (capi:choice-selection choice) 1)
```

```
(capi:choice-selected-item choice)
```

Also see these examples:

```
(example-edit-file "capi/choice/")
```

```
(example-edit-file "capi/graphics/graph-pane")
```

See also

choice-selected-item
choice-selected-item-p
choice-selected-items
choice-update-item
redisplay-collection-item
remove-items
replace-items
5 Choices - panes with items

choice-selected-item *Accessor*

Summary

Returns the currently selected item in a single selection choice.

Package

capi

Signature

choice-selected-item *choice* **=>** *item*

(setf choice-selected-item) *item choice* **=>** *item*

Arguments

choice⇓ A **choice**.

item A Lisp object.

Values

item A Lisp object.

Description

The accessor **choice-selected-item** accesses the currently selected item in a single selection choice. A **setf** method is provided as a means of setting the selection. Note that the items are compared by the *test-function* of *choice* - see **collection** or the example below.

It is an error to call this function on choices with different interactions — in that case, you should use **choice-selected-items**.

Examples

This example illustrates setting the selection. First we set up a single selection choice — in this case, a **list-panel**.

```
(setq list (capi:contain
            (make-instance 'capi:list-panel
                           :items '(a b c d e)
                           :selection 2)))
```

The following code line returns the selection of the list panel.

```
(capi:choice-selected-item list)
```

The selection can be changed, and the change viewed, using the following code.

```
(capi:apply-in-pane-process
 list #'(setf capi:choice-selected-item) 'e list)

(capi:choice-selected-item list)
```

This example illustrates the effect of the *test-function*. Make a choice with *test-function* **cl:eq**:

```
(setf *list*
      (capi:contain
       (make-instance 'capi:list-panel
                      :items (list "a" "b" "c")
                      :selection 0
                      :visible-min-height :text-height)))
```

This call loses the selection since **(eq "b" "b")** fails:

```
(capi:apply-in-pane-process
 *list* #'(setf capi:choice-selected-item)
 "b" *list*)
```

Change the test function:

```
(capi:apply-in-pane-process
 *list* #'(setf capi:collection-test-function)
 'equal *list*)
```

This call sets the selection since **(equal "b" "b")** succeeds:

```
(capi:apply-in-pane-process
 *list* #'(setf capi:choice-selected-item)
 "b" *list*)
```

See also

choice
choice-selected-item-p
choice-selected-items
collection
5 Choices - panes with items

choice-selected-item-p *Function*

Summary

Checks if an item is currently selected in a choice.

Package

capi

Signature

choice-selected-item-p *choice item* **=>** *result*

Arguments

choice⇓	A **choice**.
item⇓	An item.

Values

result A boolean.

Description

The function **`choice-selected-item-p`** is the predicate for whether an item *item* of the choice *choice* is selected.

Note that the items are compared by the *test-function* of *choice* - see **`collection`** for details.

Examples

```
(setq list
      (capi:contain
       (make-instance 'capi:list-panel
                      :items '(a b c d)
                      :selection 2
                      :visible-min-height
                      '(:character 4))))
```

```
(capi:choice-selected-item-p list 'c)
=>
t
```

Now click on another item.

```
(capi:choice-selected-item-p list 'c)
=>
nil
```

See also

`choice`
`collection`

choice-selected-items *Accessor*

Summary

Returns the currently selected items in a choice as a list of the items.

Package

`capi`

Signature

`choice-selected-items` *choice* **`=>`** *items*

`(setf choice-selected-items)` *items choice* **`=>`** *items*

Arguments

choice⇓ A **`choice`**.

items A list of items.

Values

items A list of items.

Description

The accessor **choice-selected-items** accesses the currently selected items in a choice as a list of the items. A **setf** method is provided as a means of setting the currently selected items. Note that the items are compared by the *test-function* of *choice* - see **collection** for details.

In the case of **:single-selection** choices, it is usually easier to use the complementary function **choice-selected-item**, which returns the selected item as its result.

Examples

First we set up a **:multiple-selection** choice — in this case, a list panel.

```
(setq list (capi:contain
            (make-instance
             'capi:list-panel
             :items '(a b c d e)
             :visible-min-height '(:character 5)
             :interaction :multiple-selection
             :selection '(1 3))))
```

The following code line returns the selections of the list.

```
(capi:choice-selected-items list)
```

The selections of the list panel can be changed and redisplayed using the following code.

```
(capi:apply-in-pane-process
 list #'(setf capi:choice-selected-items)
 '(a c e) list)


(capi:choice-selected-items list)
```

Note that *interaction* **:multiple-selection** is not supported for lists on macOS.

See also

choice
choice-selected-item
choice-selected-item-p
choice-selected-items
collection
5 Choices - panes with items

choice-update-item *Function*

Summary

Updates an item in a choice.

Package

capi

Signature

choice-update-item *choice item*

Arguments

choice⇓ A **choice**.

item⇓ An item.

Description

The function **choice-update-item** updates the display of the item *item* in the choice *choice*. It should be called if the display of *item* (that is, the string returned by the *print-function*) changes.

Examples

Create a list panel that displays the status of something:

```
(defun my-print-an-item (item)
        (format nil "~a: ~a"
                (substitute-if-not #\space
                                   'alphanumericp
                                   (symbol-name item))
                (symbol-value item)))


(defvar *status-one* :on)
(defvar *status-two* :off)


(setq list
      (capi:contain
       (make-instance
        'capi:list-panel
        :items '(*status-one* *status-two*)
        :print-function 'my-print-an-item
        :visible-min-height :text-height
        :visible-min-width :text-width)))
```

Setting the status variables does not change the display:

```
(setq *status-one* :error)
```

Update the item to change the display:

```
(capi:choice-update-item list '*status-one*)
```

This example also demonstrates **choice-update-item**:

```
(example-edit-file "capi/choice/alternative-action-callback")
```

See also

choice

clipboard *Function*

Summary

Returns the contents of the system clipboard.

Package

capi

Signature

clipboard *self* **&optional** *format* **=>** *result*

Arguments

self⇓ A displayed CAPI pane or interface.

format⇓ A keyword.

Values

result A string, an **image**, a Lisp object, or **nil**.

Description

The function **clipboard** returns the contents of the system clipboard as a string, or **nil** if the clipboard is empty.

format controls what kind of object is read. The following values of *format* are recognized:

:string The object is a string. This is the default value.

:image The object is of type **image**, converted from whatever format the platform supports.

:value The object is the Lisp value.

:metafile The object is a metafile.

When *format* is **:image**, the image returned by **clipboard** is associated with *self*, so you can free it explicitly with **free-image** or it will be freed automatically when the pane is destroyed.

When *format* is **:metafile** the object is a metafile which should be freed using **free-metafile** when no longer needed. See also **draw-metafile** and **draw-metafile-to-image**. *format* **:metafile** is not supported on GTK+ or X11/Motif.

The Microsoft Windows clipboard is usually set by the user with the **Ctrl+C** and **Ctrl+X** gestures. Note that the LispWorks editor uses these gestures when in Windows emulation mode.

On X11/Motif, various gestures may set the clipboard. Note that LispWorks uses **`Ctrl+C`** and **`Ctrl+X`** when in KDE/Gnome editor emulation mode. The X clipboard can also be accessed by running the program **`xclipboard`** or the Emacs function **`x-get-clipboard`**.

The macOS clipboard is usually set by the user with the **`Command+C`** and **`Command+X`** gestures.

See also

`clipboard-empty`
`draw-metafile`
`draw-metafile-to-image`
`free-image`
`free-metafile`
`image`
`selection`
`set-clipboard`
`text-input-pane-paste`
18.6 Clipboard

clipboard-empty *Function*

Summary

Determines whether the system clipboard contains an object of the specified kind.

Package

`capi`

Signature

`clipboard-empty` *self* **`&optional`** *format* **`=>`** *result*

Arguments

self⇓ A displayed CAPI pane or interface.

format⇓ A keyword.

Values

result **`t`** or **`nil`**.

Description

The function **`clipboard-empty`** returns **`nil`** if there is an object of the kind indicated by *format* on the clipboard associated with *self*, or **`t`** otherwise.

format controls what kind of object is checked. The allowed values of *format* are as described for **`clipboard`**.

See also

`clipboard`

image
18.6 Clipboard

clone *Generic Function*

Summary

Creates a copy of a CAPI object.

Package

capi

Signature

clone *capi-object* **=>** *cloned-object*

Arguments

capi-object⇓ A **capi-object**.

Values

cloned-object⇓ A copy of *capi-object*.

Description

The generic function **clone** returns a new object *cloned-object* which is a copy of *capi-object*. It does not share any data with *capi-object*, but has a copy of the useful part of its state.

The system contains methods on **clone**. You may add methods on your own interface classes.

See also

capi-object

cocoa-default-application-interface *Class*

Summary

The class supporting application menus and message processing for a Cocoa application.

Package

capi

Superclasses

interface

Initargs

`:message-callback`	A function or **`nil`**.
`:application-menu`	**`nil`**, a **`menu`**, or the name of a slot containing a **`menu`** in the application interface.
`:dock-menu`	**`nil`**, a **`menu`**, or a function designator.

Accessors

`application-interface-message-callback`
`application-interface-application-menu`
`application-interface-dock-menu`

Description

The class **`cocoa-default-application-interface`** supports the application menu, application messages and other functionality for a Cocoa application.

All Cocoa applications in LispWorks for Macintosh have an application interface, which is a hidden interface that provides the following:

1. The application menu (the leftmost menu in the menu bar, named after the application). See *application-menu* below.
2. The menu bar items that are displayed when no other interfaces are on the screen. See *menu-bar-items* in **`interface`** and *menu-bar* in **`define-interface`**.
3. An optional Dock context menu. See *dock-menu* below.
4. Optional application message processing. See *message-callback* below.
5. Control over the lifecycle and *display-state* of the application as a whole.

If you wish to override the defaults, then you should first define a subclass of **`cocoa-default-application-interface`** with your changes. Then set a single instance of this subclass as the application interface by calling **`set-application-interface`** before any CAPI functions that make the screen object (such as **`convert-to-screen`** and **`display`**).

Do not call **`display`** with a subclass of **`cocoa-default-application-interface`** - the application interface does not have a window on the screen and should be created in addition to the visible interfaces in your application.

When non-nil, *message-callback* should be a function with signature:

interface message **`&rest`** *args*

message-callback will be called for various application messages. The *interface* argument will be the application interface and the *message* argument will be a keyword. The *message* argument will be one of the following:

`:open-file` This message is invoked when the user double-clicks on a document associated with the application or drags a document into the application icon. The *args* contain the name of the file to open.

`:finished-launching`

This message is invoked just after the user has started the application and all other initialization has been done (including any **`:open-file`** message if applicable). You can use it to open a default document for example. There are no *args*.

application-menu controls the application's main menu. If this is **`nil`**, then a minimal application menu will be made using the title of the application interface, otherwise it should be a **`menu`** containing the usual items or the name of a slot containing such a menu in the application interface. Note that the **Quit** item in the *application-menu* needs to call **destroy** on the interface, rather than call **lw:quit**.

dock-menu provides a menu for use by the macOS Dock icon. If the value is **nil** (the default), then the standard menu is used. If *dock-menu* is a function designator, it is called with the application interface as its argument when the menu is popped up and should return a menu. Otherwise *dock-menu* should be a menu, which is used directly. The Dock will add the standard items such as **Quit** to the end of the menu you supply.

interface initargs are interpreted as follows:

- The *activate-callback* is called when the application is activated or deactivated.
- The *create-callback* is called when the application starts up.
- The *destroy-callback* is called when the application shuts down.
- The *confirm-destroy-function* is called to confirm whether the application should shut down.

All of these callbacks execute in the thread that runs the Cocoa event loop, so they can call CAPI and GP functions.

The application interface also allows you to control aspects of the application. In particular:

- The function **destroy** will cause the application to shut down.
- The function **top-level-interface-display-state** will return **:hidden** if the whole application is hidden and will return **:normal** otherwise.
- The function **(setf top-level-interface-display-state)** can be used to perform some operations typically found on the application menu.

The *display-state* value can one of:

:normal	Show the application and activate it.
:restore	Show the application again without activating it.
:hidden	Hide.
:others-hidden	Hide Others.
:all-normal	Show All.

Notes

cocoa-default-application-interface is implemented only in LispWorks for Macintosh with the Cocoa IDE.

Examples

```
(example-edit-file "capi/applications/cocoa-application")

(example-edit-file "capi/applications/cocoa-application-single-window")

(example-edit-file "delivery/macos/multiple-window-application")

(example-edit-file "delivery/macos/single-window-application")
```

See also

set-application-interface
3.9 Special kinds of windows
8 Creating Menus

cocoa-view-pane *Class*

Summary

Allows an arbitrary Cocoa view class to be used on the Macintosh.

Package

capi

Superclasses

simple-pane
titled-object

Initargs

:view-class	A string naming the view class to use.
:init-function	A function that initializes the view class.

Accessors

cocoa-view-pane-view-class
cocoa-view-pane-init-function

Description

The class **cocoa-view-pane** allows an instance of an arbitrary Cocoa view class to be displayed within a CAPI interface.

When the pane becomes visible, the CAPI allocates and initialize a Cocoa view object using the initargs as follows:

- If *view-class* is specified, then it should be a string naming the Cocoa view class to allocate. Otherwise the class **NSView** is allocated.
- If *init-function* is not **nil**, then it should be a function which is called with of two arguments, the pane and a foreign pointer to the newly allocated Cocoa view object. The function should initialize the Cocoa view object in whatever way is required, including invoking the appropriate Objective-C initialization method, and return the initialized view. If *init-function* is **nil** then the Objective-C method **init** is called and the result is returned.

After the Cocoa view has been initialized, the function **cocoa-view-pane-view** can be used the retrieve it.

You can use the functions **(setf cocoa-view-pane-view-class)** and **(setf cocoa-view-pane-init-function)** to modify the *view-class* and *init-function*, but the values will be ignored if this is done after the pane becomes visible.

See the *LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual* for details on using Cocoa.

Notes

cocoa-view-pane is implemented only in LispWorks for Macintosh with the Cocoa IDE.

Examples

The following code uses **cocoa-view-pane** to display an **NSMovieView** displaying an existing movie.

```
(defun show-movie (movie)
  (capi:contain
   (make-instance
    'cocoa-view-pane
    :view-class "NSMovieView"
    :init-function
    #'(lambda (pane view)
        (setq view
              (objc:invoke view "init"))
        (objc:invoke view "setMovie:" movie)
        view))))
```

See also

cocoa-view-pane-view
3.9 Special kinds of windows

cocoa-view-pane-view *Function*

Summary

Returns the Cocoa view of a **cocoa-view-pane**.

Package

capi

Signature

cocoa-view-pane-view *pane* **=>** *view*

Arguments

pane⇓ A **cocoa-view-pane**.

Values

view A foreign pointer to a Cocoa view or **nil**.

Description

The function **cocoa-view-pane-view** returns the Cocoa view for the **cocoa-view-pane** *pane* as a foreign pointer. This view is only accessible when the pane is visible and **nil** is returned in other cases.

Notes

cocoa-view-pane-view is implemented only in LispWorks for Macintosh with the Cocoa IDE. See the *LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual* for details on using Cocoa.

Examples

```
(example-edit-file "objc/movie-view")
```

See also

cocoa-view-pane
3.9 Special kinds of windows

collect-interfaces *Generic Function*

Summary

Finds all interfaces of a given class.

Package

capi

Signature

collect-interfaces *proto* **&key** *screen current-process-first sort-by* **=>** *interfaces*

Arguments

proto⇓	A class, class name, or an interface.
screen⇓	**nil**, the symbol **:any**, a screen, or a keyword naming a library.
current-process-first⇓	A boolean.
sort-by⇓	**:visible** or **:create**.

Values

interfaces⇓	A list.

Description

The generic function **collect-interfaces** returns a list of CAPI interfaces which are instances of the class indicated by *proto*, or subclasses thereof.

If screen is **nil**, the interfaces on the default screen are returned. This is the default. If *screen* is**:any**, *interfaces* includes those on any screen. If *screen* is a **screen** object, the interfaces on that screen are returned. *screen* can also be a library name, currently the accepted values are **:win32**, **:motif** and **:cocoa**.

If interfaces on multiple screens are returned, then those on each screen are grouped together in *interfaces*.

Amongst those for each screen, the interfaces are grouped as follows. If *current-process-first* is true, then the interfaces in the current process appear together at the beginning of the group. If *sort-by* is **:create** then these interfaces are sorted by creation time, otherwise *sort-by* is **:visible** and they are are sorted in Z-order. The interfaces of other processes appear at the end of the group, also sorted according to *sort-by*.

If *current-process-first* is **`nil`**, then the interfaces for each screen are sorted according to *sort-by*.

The default value of *sort-by* is **`:create`** and of *current-process-first* is **`t`**.

See also

`find-interface`
`installed-libraries`

collection *Class*

Summary

A class that collects together a set of items, and provides functionality for accessing and displaying them.

Package

`capi`

Superclasses

`capi-object`
`callbacks`

Subclasses

`choice`

Initargs

`:items`	The items in the collection.
`:print-function`	A function that prints an item.
`:test-function`	A comparison function between two items.
`:items-count-function`	A function which returns the length of items.
`:items-get-function`	A function that returns the *n*th item.
`:items-map-function`	A function that maps a function over the items.
`:accepts-focus-p`	Specifies that the collection should accept input. The default value is **`t`**.
`:help-key`	An object used for lookup of help.

Accessors

`collection-items`
`collection-print-function`
`collection-test-function`

Readers

`collection-items-count-function`

collection-items-get-function
collection-items-map-function
help-key

Description

The main use of the class **collection** is as a part of the class **choice**, which provides selection capabilities on top of the collection handling, and which is used by list panels, button panels and menus amongst others.

The items in the collection are printed by **print-collection-item**.

Items can be instances of the CAPI class **item** or any Lisp object. The main difference is that non-CAPI items use the callbacks specified for the collection, while the CAPI **item**s will use their callbacks in preference if these are specified.

By default, *items* must be a sequence, but this can be changed by specifying *items-get-function*, *items-count-function*, and *items-map-function*.

items-get-function should take as arguments the items and an index, and should return the indexed item. The default is **svref**.

items-count-function should take the items as an argument and should return the number of them.

items-map-function should take as arguments the items, a function *function* and a flag *collect-results-p*, and should call *function* on each of the items in turn. If *collect-results-p* is non-nil, then it should also return the results of these calls in a list.

print-function should be a one argument function which returns a string. The default is **princ-to-string**. To display an item, the collection call *print-function* with the item, and then draws the resulting string (the way it draws is different between the subclasses of **choice**). The time when *print-function* is called is not defined; it may happen before the string is needed for drawing, and may be cached so not called each time the item is drawn. The function **choice-update-item** can be used to flush the cache when needed.

test-function should be suitable for comparing the items in your collection, returning a boolean. For example, if there are both strings and integers amongst your *items*, you should supply *test-function* **cl:equal**. The default value of *test-function* is **cl:eq**.

You can change the items using **(setf collection-items)**. Note that there is an optimization **append-items** that is sometimes useful when adding items.

accepts-focus-p and *help-key* are interpreted as described in **element**.

Examples

The following code uses **push-button-panel**, a subclass of **collection**.

```
(capi:contain (make-instance 'capi:push-button-panel
                             :items '(one two three)))


(capi:contain (make-instance
               'capi:push-button-panel
               :items '(one two three)
               :print-function 'string-capitalize))
```

The following example provides a collection with all values from 1 to 6 by providing an *items-get-function* and an *items-count-function*.

```
(capi:contain (make-instance
               'capi:push-button-panel
               :items 6
               :items-get-function
```

```
                    #'(lambda (items index) (1+ index))
                  :items-count-function
                    #'(lambda (items) items)))
```

Here is an example demonstrating the use of CAPI items in a collections list of items to get more specific callbacks.

```
(defun specific-callback (data interface)
  (capi:display-message "Specific callback for ~S"
                        data))


(defun generic-callback (data interface)
  (capi:display-message "Ordinary callback for ~S"
                        data))


(capi:contain (make-instance
               'capi:list-panel
               :items (list (make-instance
                             'capi:item
                             :text "Special"
                             :data 1000
                             :selection-callback
                             'specific-callback)
                            2 3 4)
               :selection-callback 'generic-callback)
              :visible-min-width 200
              :visible-min-height 200)
```

See also

append-items
count-collection-items
get-collection-item
item
map-collection-items
print-collection-item
search-for-item
3.12 Tooltips
5 Choices - panes with items

collection-find-next-string *Generic Function*

Summary

Finds the next occurrence of the string that was previously searched for in a collection.

Package

capi

Signature

collection-find-next-string *collection* **&key** *set* **=>** *index*

Arguments

collection⇓ A **collection**.

set⇓ A boolean.

Values

index A non-negative integer or **nil**.

Description

The generic function **collection-find-next-string** must be called after one of **collection-search**, **collection-find-string** or **find-string-in-collection** was called on *collection*. It searches for the next item in *collection* with printed representation matching the last string searched for and returns its index, or **nil** if no match is found.

If *set* is true, then if an item matching the string is found, the selection is set to this item. *set* defaults to **t**.

See also

collection-find-string
collection-last-search
find-string-in-collection

collection-find-string *Generic Function*

Summary

Finds the next occurrence of a string in a collection, prompting for the string if it is not supplied.

Package

capi

Signature

collection-find-string *collection* **&key** *set* *string* **=>** *index*

Arguments

collection⇓ A **collection**.

set⇓ A boolean.

string⇓ A string, or **nil**.

Values

index A non-negative integer or **nil**.

Description

The generic function **collection-find-string** calls **find-string-in-collection** with *collection* and *set*.

string is also passed if non-nil. If *string* is **nil**, **collection-find-string** first prompts the user for a string to pass.

set defaults to **t**.

See also

find-string-in-collection

collection-last-search *Generic Function*

Summary

Returns the last string searched for in a collection.

Package

capi

Signature

collection-last-search *collection* **=>** *string*

Arguments

collection⇓ A **collection**.

Values

string⇓ A string, or **nil**.

Description

The generic function **collection-last-search** returns the last string searched for in collection by **find-string-in-collection**.

If neither of these functions has been called on *collection*, then the return value *string* is **nil**.

See also

find-string-in-collection

collection-search *Generic Function*

Summary

The generic function **collection-search** calls **find-string-in-collection** with a string provided by the user.

Package

capi

Signature

collection-search *collection* **&optional** *set*

Arguments

collection⇓	A **collection**.
set⇓	A boolean.

Description

Prompts the user for a string and calls **find-string-in-collection** with *collection*, *set* and this string.

set defaults to **t**.

Notes

collection-search is deprecated. Use **collection-find-string** instead.

See also

collection
collection-find-string
find-string-in-collection

collector-pane *Class*

Summary

Displays an editor buffer with an associates output stream.

Package

capi

Superclasses

editor-pane

Initargs

:buffer-name	The name of a buffer onto an editor stream.
:stream	The editor stream to be collected.

Readers

collector-pane-stream

Description

The class **collector-pane** is a subclass of **editor-pane** which displays the output sent to a particular type of character stream called an editor stream, the contents of which are stored in an editor buffer.

A new instance **collector-pane** can be created to view an existing editor stream by passing the stream itself or by passing the buffer name of that stream.

To create a new stream, either specify *buffer-name* which does not match any existing buffer, or do not pass *buffer-name* in which case the CAPI will create a unique buffer name for you.

To access the stream, use the reader **collector-pane-stream** on the **collector-pane**.

Note that the editor buffer **"Background Output"** is a buffer onto the output stream ***standard-output***.

Examples

Here is an example that creates two collector panes onto a new stream (that is created by the first collector pane).

```
(setq collector (capi:contain
                 (make-instance 'capi:collector-pane)))


(setq *test-stream*
  (capi:collector-pane-stream collector))


(capi:contain
  (make-instance 'capi:collector-pane
                 :stream *test-stream*))


(format *test-stream* "Hello World~%")
```

Finally, this example shows how to create a collector pane onto the "**Background Output**" stream.

```
(capi:contain (make-instance 'capi:collector-pane
               :buffer-name "Background Output"))
```

See also

with-random-typeout
map-typeout
unmap-typeout
3.9.6 Stream panes

color-screen *Class*

Summary

A class for screens that can display color.

Package

capi

Superclasses

screen

Description

Instances of the class **color-screen** are created for color screens. It is primarily available as a means of discriminating on whether or not to use colors in an interface.

See also

element-screen
mono-screen

column-layout *Class*

Summary

A layout which arranges its children in a column.

Package

capi

Superclasses

grid-layout

Initargs

:ratios	The size ratios between the layout's children.
:adjust	The horizontal adjustment for each child.
:gap	The gap between each child.
:uniform-size-p	If **t**, each child in the column has the same height.

Accessors

layout-ratios

Description

The class **column-layout** lays its children out in a column. It inherits the behavior from **grid-layout**. The *description* is a list of the layout's children, and the layout also translates the initargs *ratios*, *adjust*, *gap* and *uniform-size-p* into the equivalent **grid-layout** initargs *y-ratios*, *x-adjust*, *y-gap* and *y-uniform-size-p*.

description may also contain the keywords **:divider** and **:separator** which create a divider or separator as a child of the **column-layout**. The user can move a divider, but cannot move a separator.

When specifying **:ratios** in a row with **:divider** or **:separator**, you should use **nil** to specify that the divider or separator is given its minimum size, as in the example below.

Examples

```
(capi:contain (make-instance
               'capi:column-layout
               :description
                (list
```

```
                    (make-instance 'capi:push-button
                                   :text "Press me")
                   "Title"
                   (make-instance 'capi:list-panel
                                  :items '(1 2 3)))))


(setq column (capi:contain
              (make-instance
               'capi:column-layout
               :description
               (list
                (make-instance 'capi:push-button
                               :text "Press me")
                "Title:"
                (make-instance 'capi:list-panel
                               :items '(1 2 3)))
               :adjust :center)))

(capi:apply-in-pane-process
 column #'(setf capi:layout-x-adjust) :right column)

(capi:apply-in-pane-process
 column #'(setf capi:layout-x-adjust) :left column)

(capi:apply-in-pane-process
 column #'(setf capi:layout-x-adjust) :center column)

(flet ((make-list-panel (x y)
         (make-instance
          'capi:list-panel
          :items
          (loop for i below x
                collect i)
          :selection
          (loop for i below x by y
                collect i)
          :interaction
          :multiple-selection)))
  (capi:contain
   (make-instance
    'capi:column-layout
    :description
    (list
     (make-list-panel 100 5)
     :divider
     (make-list-panel 100 10))
    :ratios '(1 nil 2))))
```

See also

`row-layout`
1.2.1 CAPI elements
5.2 Button panel classes
6 Laying Out CAPI Panes
7 Programming with CAPI Windows
11 Defining Interface Classes - top level windows

component-name *Accessor*

Summary

Gets and sets the *component-name* of an **ole-control-pane**.

Package

capi

Signature

component-name *pane* **=>** *name*

(setf component-name) *name pane* **=>** *name*

Arguments

pane⇓	An **ole-control-pane**.
name	A string.

Values

name	A string.

Description

The accessor **component-name** accesses the *component-name* of *pane*.

When *pane* is created, it automatically opens the component and inserts it.

If **(setf component-name)** is called on a pane that is already created, any existing component is closed, and the new component is opened and inserted. **(setf component-name)** also sets the pane's *user-component* to **nil**.

Notes

component-name is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

ole-control-pane

confirmer-pane *Function*

Summary

Returns the pane associated with a confirmer interface.

Package

capi

Signature

`confirmer-pane` *interface* **=>** *pane*

Arguments

interface⇓ A confirmer interface displayed by **`popup-confirmer`**.

Values

pane The *pane* argument passed to **`popup-confirmer`**.

Description

The function **`confirmer-pane`** returns the pane associated with *interface*, which must have been displayed by **`popup-confirmer`**.

In most cases the programmer does not have access to this interface, but it can be passed to the confirmer's callbacks when extra buttons are added via the *buttons* argument.

See also

`popup-confirmer`

confirm-quit *Function*

Summary

Quits the Lisp session, potentially after user confirmation.

Package

`capi`

Signature

`confirm-quit` *application-name*

Arguments

application-name⇓ A string.

Description

The function **`confirm-quit`** calls **`quit`**, potentially after confirmation from the user.

The behavior of **`confirm-quit`** when called within LispWorks is determined by a LispWorks user preference, which can be set by **Tools > Preferences... > Environment > General > Confirm Before Exiting**. This preference can also be set programmatically (for example in an application) by **`set-confirm-quit-flag`**.

If the value of the flag is **`:check-editor-files`** (the default), **`confirm-quit`** checks whether there are editor buffers which are associated with files and are modified. If there is at least one such modified buffer, **`confirm-quit`** prompts the user to decide between three options:

Save Changes	Saves all modified buffers before quitting.
Discard Changes	Quits without saving.
Cancel	Does not save or quit.

If there are no such modified buffers, **`confirm-quit`** simply calls **`quit`**.

If the flag is **`nil`** then **`confirm-quit`** simply calls **`quit`**.

If the flag is **`t`** then **`confirm-quit`** prompts the user. If there are unsaved buffers, the prompt is as described above, otherwise the prompt is a simple yes/no confirmer dialog.

application-name is used in the prompt to identify the application.

Notes

The LispWorks IDE uses **`confirm-quit`**.

See also

`set-confirm-quit-flag`

confirm-yes-or-no *Function*

Summary

Pops up a dialog button containing a message and a **Yes** and **No** button.

Package

`capi`

Signature

`confirm-yes-or-no` *format-string* **`&rest`** *format-args* **`=>`** *result*

Arguments

format-string⇓	A string.
format-args⇓	Lisp objects.

Values

result	**`t`** or **`nil`**,.

Description

The function **`confirm-yes-or-no`** pops up a dialog box containing a message and the buttons **Yes** and **No**, returns **`t`** when the **Yes** button is clicked, and **`nil`** when the **No** button is clicked. The message is obtained by calling the Common Lisp function **`format`** with *format-string* and objects in *format-args*.

This function is actually a convenient version of **`prompt-for-confirmation`**, but has the disadvantage that you cannot specify any customization arguments. For more flexibility, use **`prompt-for-confirmation`** itself.

Examples

```
(setq pane (capi:contain
             (make-instance 'capi:text-input-pane)
                            :title "Test Interface"))


(when (capi:confirm-yes-or-no "Close ~S?" pane)
  (capi:apply-in-pane-process
   pane 'capi:quit-interface pane))
```

See also

prompt-for-confirmation
display-dialog
popup-confirmer
10 Dialogs: Prompting for Input

contain *Function*

Summary

Displays a window containing an element.

Package

capi

Signature

contain *element* **&rest** *interface-args* **&key** *screen process title as-dialog* **&allow-other-keys =>** *element*

Arguments

element⇓	A CAPI **element**.
interface-args⇓	A plist of keywords and values.
screen⇓	A **screen**, or any argument accepted by **convert-to-screen**.
process⇓	On GTK+, Microsoft Windows or Motif, a CAPI process, **t** or **nil**. On Cocoa, this argument is not supported.
title⇓	A string.
as-dialog⇓	A generalized boolean.

Values

element	A CAPI **element**.

Description

The function **contain** creates and displays a container for the CAPI element *element*. **contain** returns *element* as its result.

contain is provided as a convenient way of testing CAPI functionality and is useful mainly during interactive development. Many of the CAPI examples use it.

The container is created using **make-container**, which can make containers for any of the following classes:

simple-pane

layout

interface

pinboard-object

menu

menu-item

menu-component

cl:list

In the case of a **cl:list**, the CAPI tries to see what sort of objects they are and makes an appropriate container. For instance, if they were all **simple-pane**s it would put them into a **column-layout**.

interface-args, after removing the arguments *screen* and *process*, are passed to **make-container** as the initargs to the interface. *title* is used as the title of the container.

as-dialog can be **nil**, **t** or **:no-escape-button**. The default value of *as-dialog* is **nil**, which means display the interface as an ordinary window using **display**. When *as-dialog* is true it displays using **display-dialog**. When *as-dialog* is **t**, **contain** adds to the interface an escape button which invokes **abort-dialog**, to ensure that the user does not get stuck with a dialog that cannot be dismissed. When *as-dialog* is **:no-escape-button**, it does not add the escape button. Any value of *as-dialog* has the same effect as **t**.

The values of the arguments *screen* and *process* are passed to **display** when displaying the container.

Examples

```
(capi:contain (make-instance 'capi:text-input-pane))


(capi:contain (make-instance
               'capi:column-layout
               :description `("Title:"
                              ,(make-instance
                                'capi:text-input-pane))))


(capi:contain (make-instance 'capi:menu-item)
              :title "Test")
```

See also

make-container
display
display-dialog
element
2 Getting Started
4.1 The correct thread for CAPI operations
12 Creating Panes with Your Own Drawing and Input

convert-relative-position *Function*

Summary

Converts a screen position from one coordinate system to another.

Package

capi

Signature

convert-relative-position *from to x y* **=>** *to-x*, *to-y*

Arguments

from⇓	A pane, interface or screen.
to⇓	A pane, interface or screen.
x⇓	An integer.
y⇓	An integer.

Values

to-x	An integer.
to-y	An integer.

Description

The function **convert-relative-position** converts the position *x*,*y* in the coordinate system of *from* to that of *to*.

Examples

```
(example-edit-file "capi/elements/convert-relative-position")
```

See also

top-level-interface-geometry
with-geometry

convert-to-screen *Function*

Summary

Finds the appropriate screen or container for a CAPI object.

Package

capi

Signature

convert-to-screen &optional *object* **=>** *result*

Arguments

object⇓ A CAPI object, a plist, or keyword or **nil**.

Values

result⇓ A screen or a container.

Description

The function **convert-to-screen** finds the appropriate screen or container for the CAPI object *object*.

If *object* is **nil**, *result* is the default screen. *object* defaults to **nil**.

If *object* is a pane inside a MDI interface, then *result* is the **capi:container** of the interface, rather than the real screen, because this is more useful in most cases. To obtain the real screen, call **convert-to-screen** on the top level interface. See **document-frame** for a description of MDI interfaces.

object can be a keyword representing the CAPI library. This is equivalent to using the **:library** key in the plist case below. *object* can also be the special keyword **:if-any**, which finds a screen if there is any active screen, otherwise it returns **nil**.

object can be a plist. The keys below are supported on GTK+ and Motif. Other libraries ignore them.

:display The value is an X Window System display string describing the X display and screen to use. The default value is derived from the **DISPLAY** environment variable or (on Motif) the **-display** command-line option, or (on GTK+) the **--display** command-line option. If neither is supplied, the default is to use the default screen on the local host.

:host The name of the host to use for the X Window System display. This key is valid only if no **:display** key/value is supplied. The default value is the local host.

:server-number The number of the display server to use for the X Window System display. This key is valid only if no **:display** key/value is supplied. The default value is 0.

:screen-number The number of the screen to use for the X Window System display. This key is valid only if no **:display** key/value is supplied. The default value is the default screen of the display.

:application-class The value is a string naming the application class used for X Window System resources. The default value is "Lispworks". When running a delivered LispWorks image, you should specify the **:application-class** key if you want to provide application-specific resources.

On GTK+ the value is used for constructing the default *widget-name* for top-level interfaces. The application-class is prepended to the interface name followed by a ".", so if *application-class* is **"my-application"**, a top-level-interface of class **my-interface** will have a default *widget-name* **"my-application.my-interface"**.

See **element** for the description of *widget-name*.

Example GTK+ resource files are in **lib/8-0-0-0/examples/gtk/**.

:fallback-resources

On GTK+ the fallback resources are global, so they cannot be used to define different resources for different screens. Each call to **convert-to-screen** where *fallback-resources* is passed overrides the previous call. The value of *fallback-resources* is either a single string or a list of strings. In either case each string must be a complete specification according to the standard resource specification of GTK+ resource files (gtk_rc_parse_string should be able to parse it).

On Motif the value is a list of strings representing the set of application context fallback resources to use (see **XtAppSetFallbackResources**). Each string corresponds to a single line of an X resource file.

:library The value specifies the CAPI library. This is useful on Linux, FreeBSD and x86/x64 Solaris platforms, and in the macOS/GTK+ image, to choose between **:gtk** and **:motif** if the deprecated "capi-motif" module is loaded.

This keys is supported on Motif only. Other libraries ignore it.

:command-line-args The value is a list of strings representing the set of command-line arguments to pass to **XtOpenDisplay**. Each string corresponds to a single argument. The default value is derived from the command line used to start Lisp.

The resources are used only when no other system resource files can be found. When running a non-delivered LispWorks image, the default value of the **:fallback-resources** key is read from the file whose name is the value of the **:application-class** key in the **app-defaults** directory of the current LispWorks library. When running a delivered LispWorks image, you should specify the **:fallback-resources** key if your application needs fallback resources.

Examples

```
(capi:convert-to-screen)
```

See also

document-frame
screen
19 Host Window System-specific issues

count-collection-items *Generic Function*

Summary

Returns the number of items in a collection.

Package

capi

Signature

count-collection-items *collection* **&optional** *items*

Arguments

collection⇓ A **collection**

items⇓ A sequence.

Description

The generic function **count-collection-items** returns the number of items in *collection* by calling the *items-count-function*.

items defaults to **nil**. If it is non-nil, it is used instead of the *items* of *collection*.

Examples

The following example uses **count-collection-items** to return the number of items in a list panel.

```
(setq list (make-instance 'capi:list-panel
                          :items '(1 2 3 4 5)))


(capi:count-collection-items list)
```

The following example shows how to count the number of items in a specified list.

```
(capi:count-collection-items list '(1 2))
```

See also

collection
get-collection-item
search-for-item

create-dummy-graphics-port *Function*

Summary

Creates a graphics port object that can be used for querying fonts and measuring text or images.

Package

capi

Signature

create-dummy-graphics-port &optional *screen* **=>** *graphics-port*

Arguments

screen⇓ A value suitable as the argument to **convert-to-screen**.

Values

graphics-port⇓ A graphics port.

Description

The function **`create-dummy-graphics-port`** creates a graphics port object that can be used for font queries, measuring text and images.

graphics-port is a graphics port object associated with *screen*. *graphics-port* is never visible on the screen, but can be used to query fonts, measure text and load images to obtain their width and height. Drawing functions are not supported.

See also

`convert-to-screen`

current-dialog-handle *Function*

Summary

Returns the underlying handle of the current dialog.

Package

`capi`

Signature

`current-dialog-handle =>` *handle*

Values

handle	A platform-specific value, or **`nil`**.

Description

The function **`current-dialog-handle`** returns the underlying handle of the current dialog, as follows:

Microsoft Windows	The hwnd of the dialog.
GTK+	A pointer to the GdkWindow.
Motif	A windowid of the dialog.
Cocoa	The value returned by the NSWindow's **`windowNumber`** method.

This value is useful if you want to perform some operation on the underlying handle that the CAPI does not supply.

If there is no current dialog, **`current-dialog-handle`** returns **`nil`**.

Examples

Press on "Get handle" to see the handle of the dialog.

```
(capi:popup-confirmer
 (make-instance
  'capi:push-button
  :text "Get handle"
  :callback-type :none
  :selection-callback
```

```
      #'(lambda ()
          (capi:display-message
           (format nil "current-dialog-handle ~a~%"
                   (capi:current-dialog-handle)))))
     nil
     :title "A dialog")
```

See also

simple-pane-handle
18.7 Handles

current-document *Generic Function*

Summary

Returns the current document of a MDI interface.

Package

capi

Signature

current-document *mdi-interface* **=>** *child*

Arguments

mdi-interface⇓	An instance of a subclass of **document-frame**.

Values

child	The current document of *mdi-interface*.

Description

The generic function **current-document** returns the top child interface of *mdi-interface*.

See also

document-frame

current-pointer-position *Function*

Summary

Returns the current position of the pointer.

Package

capi

Signature

current-pointer-position &key *relative-to pane-relative-p* **=>** *x*, *y*

Arguments

relative-to⇓ A **screen** or a displayed **interface** or a CAPI pane.

pane-relative-p⇓ A boolean.

Values

x An integer.

y An integer.

Description

The function **current-pointer-position** returns the current x,y position of the pointer on the screen of *relative-to*, which defaults to the current screen.

If *pane-relative-p* is true then the position is returned relative to *relative-to*, otherwise it is returned relative to the screen. The default value of *pane-relative-p* is **t**.

See also

interface
screen

current-popup *Function*

Summary

Returns the current popup pane if there is one.

Package

capi

Signature

current-popup => *result*

Values

result A pane or **nil**.

Description

The function **current-popup** returns the current popup pane or **nil** if there is none. A current popup exists in the scope of callbacks which are done while a dialog is displayed on the screen in the current process.

If the dialog was raised by an explicit call to **display-dialog** or **popup-confirmer**, current-popup returns the first argument of **display-dialog** or **popup-confirmer**. For other functions that raise a dialog (such as the **prompt-for-file**, **prompt-for-confirmation** and so on), the result is CAPI pane created by the system.

See also

display-dialog
popup-confirmer

current-printer *Function*

Summary

Returns the currently selected printer object.

Package

capi

Signature

current-printer &key *interactive* **=>** *printer*

Arguments

interactive⇓ A boolean.

Values

printer⇓ A printer, or **nil**.

Description

The function **current-printer** returns the currently selected printer object for the default library.

If *interactive* is non-nil and there is no current printer, a confirmer is displayed warning the user and *printer* is **nil**. The default value of interactive is **nil**.

See also

page-setup-dialog
set-printer-options
16 Printing from the CAPI—the Hardcopy API

default-editor-pane-line-wrap-marker *Variable*

Summary

The default line wrap marker for editor panes.

Package

capi

Initial Value

`#\!`

Description

The variable **`*default-editor-pane-line-wrap-marker*`** provides the default value for the *line-wrap-marker* of an **`editor-pane`**. The value should be a **`character`** object, or **`nil`**.

See also

`editor-pane`

default-library *Function*

Summary

Returns the default library.

Package

`capi`

Signature

`default-library =>` *library*

Values

library	A library name.

Description

The function **`default-library`** returns a keyword naming the the default library.

On Linux, FreeBSD and x86/x64 Solaris platforms, the default library is **`:gtk`**. If you load the deprecated "capi-motif" module, then the library will be **`:motif`**.

On Microsoft Windows platforms, currently the only library available is **`:win32`**, hence this is the default library.

On macOS platforms, the only library available in the native GUI image is **`:cocoa`**, hence this is the default library. In the macOS/GTK+ image, the default library is **`:gtk`**, but you load the deprecated "capi-motif" module, then the library will be **`:motif`**.

See also

`installed-libraries`
19.5 CAPI communication with host window system - libraries

default-non-focus-message-timeout *Variable*

Summary

Specify the default *timeout* in **display-non-focus-message**.

Package

capi

Initial Value

2

Description

The variable ***default-non-focus-message-timeout*** specifies the default *timeout* in **display-non-focus-message**.

See **display-non-focus-message** for details.

See also

display-non-focus-message
default-non-focus-message-timeout-extension

default-non-focus-message-timeout-extension *Variable*

Summary

Specify the default *timeout-extension* in **display-non-focus-message**.

Package

capi

Initial Value

60

Description

The variable ***default-non-focus-message-timeout-extension*** specifies the default *timeout-extension* in **display-non-focus-message** respectively.

See **display-non-focus-message** for details.

See also

display-non-focus-message
default-non-focus-message-timeout

define-command *Macro*

Summary

Defines an alias for a mouse or keyboard gesture that can be used in the input model of an output pane.

Package

capi

Signature

define-command *name gesture* **&key** *translator host library*

Arguments

name⇓ A unique Lisp object.

gesture⇓ A valid input model gesture.

translator⇓ A function.

host⇓ Alias for library, for backwards compatibility.

library⇓ Specifies for which library this mapping is applicable. See <new section above about libraries> for which libraries are applicable. By default the mapping is applicable to all libraries.

Description

The macro **define-command** defines an alias for an input gesture that can then be used in the input model of an **output-pane**.

name is the name of the alias, which should be a symbol.

gesture is one of the gestures accepted by **output-pane**. For a full description of the gesture syntax and arguments for the callback, see **12.2.1 Detailed description of the input model**. It is possible to specify multiple gestures by passing as *gesture* a list of the form:

```
(:one-off gesture1 gesture2 ...)
```

If *translator* is supplied it needs to be a function that takes the same arguments that a callback for the gesture would take (not including the *extra-callback-args*), and returns a list which is used after *pane* instead of the gesture callback arguments. When there is a *translator*, the callbacks for commands in the models are invoked by:

```
(apply callback pane
       (append (apply translator gesture-callback-args)
               extra-callback-args))
```

library specifies which library this mapping is applicable to. It is possible to have distinct definitions for different libraries, but redefinition with the same library overrides the previous definition. The default value of *library* is **nil**, which means all libraries. *host* is recognised an alias *library* for backwards compatibility.

Examples

Firstly, here is an example of defining a command which maps onto a gesture.

```
(defun gesture-callback (output-pane x y)
  (capi:display-message
   "Pressed ~S at (~S,~S)"
   output-pane x y))

(capi:define-command :select (:button-1 :press))

(capi:contain (make-instance
               'capi:output-pane
               :input-model '((:select
                               gesture-callback))))
```

Here is a more complicated example demonstrating the use of *translator* to affect the arguments passed to a callback.

```
(capi:define-command
  :select-object (:button-1 :press)
  :translator #'(lambda (output-pane x y)
                  (let ((object
                        (capi:pinboard-object-at-position
                                   output-pane x y)))
                            (when object
                              (list object)))))


(defun object-select-callback (output-pane
                               &optional object)
  (when object (capi:display-message
                "Pressed on ~S in ~S"
                object output-pane)))


(setq pinboard
      (capi:contain (make-instance
                     'capi:pinboard-layout
                     :input-model '((:select-object
                                     object-select-callback)))))


(make-instance 'capi:item-pinboard-object
               :text "Press Me!"
               :parent pinboard
               :x 10 :y 20)


(make-instance 'capi:line-pinboard-object
               :parent pinboard
               :start-x 20 :start-y 50
               :end-x 120 :end-y 150)
```

Here is a further example:

```
(example-edit-file "capi/output-panes/commands")
```

See also

output-pane
invoke-command
invoke-untranslated-command

12.2.2 Commands - aliases

define-interface *Macro*

Summary

Defines subclasses of **interface**.

Package

capi

Signature

define-interface *name superclasses slots* **&rest** *options*

Arguments

name⇓	A symbol.
superclasses⇓	A list of symbols naming classes.
slots⇓	A list of slot specifiers as in **defclass**.
options⇓	Class options as in **defclass**, plus specific options (see below).

Description

The macro **define-interface** is used to define subclasses of **interface**, which when created with **make-instance** has the specified panes, layouts and menus created automatically. *slots* and *superclasses* are used to describe the slots and superclasses of *name* as in the **defclass** macro, except that if *superclasses* is non-nil it must include **interface** or a subclass of it.

define-interface accepts the same class options in *options* as **defclass**, plus the following extra options:

:panes	Descriptions of the interface's panes.
:layouts	Descriptions of the interface's layouts.
:menus	Descriptions of the interface's menus.
:menu-bar	A list of menus for the interface's menu bar.
:definition	Options to alter **define-interface**.

The class options **:panes**, **:layouts** and **:menus** add extra slots to the class that will contain the CAPI object described in their description. Within the scope of the extra options, the slots themselves are available by referencing the name of the slot, and the interface itself is available with the variable **interface**. Each of the slots can be made to have readers, writers, accessors or documentation by passing the appropriate **defclass** keyword as one of the optional arguments in the description. Therefore, if you need to find a pane within an interface instance, you can provide an accessor, or simply use **with-slots**.

The option **:panes** is a list of pane descriptions of the following form:

```
(:panes
  (slot-name pane-class initargs)
  …
```

```
   (slot-name  pane-class  initargs)
)
```

where *slot-name* is a name for the slot, *pane-class* is the class of the pane being included in the interface, and *initargs* are the initialization arguments for the pane - the allowed forms are described below.

The option **:layouts** is a list of layout descriptions of the following form:

```
(:layouts
   (slot-name  layout-class  children  initargs)
   …
   (slot-name  layout-class  children  initargs)
)
```

where *slot-name* is a name for the slot, *layout-class* specifies the type of layout, *children* is a list of children for the layout, and *initargs* are the initialization arguments for the layout - the allowed forms are described below. The primary layout for the interface defaults to the first layout described, but can be specified as the **:layout** initarg to the interface. If no layouts are specified, then the CAPI will place all of the defined panes into a column layout and make that the primary layout.

The option **:menus** is a list of menu and menu component descriptions of the following form:

```
(:menus
   (slot-name  title  descriptions  initargs)
   …
   (slot-name  title  descriptions  initargs)
)
```

slot-name is the slot name for each menu or menu component.

title is the menu's title, the keyword **:menu**, or the keyword **:component**. For an example showing how you can specify mnemonics for menu titles, see **8.6 Mnemonics in menus**.

descriptions is a list of menu item descriptions. Each menu item description is either a title, a slot name for a menu, or a list of items containing a title, descriptions, and a list of initialization arguments for the menu item. *descriptions* should **nil** if you specify the **:items-function** initarg.

initargs are the initialization arguments for the menu.

The values given in *initargs* under **:panes**, **:layouts** and **:menus** can be lists of the form:

```
(:initarg keyword-name)
(:initarg key-spec)
(:initarg key-spec  initarg-value)

key-spec  := var
           | (var)
           | (var  initform)
           | ((keyword-name  var))
           | ((keyword-name  var)  initform)

keyword-name  := any keyword
```

key-spec is interpreted as in the **&key** symbol of ordinary Common Lisp lambda lists. When this form of value is used, the specified *keyword-name* is added as an extra initarg to the class defined by the **define-interface** form.

If *key-spec* is followed by *initarg-value*, then its value is used as the initarg of the pane. Otherwise the value from *key-spec* is used.

Additionally *initargs* may contain the keyword argument **:make-instance-extra-apply-args** which is useful when you want to supply initargs to the pane *slot-name* when the interface is initialized. The value *make-instance-extra-apply-args* should be a keyword which becomes an extra initarg to the interface class *name*. The value of that initarg should be a list of pane initargs and values which is passed when the pane is initialized. For an example, see:

```
(example-edit-file "capi/applications/argument-passing")
```

The option **:menu-bar** is a list of slot names, where each slot referred to contains a menu that should appear on the menu bar.

The option **:definition** is a property list of arguments which **define-interface** uses to change the way that it behaves. Currently there is only one definition option:

:interface-variable

Allows you to specify the name of a variable which (lexically within the **define-interface** form) refers to the interface instance. By default this variable is **interface**. See the example below.

Examples

Firstly, a couple of pane examples:

```
(capi:define-interface test1 ()
  ()
  (:panes
    (text capi:text-input-pane))
  (:default-initargs :title "Test1"))

(capi:display (make-instance 'test1))

(capi:define-interface test2 ()
  ()
  (:panes
    (text capi:text-input-pane)
    (buttons capi:button-panel :items '(1 2 3)
             :reader test2-buttons))
  (:layouts
    (main-layout capi:column-layout '(text buttons)))
  (:default-initargs :title "Test2"))

(test2-buttons
 (capi:display (make-instance 'test2)))
```

Here are a couple of menu examples:

```
(capi:define-interface test3 ()
  ()
  (:menus
    (color-menu "Colors" (:red :green :blue)
       :print-function 'string-capitalize))
  (:menu-bar color-menu)
  (:default-initargs :title "Test3"))


(capi:display (make-instance 'test3))


(capi:define-interface test4 ()
  ()
  (:menus
   (colors-menu "Colors"
                ((:component
```

```
                      (:red :green :blue)
                      :interaction :single-selection
                      :print-function
                      'string-capitalize)
                     more-colors-menu))
      (more-colors-menu "More Colors"
                         (:pink :yellow :cyan)
                         :print-function
                         'string-capitalize))
     (:menu-bar colors-menu)
     (:default-initargs :title "Test4"))
```

```
(capi:display (make-instance 'test4))
```

This example demonstrates inheritance amongst subclasses of **interface**:

```
(capi:define-interface test5 (test4 test1)
  ()
  (:default-initargs :title "Test5"))
```

```
(capi:display (make-instance 'test5))
```

The next three examples illustrate the use of **:initarg** in initarg specifications for **:panes**.

Here we initialize the **:selected-items** initarg of the pane **foo** to the value passed by **:select** when making the interface object, or **nil** otherwise:

```
(capi:define-interface init1 () ()
  (:panes
   (foo
    capi:list-panel
    :items '(0 1 2 3 4)
    :visible-min-height '(:character 5)
    :interaction :multiple-selection
    :selected-items (:initarg select))))

(capi:contain (make-instance 'init1
                             :select '(1 3)))

(capi:contain (make-instance 'init1))
```

Here we initialize the **:selected-items** initarg of pane **foo** to the value passed by **:select** initarg when making the interface object, or **(1 3)** otherwise:

```
(capi:define-interface init2 () ()
  (:panes
   (foo
    capi:list-panel
    :items '(0 1 2 3 4)
    :visible-min-height '(:character 5)
    :interaction :multiple-selection
    :selected-items
    (:initarg (select '(1 3))))))

(capi:contain (make-instance 'init2))
```

Here we increment the indices passed in the interface's **:select** initarg before passing them in the **:selected-items** initarg of pane **foo**:

```
(capi:define-interface init3 () ()
```

```
    (:panes
     (foo
      capi:list-panel
      :items '(0 1 2 3 4)
      :visible-min-height '(:character 5)
      :interaction :multiple-selection
      :selected-items
      (:initarg select
       (mapcar '1+ select)))))

(capi:contain (make-instance 'init3
                             :select '(1 3)))
```

This example illustrates the use of **:interface-variable**. Both menu commands act on the interface itself, but they receive this argument in different ways:

```
(capi:define-interface foo () ()
  (:menus
   (menu "Run"
           (("Interface Variable"
             :callback (lambda () (test xxx))
             :callback-type :none)
           (":callback-type :interface"
            :callback 'test
            :callback-type :interface))))
  (:menu-bar menu)
  (:definition :interface-variable xxx))

(defmethod test ((foo foo))
  (capi:display-message "foo"))

(capi:display (make-instance 'foo))
```

There are many more examples in the LispWorks installation directory under **examples/capi/**.

See also

interface
layout
menu
8 Creating Menus
11 Defining Interface Classes - top level windows

define-layout *Macro*

Summary

Defines new classes of **layout**.

Package

capi

Signature

define-layout *name superclasses slots* **&rest** *options*

Arguments

name⇓ A symbol.

superclasses⇓ A list of symbols naming classes.

slots⇓ A list of slot specifiers as in **defclass**.

options⇓ Class options as in **defclass**.

Description

The macro **define-layout** is used to create new classes of **layout**. The macro is essentially the same as **defclass** except that its default superclass is **layout**. See **defclass** for a description of *name*, *superclasses*, *slots* and *options*.

To implement a new class of **layout**, methods need to be provided for the following generic functions:

interpret-description Translate the layout's child descriptions.

calculate-constraints Calculate the constraints for the layout.

calculate-layout Layout the children of the layout.

See also

interpret-description
calculate-constraints
calculate-layout
layout

define-menu *Macro*

Summary

Defines a menu function.

Package

capi

Signature

define-menu *function-name* **(***self***)** *title* *descriptions* **&rest** *initargs*

Arguments

function-name⇓ A symbol.

self⇓ A symbol.

title⇓ A string.

descriptions⇓ Lisp forms describing menu items.

initargs⇓ Keywords and values.

Description

The macro **define-menu** defines a function called *function-name* with a single argument *self* that will make a menu from *title*, *descriptions* and *initargs*, which take the same form as the **:menus** section of **define-interface**.

Examples

```
(capi:define-menu make-test-menu (self)
   "Test"
   ("Item1"
    "Item2"
    (:component
     ("Item3"
      "Item4")
     :interaction :single-selection)
    (:menu
     ("Item5"
      "Item6")
     :title "More Items")))

(setq interface (make-instance 'capi:interface))

(setf (capi:interface-menu-bar-items interface)
      (list (make-test-menu interface)))

(capi:display interface)
```

See also

define-interface
menu
undefine-menu

define-ole-control-component *Macro*

Summary

Defines a class that implements the OLE Control protocol for a CAPI pane.

Package

capi

Signature

define-ole-control-component *class-name* **(***superclass-name****)** *slots* **&rest** *class-options*

Arguments

class-name⇓	A symbol.
superclass-name⇓	A symbol naming a class.
slots⇓	A list of slot specifiers as in **defclass**.
class-options⇓	Class options as in **defclass**, plus specific options (see below).

Description

The macro **define-ole-control-component** defines an Automation component class *class-name* (like **com:define-automation-component**) that also implements the OLE Control protocols and other named interfaces or a coclass. This allows a CAPI pane to be embedded in an OLE Control container implemented outside LispWorks.

Each *superclass-name* argument specifies a direct superclass of the new class, which can be any **standard-class** provided that certain standard classes are included somewhere in the overall class precedence list. These standard classes depend on the other options and provide the default superclass list if none is specified. The following standard classes are available:

ole-control-component is always needed and provides an implementation of the OLE Control protocol.

com:standard-i-dispatch is always needed and provides a complete implementation of the i-dispatch interface, based on the type information in a type library.

com:standard-i-connection-point-container is needed if there are any source interfaces specified (via the **:coclass** or **:source-interfaces** options). This provides a complete implementation of the Connection Point protocols, used to support events.

slots is a list of standard **defclass** slot definitions.

class-options are standard **defclass** options. In addition the following options are recognized:

> **(:coclass** *coclass-name***)**
>
> **(:interfaces** *interface-name****)**
>
> **(:source-interfaces** *interface-name****)**

See **com:define-automation-component** in the *COM/Automation User Guide and Reference Manual* for details of these options.

Typically the **:pane-function** and **:create-callback** initargs are supplied using the **:default-initarg** option.

Implementations of the methods in the **:coclass** and **:interfaces** options should be defined using **com:define-com-method**, **com:define-dispinterface-method** or **com:com-object-dispinterface-invoke**.

Notes

define-ole-control-component is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

Examples

```
(example-edit-file "com/ole/control-implementation/deliver.lisp")
```

See also

ole-control-component

destroy *Generic Function*

Summary

Closes a window and calls the *destroy-callback*.

Package

capi

Signature

destroy *interface*

Arguments

interface⇓ An **interface**.

Description

The generic function **destroy** closes the window associated with *interface*, and then calls the interface's *destroy-callback* if it has one.

There is a complementary function **quit-interface** which calls the interface's *confirm-destroy-function* to confirm that the destroy should be done, and it is advisable to always use this unless you want to make sure that the interface's *confirm-destroy-function* is ignored.

Notes

destroy must only be called in the process of *interface*. Menu callbacks on *interface* will be called in that process, but otherwise you probably need to use **execute-with-interface** or **apply-in-pane-process**.

Examples

```
(setq interface
      (capi:display (make-instance
                     'capi:interface
                     :title "Test Interface"
                     :destroy-callback
                       #'(lambda (interface)
                           (capi:display-message
                             "Quitting ~S"
                              interface)))))
```

```
(capi:apply-in-pane-process
 interface 'capi:destroy interface)
```

See also

interface
quit-interface
update-screen-interfaces-hooks
7 Programming with CAPI Windows

destroy-dependent-object *Generic Function*

Summary

A mechanism to destroy objects when a **pinboard-layout** is destroyed.

Package

capi

Signature

destroy-dependent-object *object*

Method signatures

destroy-dependent-object (*object* **cons)**

destroy-dependent-object (*object* **process)**

Arguments

object⇓	A Lisp object.

Description

The generic function **destroy-dependent-object** is part of a mechanism for destroying objects when a **pinboard-layout** is destroyed.

Objects may be registered for destruction by calling **record-dependent-object** and unregistered by calling **unrecord-dependent-object**.

The predefined **destroy-dependent-object** method specializing on **cl:cons** expects a list where the car is a function and the cdr are its arguments. It applies the function to the arguments. The predefined method specializing on **mp:process** calls **mp:process-terminate** on the process *object*.

See also

pinboard-layout
record-dependent-object
unrecord-dependent-object

detach-simple-sink *Function*

Summary

Detaches a previously-attached simple sink object.

Package

capi

Signature

detach-simple-sink *sink pane*

Arguments

sink⇓	A class instance.
pane⇓	An **ole-control-pane**.

Description

The function **detach-simple-sink** detaches a sink that was previously attached to the active component in the **ole-control-pane** *pane* by a call to **attach-simple-sink**.

sink is the value returned by **attach-simple-sink** when the sink was attached.

pane is an **ole-control-pane** which is the pane where the component is.

Attached sinks are automatically disconnected when the object is closed.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

attach-simple-sink
ole-control-pane

detach-sink *Function*

Summary

Detaches a previously-attached sink.

Package

capi

Signature

detach-sink *sink pane interface-name*

Arguments

sink⇓	A class instance.
pane⇓	An **ole-control-pane**.
interface-name⇓	A refguid or the symbol **:default**.

Description

The function **detach-sink** detaches a sink which was previously attached to the active component in the

ole-control-pane *pane*.

sink is an instance of a class that implements the interface *interface-name*.

pane is an **ole-control-pane** which is the pane where the component is.

interface-name is either a string naming a source interface that the component in *pane* supports or **:default** to disconnect from the default source interface.

Attached sinks are automatically disconnected when the object is closed.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

attach-simple-sink
attach-sink
ole-control-pane

display *Function*

Summary

Displays a CAPI interface on a specified screen.

Package

capi

Signature

display *interface* **&key** *screen owner window-styles process* **=>** *interface*

Arguments

interface⇓	A CAPI interface.
screen⇓	A screen, or any argument accepted by **convert-to-screen**.
owner⇓	A CAPI interface.
window-styles⇓	A list of keywords.
process⇓	On GTK+, Microsoft Windows or Motif, a CAPI process, **t** or **nil**. On Cocoa, this argument is not supported.

Values

interface	A CAPI interface.

Description

The function **display** displays the CAPI interface *interface* on the specified *screen* (or the current one if not supplied).

If *process* is not supplied, then if *owner* is supplied *interface* runs in *owner*'s process, otherwise interface runs in the process of the parent of *interface* if it is a **document-container**, or in a new process created for *interface* if not.

On Microsoft Windows and Motif, if *process* is **t**, then *interface* runs in a newly-created process. If process is **nil**, *interface* runs in the current process. Otherwise *process* is expected to be a CAPI process, and *interface* runs in it. A CAPI process is a **mp:process** which was created by calling **display**. You can pass only a CAPI process as *process*, because it needs to handle messages using the LispWorks event loop. The default value of *process* is **t**.

On Cocoa, all CAPI interfaces run in the Cocoa Event Loop process (which is the main thread of LispWorks) and therefore *process* is not supported. If *process* is any process other than the Cocoa Event Loop process an error is signalled.

owner specifies an owner for *interface*, which should be another CAPI interface. *interface* inherits a number of attributes from *owner*, including the default process, default screen and default display state.

window-styles, if supplied, sets the *window-styles* slot of *interface*. See **interface** for information about *window-styles*.

display returns its *interface* argument.

Notes

1. Use the function **contain** to display objects other than interfaces.
2. Once **display** has finished preparing the interface to display, it calls **interface-display** to actually do the display. The primary method does the actual display, and you can **:before** or **:after** methods to execute code just before or just after the window appears.

Examples

```
(capi:display (make-instance 'capi:interface
                             :title "Test"))
```

See also

contain
convert-to-screen
display-dialog
document-container
execute-with-interface
interface
interface-display
quit-interface
update-screen-interfaces-hooks
2 Getting Started
4.1 The correct thread for CAPI operations
19 Host Window System-specific issues
7 Programming with CAPI Windows
10.4 Dialog Owners

display-dialog *Function*

Summary

Displays a CAPI interface as a dialog box.

Package

`capi`

Signature

`display-dialog` *interface* **`&key`** *screen focus modal timeout owner x y position-relative-to continuation callback-error-handler* **`=>`** *result*, *okp*

Arguments

interface⇓ A CAPI interface.

screen⇓ A screen.

focus⇓ A pane of *interface*.

modal⇓ **`t`**, **`:dismiss-on-input`** or **`nil`**.

timeout⇓ **`nil`** or a real number.

owner⇓ A pane.

x⇓, *y*⇓ Real numbers representing coordinates, or keywords or lists specifying an adjusted position.

position-relative-to⇓ **`:owner`** or **`nil`**.

continuation⇓ A function or **`nil`**.

callback-error-handler⇓

A function designator or **`nil`**.

Values

result⇓ An object.

okp A boolean.

Description

The function **`display-dialog`** displays the CAPI **`interface`** *interface* as a dialog box.

screen is the **`screen`** for the dialog to be displayed on.

focus should be the pane within the interface that should be given the focus initially. If a focus is not supplied, then it lets the window system decide.

A true value of *modal* indicates that the dialog takes over all input to the application. Additionally, if *modal* is **`:dismiss-on-input`** then any user gesture (a button or key press) causes the dialog to disappear. **`:dismiss-on-input`** works on platforms other than Motif. The default value of *modal* is **`t`**.

owner specifies an owner window for the dialog. See **10.4 Dialog Owners** for details.

If *timeout* is non-nil, the dialog automatically aborts if it is still displayed after *timeout* seconds.

If *x* and *y* are numbers they specify the coordinates of the dialog. Alternatively *x* and *y* can be keywords like **`:left`** and **`:top`**, or lists like **`(:left 100)`**, **`(:bottom 50)`** and so on.. These values cause the dialog to be positioned relative to its owner in the same way as the *adjust* argument to **`pane-adjusted-position`**. The default location is at the center of the dialog's owner.

position-relative-to has a default value **`:owner`**, meaning that *x* and *y* are relative to dialog's owner. The value **`nil`** means that *x* and *y* are relative to the screen.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **`display-dialog`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`display-dialog`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

The values returned depend on how the dialog is dismissed. Typically a user gesture will trigger a call to **`abort-dialog`**, causing the values **`nil`**, **`nil`** to be returned or to **`exit-dialog`** causing the values *result*, **`t`** to be returned, where *result* is the argument to **`exit-dialog`**. If *continuation* is non-nil, then the returned values are always **`:continuation`**, **`nil`**.

The CAPI also provides **`popup-confirmer`** which gives you the standard **OK** and **Cancel** button functionality.

callback-error-handler allows error handling in callbacks which is uniform across platforms, as described for **`popup-confirmer`**.

Notes

1. If you need to replace one dialog with another, you can use **`display-replacable-dialog`** and **`replace-dialog`**.
2. In a modal dialog at least one button which aborts or exits the dialog must be provided in *interface*. This is the programmer's responsibility, as without such a button there is no way to clear the modal dialog. A straightforward way to add these buttons is to display the window via **`popup-confirmer`** which adds the buttons for you.

Examples

```
(capi:display-dialog
 (capi:make-container
  (make-instance 'capi:push-button-panel
                 :items '("OK" "Cancel")
                 :callback-type :data
                   :callbacks '(capi:exit-dialog
                                  capi:abort-dialog))
  :title "Empty Dialog"))
```

There are further examples:

```
(example-edit-file "capi/dialogs/")
```

See also

`abort-dialog`
`display`
`display-replacable-dialog`
`exit-dialog`
`interface`
`popup-confirmer`
`with-dialog-results`
`*update-screen-interfaces-hooks*`
10 Dialogs: Prompting for Input

display-errors *Macro*

Summary

Displays a message if an error is signalled.

Package

`capi`

Signature

`display-errors &body` *body*

Arguments

body⇓ Lisp forms.

Description

The macro **`display-errors`** executes the forms in *body* inside a **`handler-case`** form. If an error is signalled inside *body*, a message is displayed and the debugger is not entered.

display-message *Function*

Summary

Displays a message on the current CAPI screen.

Package

`capi`

Signature

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

Arguments

format-string⇓ A string.

format-args⇓ Lisp objects.

Description

The function **`display-message`** creates a message from *format-string* and *format-args* using **`format`**, and then displays it on the current CAPI screen.

Notes

If you need to make a window-modal sheet on Cocoa, then use the function **prompt-with-message**.

Examples

```
(capi:display-message "Current screen = ~S"
                      (capi:convert-to-screen))
```

See also

prompt-with-message
display-message-for-pane
display-non-focus-message
display-dialog
2 Getting Started
10 Dialogs: Prompting for Input

display-message-for-pane *Function*

Summary

Displays a message on the same screen as a specified pane.

Package

capi

Signature

display-message-for-pane *pane* *format-string* **&rest** *format-args*

Arguments

pane⇓	A **simple-pane**.
format-string⇓	A string.
format-args⇓	Lisp objects.

Description

The function **display-message-for-pane** creates a message from the arguments *format-string* and *format-args* using **format**, and then displays it on the same screen as *pane*.

Notes

If you need to make a window-modal sheet on Cocoa, then use the function **prompt-with-message**.

Compatibility note

The function **display-message-on-screen** is retained for compatibility with previous versions of LispWorks. It is a synonym for **display-message-for-pane**.

Examples

```
(setq pane (capi:contain (make-instance
                          'capi:text-input-pane)))


(capi:display-message-for-pane pane
                               "Just created ~S" pane)
```

See also

prompt-with-message
display-message

display-non-focus-message *Function*

Summary

Display a message in a non-focus window for a short period of time.

Package

capi

Signature

display-non-focus-message *string* **&key** *timeout timeout-extension owner x alternative-x right y alternative-y bottom alternative-right alternative-bottom transparency background font widget-name*

Arguments

string⇓ A string or a list of strings.

timeout⇓ A positive integer.

timeout-extension⇓ A positive integer.

owner⇓ A visible CAPI pane.

x⇓, *alternative-x*⇓, *right*⇓

Integers, or one of the keywords **:left**, **:right**, **:center** and **:centre**.

y⇓, *alternative-y*⇓, *bottom*⇓

Integers, or one of the keywords **:top**, **:bottom**, **:center** and **:centre**.

alternative-right⇓ An integer, or one of the keywords **:left**, **:right**, **:center** and **:centre**, or **t**.

alternative-bottom⇓ An integer, or one of the keywords **:top**, **:bottom**, **:center** and **:centre**, or **t**.

transparency⇓ A real number in the inclusive range [0,1].

background⇓ A color in the Graphics Ports color system.

font⇓ A **font** or a **font-description**, or a positive integer.

widget-name⇓ A string designator.

Description

The function **`display-non-focus-message`** displays a message in a non-focus window for a short period of time, to notify the user of something that does not actually require their attention.

string is the message. It should be either a string, or a list of strings, which are concatenated with newlines to give the actual text to display. **`#\Newline`** characters in *string* break lines as expected.

timeout, if supplied, should be a positive integer. It specifies the time in seconds before the window displaying the message disappears. The default value of *timeout* is **`*default-non-focus-message-timeout*`**.

timeout-extension is used when the user tries to copy the message text. The default value of *timeout-extension* is **`*default-non-focus-message-timeout-extension*`**. See **"Copying from the message"** below for discussion.

owner should be a visible CAPI pane. The positioning of the non-focus window is with respect to *owner*.

x, *y*, *right*, *bottom*, *alternative-x*, *alternative-y*, *alternative-right*, and *alternative-bottom* are used for positioning the window. *x*, *alternative-right*, *alternative-x* and *right* are the horizontal keywords, and one of them determines the horizontal position as described below. *y*, *alternative-bottom*, *alternative-y* and *bottom* are the vertical keywords, and one of them determines the vertical position. The values **`:center`** and **`:centre`** are synonyms here.

x and *y* specify the positioning of the left and top sides of the window, except for **`:center`**/**`:centre`**. An integer means the offset in pixels from the left or top of *owner*. **`:left`**, **`:right`**, **`:top`** and **`:bottom`** mean the left/right/top/bottom of *owner*. **`:center`** means the center of *owner*, and in this case it specifies the location of the center of the window in the x or y dimension. The default value of both *x* and *y* is **`:center`**.

right and *bottom* override *x* and *y* respectively. They specify the positioning of the right or bottom of the window, except for **`:center`**/**`:centre`**, where they are interpreted in the same way as *x* and *y*.

alternative-x, *alternative-y*, *alternative-right*, and *alternative-bottom* are used if positioning the window using *x* or *right* and *y* or *bottom* would place it outside of the screen, and are interpreted the same way as the non-alternative keywords. The decision to use the alternative variables is made independently in the horizontal and vertical directions. *alternative-right* and *alternative-bottom* can both take the special value **`t`**, meaning the screen width and height.

transparency specifies the transparency of the window. See **`interface`** for details.

background specifies the background color of the window.

font specifies the font to use. If it is a positive integer it specifies the font size, that is equivalent to:

```
(gp:make-font-description :size font)
```

widget-name specifies the *widget-name* of the **`interface`** that displays the window. See **`element`** for details.

Copying from the message

The user can select part of the message with the mouse, and then copy it using the context menu (raised by right-click). Whenever the user changes the selection or cursor position, a timout specified by *timeout* is re-scheduled with *timeout-extension* seconds, so the window does not disappear while the user tries to copy.

The context menu also has a **Close** item, so the user can explicitly close the window once she has finished.

Notes

Because **`display-non-focus-message`** raises a window that does not take the focus, it does not interfere with what the user is already doing (except when the user clicks on the window). It is therefore useful to notify the user about events that do not actually require the user to stop what they are doing and do something, for example when a saving operation is complete.

See also

display-message
default-non-focus-message-timeout
default-non-focus-message-timeout-extension

display-pane *Class*

Summary

The class **display-pane** is a pane that displays multiple lines of text.

Package

capi

Superclasses

titled-object
simple-pane

Initargs

:text	A string or a list of strings to be displayed.

Accessors

display-pane-text

Description

The *text* passed to a display pane can be provided either as a single string containing newlines, or else as a list of strings where each string represents a line.

Examples

```
(capi:contain (make-instance
               'capi:display-pane
               :text
                 '("One" "Line" "At" "A" "Time...")))

(setq dp (capi:contain
          (make-instance
           'capi:display-pane
           :text
           '("One" "Line" "At" "A" "Time...")
           :visible-min-height
           '(:character 5))))

(capi:apply-in-pane-process
 dp #'(setf capi:display-pane-text)
 '("Some" "New" "Text") dp)
```

See also

display-pane-selected-text
display-pane-selection
display-pane-selection-p
editor-pane
set-display-pane-selection
text-input-pane
title-pane
3.5 Displaying and entering text

display-pane-selected-text *Function*

Summary

Returns the selected text in a **display-pane**.

Package

capi

Signature

display-pane-selected-text *display-pane* **=>** *result*

Arguments

display-pane⇓ An instance of **display-pane** or a subclass.

Values

result A string or **nil**.

Description

The function **display-pane-selected-text** returns the selected text in *display-pane*, or **nil** if there is no selection.

See also

display-pane
display-pane-selection-p
display-pane-selection

display-pane-selection *Function*

Summary

Returns the bounds of the selection in a **display-pane**.

Package

capi

Signature

display-pane-selection *pane* **=>** *start*, *end*

Arguments

pane⇓ A **display-pane**.

Values

start⇓, *end*⇓ Non-negative integers.

Description

The function **display-pane-selection** returns as multiple values the bounding indexes of the selection in *pane*. That is, *start* is the inclusive index of the first selected character, and *end* is one greater than the index of the last selected character.

If there is no selection, then both *start* and *end* are the caret position in *pane*.

See also

set-display-pane-selection
display-pane
display-pane-selected-text
display-pane-selection-p

display-pane-selection-p *Function*

Summary

Returns true if there is selected text in a **display-pane**.

Package

capi

Signature

display-pane-selection-p *pane* **=>** *selectionp*

Arguments

pane⇓ A **display-pane**.

Values

selectionp A boolean.

Description

The function **display-pane-selection-p** returns **t** if there is a selected region in *pane* and **nil** otherwise.

See also

set-display-pane-selection
display-pane
display-pane-selected-text
display-pane-selection

display-popup-menu *Function*

Summary

Displays a popup menu.

Package

capi

Signature

display-popup-menu *menu* **&key** *owner* *x* *y* *button* **=>** *result*

Arguments

menu⇓	A menu.
owner⇓	A pane.
x⇓	The horizontal coordinate of *menu*'s position relative to *owner*.
y⇓	The vertical coordinate of *menu*'s position relative to *owner*.
button⇓	The mouse button that raises the menu.

Values

result	**t** or **nil**.

Description

The function **display-popup-menu** displays the **menu** *menu* at position *x*,*y*. **display-popup-menu** should be used in response to the user clicking a mouse button, and is typically used to implement context ("right button") menus.

The user may select an item in the menu, in which case the item's *selection-callback* is invoked, and **display-popup-menu** returns **t**.

Alternatively the user may cancel the menu, by clicking elsewhere or pressing the **Escape** key. In this case, **display-popup-menu** returns **nil**.

owner specifies the owner of the menu, that is, a pane that the menu is associated with. If *owner* is not supplied the system tries to find the appropriate owner, which usually suffices.

x and *y* default to the horizontal and vertical coordinates, relative to *owner*, of the location of the mouse pointer.

button defaults to **:button-3**.

Examples

See **8.13 Displaying menus programmatically**.

See also

menu
pinboard-layout
popup-menu-force-popdown
8.13 Displaying menus programmatically

display-replacable-dialog *Function*

Summary

Displays a replacable dialog.

Package

capi

Signature

display-replacable-dialog *interface* **&rest** *args* **=>** *result*

Arguments

interface⇓	An interface.
args⇓	Other arguments as for **display-dialog**.

Values

result	The value returned by the dialog.

Description

The function **display-replacable-dialog** displays a dialog that can be replaced by another dialog.

interface is a CAPI interface to be displayed as a dialog.

The arguments *args* are interpreted the same as the arguments to **display-dialog**, except that *modal* is ignored. **display-replacable-dialog** displays the dialog like **display-dialog**.

Within the scope of **display-replacable-dialog** (that is, inside the callbacks) the programmer can call **replace-dialog** which replaces the dialog by a new dialog and destroys the existing one. There can be many calls to **replace-dialog** inside the same scope of **display-replacable-dialog**.

display-replacable-dialog returns the last dialog that was displayed.

Inside **display-replacable-dialog**, the functions that use the current dialog, such as **exit-dialog** and **abort-dialog**, work in the same way that they work inside **display-dialog**, except that they do not affect the return value of **display-replacable-dialog**.

See also

abort-dialog
display-dialog
exit-dialog
replace-dialog

display-tooltip *Generic Function*

Summary

Displays tooltip help on an output pane.

Package

capi

Signature

display-tooltip *output-pane* **&key** *x* *y* *text*

Arguments

output-pane⇓	An instance of a subclass of **output-pane**.
x⇓	The horizontal coordinate of the tooltip position.
y⇓	The vertical coordinate of the tooltip position.
text⇓	The help text.

Description

The generic function **display-tooltip** displays *text* as tooltip help at position *x*,*y* in *output-pane*.

Notes

1. On GTK+, **display-tooltip** is implemented only for GTK+ versions 2.12 and later.
2. On GTK+, the **:x** and **:y** arguments might not be handled.

Compatibility note

On GTK+, **display-tooltip** is not implemented in LispWorks 6.0.

Examples

```
(example-edit-file "capi/graphics/pinboard-help")
```

See also

3.12.1 Tooltips for output panes

docking-layout *Class*

Summary

A class that implements docking of panes.

Package

capi

Superclasses

simple-layout

Initargs

:items	A list of pane specifications. The panes become the items in the layout.
:controller	A docking layout controller.
:docking-test-function	A function controlling whether a pane can be docked.
:docking-callback	A function called when a pane is docked or undocked.
:divider-p	A boolean allowing a visible edge around the layout.
:orientation	One of **:horizontal** or **:vertical**.

Accessors

docking-layout-controller
docking-layout-divider-p
docking-layout-docking-test-function
docking-layout-items

Readers

docking-layout-orientation

Description

The class **docking-layout** defines a region in which panes can be docked and undocked. The undocking functionality works only in LispWorks for Windows.

If *controller* is non-nil, it must be a controller object as returned by a call to **make-docking-layout-controller**. In this case the **docking-layout** is one of a group of **docking-layout**s which share that same controller, known as the Docking Group. The panes that can be docked and undocked are shared between the members of the Docking Group. If *controller* is **nil** (the default value), the **docking-layout** is in a Docking Group of one.

A pane *pane* is dockable in a Docking Group when it is an item of any member of the Docking Group. This is the case when it is one of the *items* passed to **make-instance** for some member of the group, or it has been set in some member by **(setf docking-layout-items)**. The user can dock and undock *pane* in any member of the Docking Group. You can change the dockable status of panes programmatically by **(setf docking-layout-items)**. You can query a pane's docked and visible status in a **docking-layout** by **docking-layout-pane-docked-p** and **docking-layout-pane-visible-p**. You can change a pane's docked and visible status in a **docking-layout** by

(setf docking-layout-pane-docked-p) and **(setf docking-layout-pane-visible-p)**.

By default, the context menu allows the user to alter the visibility status of each of the panes in the Docking Group.

items is a list of pane specifications. Each specification in the list is either an atom denoting a pane, or a list wherein the **cl:car** is an object denoting a pane and the **cl:cdr** is a plist of options and values. The object denoting the pane can be:

- The pane itself.
- A symbol naming a slot in the interface which contains the **docking-layout**. The value in that slot, which must be a pane, is used. Typically the slot name is defined in the **:panes** or **:layouts** class option in the **define-interface** form.
- A string, denoting a **title-pane** with that text.
- A list, wherein the car is the name of a pane class and the cdr is a list of initialization arguments for that class. This denotes the pane created by applying **make-instance** to the list. Note that in this case the list cannot be the item in the *items* list, because it would be wrongly interpreted as a list wherein the car denotes a pane directly and the cdr is a plist of options and values.

When an item in the *items* list is a list, the cdr is a plist of options and values, which can contain these options:

:title A string which is title associated with the pane. This is used when the pane is presented to the user, for example in the default context menu.

:docked-p A boolean specifying whether the pane should be docked. The default value is **t**. When a pane is not docked and is visible, it is displayed in its own window.

:visible-p A boolean specifying whether the pane is visible. The default value is **t**.

:undocked-geometry A list of four integers specifying the geometry of the pane when undocked, as **(***x y width height***)**.

:start-new-line-p A boolean specifying whether to place the pane on a new line in the **docking-layout**. The default value is **nil**.

docking-layout-items always returns the items as lists, with the cdr containing the options and values.

docking-test-function is a function of two arguments with a boolean return value. When the user attempts to dock a pane *pane* in the **docking-layout**, *docking-test-function* is called with the **docking-layout** and *pane*. If it returns **nil**, *pane* is not docked. If it returns true, *pane* is docked. The default behavior is that all panes under the controller which is the *controller* in this **docking-layout**, and only these panes, can be docked.

docking-callback, if non-nil, is a function of three arguments: the **docking-layout**, the pane and a boolean. This third argument is **t** when the pane is docked, and **nil** when the pane is undocked. The default value of *docking-callback* is **nil**.

divider-p controls whether a visible edge is drawn around the border of the **docking-layout**. The default value is **nil**.

orientation specifies whether the items are laid out horizontally or vertically. The default value is **:horizontal**.

Examples

```
(example-edit-file "capi/layouts/docking-layout")
```

See also

docking-layout-pane-docked-p
docking-layout-pane-visible-p

docking-layout-pane-docked-p *Accessor*

Summary

Used to indicate whether a pane is currently docked in a **docking-layout**.

Package

capi

Signature

docking-layout-pane-docked-p *docking-layout pane* **&key** *anywhere* **=>** *dockedp*

(setf docking-layout-pane-docked-p) *dockedp docking-layout pane* **&key** *anywhere* **=>** *dockedp*

Arguments

docking-layout⇓	An instance of **docking-layout** or a subclass.
pane⇓	A pane.
anywhere⇓	A boolean.
dockedp⇓	A boolean.

Values

dockedp⇓	A boolean.

Description

The accessor **docking-layout-pane-docked-p** accesses a boolean indicating whether *pane* is currently docked.

If *anywhere* is **t**, *dockedp* is true if *pane* is docked in any member of the Docking Group of *docking-layout*. If *anywhere* is **nil**, *dockedp* is true only if *pane* is docked in *docking-layout* itself. The default value of *anywhere* is **nil**.

(setf docking-layout-pane-docked-p) may be used to change the docking state of pane in *docking-layout* only when *pane* is dockable in the Docking Group of *docking-layout*, that is, it was added to the *items* of any of the **docking-layout**s in the group.

See also

docking-layout

docking-layout-pane-visible-p *Accessor*

Summary

Used to indicate whether a pane is currently visible in a **docking-layout**.

Package

capi

Signature

docking-layout-pane-visible-p *docking-layout pane* **=>** *visiblep*

(setf docking-layout-pane-visible-p) *visiblep docking-layout pane* **=>** *visiblep*

Arguments

docking-layout⇓	An instance of **docking-layout** or a subclass.
pane⇓	A pane.
visiblep	A boolean.

Values

visiblep	A boolean.

Description

The accessor **docking-layout-pane-visible-p** accesses a boolean indicating whether *pane* is currently visible in the Docking Group of *docking-layout*. *pane* may be docked in any member of the Docking Group, or undocked.

(setf docking-layout-pane-visible-p) may be used to change the visibility of pane in *docking-layout* only when *pane* is dockable in the Docking Group of *docking-layout*, that is, it was added to the *items* of any of the **docking-layout**s in the group.

See also

docking-layout

document-container *Class*

Summary

A container for a **document-frame** (only implemented on Microsoft Windows).

Package

capi

Superclasses

capi-object

Readers

screen-interfaces

Description

The class **document-container** is used to implement the container in a **document-frame**.

A **document-container** has some screen-like functionality, responding to **screen-internal-geometry** and **screen-active-interface**.

This works only in LispWorks for Windows.

See also

display
document-frame
screen-active-interface
screen-internal-geometry
3.13 Screens
11 Defining Interface Classes - top level windows

document-frame *Class*

Summary

The class **document-frame** is used to implement MDI (only implemented on Microsoft Windows).

Package

capi

Superclasses

interface

Readers

document-frame-container

Description

The class **document-frame** is used to implement Multiple-Document Interface (MDI) which is a standard technique on Microsoft Windows (see the MSDN for documentation).

To use MDI in the CAPI, define an interface class that inherits from **document-frame**, and use the two special slots **capi:container** and **capi:windows-menu**. For the details and an example, see **6.6.7 Multiple-Document Interface (MDI)**.

This works only in LispWorks for Windows.

Notes

capi:windows-menu is a special slot in **document-frame** and this symbol should not appear elsewhere in the **define-interface** form.

See also

current-document
merge-menu-bars
3.7 Hierarchy of panes
6.6.7 Multiple-Document Interface (MDI)

double-headed-arrow-pinboard-object *Class*

Summary

A **pinboard-object** that draws itself as an arrow.

Package

capi

Superclasses

arrow-pinboard-object

Initargs

:double-head-predicate
A function determining whether a single or double arrowhead is drawn.

Description

The class **double-headed-arrow-pinboard-object** is a **pinboard-object** that draws itself as an arrow, which can switch dynamically from double-headed to single-headed.

double-head-predicate should be a function of two arguments returning a boolean value. The first argument is the output pane on which the arrow pinboard object is drawn. The second argument is the arrow pinboard object itself.

double-head-predicate should return a true value if the arrow is to be double-headed, and **nil** if a single-headed arrow should be drawn. It is called each time the arrow object is redrawn.

Examples

```
(defvar *doublep* t)

(let ((dhr
       (capi:contain
        (make-instance
         'capi:pinboard-layout
         :description
         (list
          (make-instance
           'capi:double-headed-arrow-pinboard-object
           :double-head-predicate
           #'(lambda (x y) *doublep*)
           :start-x 5 :start-y 5 :end-x 95 :end-y 95)
          (make-instance
           'capi:double-headed-arrow-pinboard-object
           :double-head-predicate
           #'(lambda (x y) *doublep*)
```

```
              :head-direction :backwards
               :start-x 5 :start-y 95 :end-x 95 :end-y 5)))
           :visible-min-width 100
           :visible-min-height 100)))
     (dotimes (x 10)
       (sleep 1)
       (setq *doublep* (not *doublep*))
       (mapcar 'capi:redraw-pinboard-object
               (capi:layout-description dhr))))
```

See also

12.3 Creating graphical objects

double-list-panel *Class*

Summary

A **choice** which displays its selected items and its unselected items in disjoint lists displayed in two sub-panels, and facilitates easy movement of items between these lists.

Package

capi

Superclasses

choice
interface

Initargs

:selected-items-title

:unselected-items-title

selected-items-title and *unselected-items-title* are passed as the **:title** initarg to the list panels.

:selected-items-filter

:unselected-items-filter

selected-items-filter and *unselected-items-filter* are passed as the **:filter** initarg to the list panels.

:list-visible-min-width

:list-visible-min-height

list-visible-min-width and *list-visible-min-height* are passed as the **:visible-min-width** and **:visible-min-height** initargs to both list panels.

:image-function

:image-state-function

:image-width

:image-height

:state-image-width

:state-image-height

image-function, *image-state-function*, *image-width*, *image-height*, *state-image-width* and *state-image-height* are passed to both of the sub-panels to specify images.

Description

The class **double-list-panel** is a **choice** which displays its *items* in two **list-panel**s. One list contains the selected items and the other contains the unselected items. There is a pair of arrow buttons which move highlighted items between the lists.

selected-items-title and *unselected-items-title* are passed as the **:title** initarg to the corresponding sub-panels (see **list-panel**). *selected-items-title* defaults to **"Selected items:"** and *unselected-items-title* defaults to **"Unselected items:"**.

selected-items-filter and *unselected-items-filter* are passed as the **:filter** initarg to the corresponding sub-panels (see **list-panel**). *selected-items-filter* and *unselected-items-filter* both default to **nil**.

list-visible-min-width and *list-visible-min-height* are passed as the **:visible-min-width** and **:visible-min-height** initargs to both sub-panels (see **list-panel**). *list-visible-min-width* and *list-visible-min-width* both default to **nil**.

image-function, *image-state-function*, *image-width*, *image-height*, *state-image-width* and *state-image-height* are passed to both of the sub-panels to specify images (see **list-panel**).

The default *interaction* of **double-list-panel** is **:extended-selection**.

The *selection-callback*, *extend-callback* or *retract-callback* is called as appropriate when items are moved between the lists. There is no *action-callback* for **double-list-panel**.

The user selects and de-selects items in the **double-list-panel** by moving them between the two lists. There are three ways to move the items:

- Highlight the items to move by normal **list-panel** selection gestures, then press an arrow button.
- Highlight a single item to move by normal **list-panel** selection gestures, then press **Return**.
- Double click on an item to move it.

Notes

1. **double-list-panel** is not a subclass of **list-panel**.
2. **double-list-panel** does not have image lists. To use sub-images from an **image-set**, use **image-locator**s.

Examples

```
(capi:display
 (make-instance
  'capi:double-list-panel
  :items '("John" "Geoff" "chicken" "blue" "water")
  :selection-callback
  #'(lambda (item choice)
      (capi:display-message "selecting ~a" item))
  :extend-callback
  #'(lambda(item choice)
      (capi:display-message "extending ~a" item))
  :retract-callback
  #'(lambda(item choice)
      (capi:display-message "deselecting ~a" item))))
```

See also

list-panel
5.3 List panels

drag-pane-object *Function*

Summary

Initiates a dragging operation.

Package

capi

Signature

drag-pane-object *pane value* **&key** *string plist image-function operations* **=>** *operation*

Arguments

pane⇓	A pane.
value⇓	An object to be dragged.
string⇓	A string to be dragged or **nil**.
plist⇓	A plist of formats and objects to be dragged.
image-function⇓	A function or **nil**.
operations⇓	A list of operation keywords allowed for the dragged objects.

Values

operation⇓	One of the operation keywords.

Description

The function **drag-pane-object** initiates a dragging operation from within the pane *pane*. It can only be called from within the button **:press** or button **:motion** callbacks of the *input-model* of an **output-pane**.

value, *string* and *plist* are combined to provide an object to be dragged in various formats.

value can be any Lisp object (not necessarily a string) to make available for dropping into a pane within the local Lisp image.

string can be a string representation of *value* to make available, or **nil**. If *string* is **nil** and *value* is a string, then that will be made available as the string.

plist is a property list of additional format/value pairs to make available. The currently supported formats are as described for **set-drop-object-supported-formats**. You can make more than one format available simultaneously.

image-function provides a graphical image for use during the dragging operation on Cocoa. If *image-function* is supplied, then it should be a function of one argument. It might be called to provide an image for use during the dragging operation. The function *image-function* should return three values: a **image** object, an x offset and a y offset. The x and y offsets are the position within the image where the mouse should be located. If the image is **nil** or *image-function* is not supplied then a default image is generated. If the x or y offsets are **nil** or not returned then the image is positioned with the mouse at its center point. The image that is returned by *image-function* is freed automatically in the end of dragging operation. It must be a new image, and cannot be reused.

operations should be a list of operation keywords that the pane will allow the target application to perform. The operation keywords are **:copy**, **:move** and **:link** as described for the effect in **drop-object-drop-effect**. If certain platform-specific modifier keys are pressed, then some of the operations will be ignored.

The return value *operation* indicates which operation was performed by the application where the dragged object was dropped. The value will be **:none** if the object was not dropped anywhere or dragging was abandoned (for example, by the user hitting the **Escape** key). If *operation* is **:move**, then you should update the data structures in your application to remove the object that was dragged.

Notes

1. **drag-pane-object** is not supported on X11/Motif. See **simple-pane** for information about drop callbacks.
2. *image-function* is only called on Cocoa. There is no way to specify an image when dragging on Microsoft Windows.
3. If **:image** is supplied in *plist*, the dragging mechanism automatically frees the **image** object as if by **free-image** when it no longer needs it.

Examples

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

See also

simple-pane
17 Drag and Drop

draw-metafile *Function*

Summary

Draws a metafile to a pane.

Package

capi

Signature

draw-metafile *pane metafile x y width height*

Arguments

pane⇓	An **output-pane**.
metafile⇓	A metafile, as described in **with-internal-metafile**.
x⇓, *y*⇓	Integers.
width⇓, *height*⇓	Non-negative integers.

Description

The function **`draw-metafile`** draws the metafile *metafile* to the pane *pane* at position *x*,*y* with size *width*, *height*.

metafile should be a metafile as returned by **`with-internal-metafile`**.

The **`graphics-state`** parameters *transform*, *mask* and *mask-transform* affect how the metafile is drawn. The other **`graphics-state`** parameters are taken from the metafile.

Notes

1. **`draw-metafile`** is supported on GTK+ only where Cairo is supported (GTK+ 2.8 and later).
2. Metafiles look bad on GTK+, because they transform the image rather than the drawing.
3. **`draw-metafile`** is not implemented on X11/Motif.

Examples

```
(example-edit-file "capi/graphics/metafile")
```

```
(example-edit-file "capi/graphics/metafile-rotation")
```

See also

`can-use-metafile-p`
`clipboard`
`draw-metafile-to-image`
`free-metafile`
`graphics-state`
`with-internal-metafile`

draw-metafile-to-image *Function*

Summary

Draws a metafile as an image.

Package

`capi`

Signature

`draw-metafile-to-image` *pane metafile* **`&key`** *width height max-width max-height background alpha* **`=>`** *image*

Arguments

pane⇓	An **`output-pane`**.
metafile⇓	A metafile.
width⇓, *height*⇓	Non-negative integers, or **`nil`**.
max-width⇓, *max-height*⇓	

	Non-negative integers, or **nil**.
background⇓	A color specification.
alpha⇓	A generalized boolean.

Values

image	An **image**.

Description

The function **draw-metafile-to-image** returns a new **image** object for *pane*, with *metafile* drawn into the image.

metafile should be a metafile as returned by **with-internal-metafile**.

If *width* and *height* are both **nil** then the size of the image is computed from the metafile. If both *width* and *height* are integers, then they specify the size of the image and the metafile is scaled to fit. If one of *width* or *height* is **nil**, then it is computed from the other dimension, preserving the aspect ratio of the metafile. The default values of *width* and *height* are both **nil**.

max-width and *max-height*, if non-nil, constrain the computed or specified values of *width* and *height* respectively. The aspect ratio is retained when the size is constrained, so specifying a *max-width* can also reduce the actual height of the image. The default values of *max-width* and *max-height* are both **nil**.

background should be a color spec, which controls the non-drawn parts of the image. For information about color specs, see **15.1 Color specs**. If *background* is omitted, then the background color of *pane* is used (see **simple-pane**).

If *alpha* is non-nil, then the image will have an alpha component. The default value of *alpha* is **nil**.

Notes

1. **draw-metafile-to-image** is supported on GTK+ only where Cairo is supported (GTK+ 2.8 and later).
2. Metafiles look bad on GTK+, because they transform the image rather than the drawing.
3. **draw-metafile-to-image** is not implemented on X11/Motif.

See also

clipboard
draw-metafile
free-metafile
with-internal-metafile

drawn-pinboard-object *Class*

Summary

The class **drawn-pinboard-object** is a subclass of **pinboard-object** which is drawn by a supplied function, and is provided as a means of the user creating their own pinboard objects.

Package

capi

Superclasses

pinboard-object

Initargs

:display-callback	Called to display the object.

Accessors

drawn-pinboard-object-display-callback

Description

The *display-callback* is called with the output pane to draw on, the **drawn-pinboard-object** itself, and the *x*, *y*, *width* and *height* of the object, and it is expected to redraw that section. The *display-callback* should not draw outside the object's bounds.

An alternative way of doing this is to create a subclass of **pinboard-object** and to provide a method for **draw-pinboard-object**.

Examples

```
(defun draw-an-ellipse
       (output-pane self x y width height)
       (let ((x-radius (floor width 2))
             (y-radius (floor height 2)))
              (gp:draw-ellipse output-pane
                (+ x x-radius) (+ y y-radius)
                x-radius y-radius
                :foreground :red
                :filled t)))


(capi:contain (make-instance
                'capi:drawn-pinboard-object
                :visible-min-width 200
                :visible-min-height 100
                :display-callback 'draw-an-ellipse))
```

There are further examples in **20 Self-contained examples**.

See also

pinboard-layout
12 Creating Panes with Your Own Drawing and Input

draw-pinboard-layout-objects *Function*

Summary

Draws the pinboard objects which intersect a given rectangle in a **pinboard-layout**.

Package

capi

Signature

draw-pinboard-layout-objects *pinboard-layout graphics-port x y width height* **=> nil**

Arguments

pinboard-layout⇓ A **pinboard-layout**.

graphics-port⇓ A graphics port.

x⇓, *y*⇓, *width*⇓, *height*⇓

Non-negative integers.

Description

The function **draw-pinboard-layout-objects** draws the pinboard objects in *pinboard-layout* which intersect the rectangle specified by *x*, *y*, *width* and *height* into the graphics port *graphics-port*.

graphics-port can be *pinboard-layout* itself or another graphics port. The drawing is done into the target rectangle, but may also draw outside it.

Notes

1. **draw-pinboard-layout-objects** is used by **pinboard-layout** when it actually needs to display the objects.
2. **draw-pinboard-layout-objects** does not do any caching. The *display-callback* of **pinboard-layout** does any caching, and may use **draw-pinboard-layout-objects** to draw into a cache (a pixmap) rather than the screen.
3. **draw-pinboard-layout-objects** is useful when you want to have your own *display-callback* for a **pinboard-layout** or a subclass. It is possible to use a graphics transformation on *graphics-port* around the call to **draw-pinboard-layout-objects** to affect the drawing. For example **with-graphics-translation** can be used to move the drawing to the origin.

See also

pinboard-layout
pinboard-layout-display
12 Creating Panes with Your Own Drawing and Input

draw-pinboard-object *Generic Function*

Summary

Draws a pinboard object.

Package

capi

Signature

draw-pinboard-object *pinboard object* **&key** *x y width height* **&allow-other-keys**

Arguments

pinboard⇓ A **pinboard-layout**.

object⇓ A **pinboard-object**.

x⇓, *y*⇓, *width*⇓, *height*⇓

Non-negative integers.

Description

The generic function **draw-pinboard-object** is called whenever *object* needs to be drawn in *pinboard*. *x*, *y*, *width* and *height* indicate the region that needs to be redrawn, but a method is free to ignore these and draw the complete object. However, it should not draw outside the pinboard object's bounds.

Examples

```
(example-edit-file "capi/graphics/circled-graph-nodes")
```

See also

pinboard-layout
pinboard-object
pinboard-object-highlighted-p

draw-pinboard-object-highlighted *Generic Function*

Summary

Draws highlighting on a pre-drawn pinboard object.

Package

capi

Signature

draw-pinboard-object-highlighted *pinboard object* **&key &allow-other-keys**

Arguments

pinboard⇓ A **pinboard-layout**.

object⇓ A **pinboard-object**.

Description

The generic function **draw-pinboard-object-highlighted** draws the highlighting for *object* in *pinboard* after *object* has already been drawn. The default highlighting method draws a box around the object, and should be sufficient for most purposes.

Examples

```
(example-edit-file "capi/graphics/circled-graph-nodes")
```

See also

highlight-pinboard-object

drop-object-allows-drop-effect-p *Function*

Summary

Queries whether a dropping operation can be performed with a given effect.

Package

capi

Signature

drop-object-allows-drop-effect-p *drop-object effect* **=>** *result*

Arguments

drop-object⇓ A *drop-object*, as passed to the *drop-callback*.

effect⇓ An effect keyword.

Values

result A boolean.

Description

The function **drop-object-allows-drop-effect-p** returns non-nil if the dropping operation can be performed for *drop-object* with the given effect *effect*. It returns **nil** if the dropping operation cannot be performed. See **drop-object-drop-effect** for information on drop effect keywords.

Notes

drop-object-allows-drop-effect-p should only be called within a *drop-callback*. It is not supported on X11/Motif. See **simple-pane** for information about drop callbacks.

See also

drop-object-drop-effect
simple-pane

drop-object-collection-index *Accessor*

Summary

Gets the index and relative place in the **collection** that an object is being dropped over.

Package

capi

Signatures

drop-object-collection-index *drop-object* **=>** *index*, *placement*

(setf (drop-object-collection-index *drop-object*) (*values new-index new-placement*))

Arguments

drop-object⇓	A *drop-object*, as passed to the *drop-callback*.
new-index⇓	An integer.
new-placement⇓	One of **:above**, **:item** or **:below**.

Values

index⇓	An integer.
placement⇓	One of **:above**, **:item** or **:below**.

Description

The accessor **drop-object-collection-index** accesses the index and place relative to that index within the **collection** that the object *drop-object* is being dropped over. This information is only meaningful when the pane is an instance of **list-panel** or **tree-view**.

The returned value *index* is the position in the **collection** (see **get-collection-item** or **choice-selection**). The returned value *placement* indicates whether the user is dropping above, on or below the item at *index*.

There is also a setf expander that can be called with the values *new-index* and *new-placement* within the **:drag** stage of the operation, to adjust where the user will be allowed to drop the object.

Notes

drop-object-collection-index should only be called within a *drop-callback*. It is not supported on X11/Motif. See **simple-pane** for information about drop callbacks.

Examples

For an example illustrating the use of drag and drop in a **choice**, see:

```
(example-edit-file "capi/choice/drag-and-drop")
```

See also

drop-object-collection-item
17 Drag and Drop

drop-object-collection-item *Accessor*

Summary

Gets the item and relative place in the **collection** that an object is being dropped over.

Package

capi

Signatures

drop-object-collection-item *drop-object* **=>** *item***,** *placement*

(setf (drop-object-collection-item *drop-object***) (***values new-item new-placement***))**

Arguments

drop-object⇓	A *drop-object*, as passed to the *drop-callback*.
new-item⇓	An item of a **collection**.
new-placement⇓	One of **:above**, **:item** or **:below**.

Values

item	An item of a collection.
placement⇓	One of **:above**, **:item** or **:below**.

Description

The accessor **drop-object-collection-item** accesses the item and place relative to that item within the **collection** that the object *drop-object* is being dropped over. This information is only meaningful when the pane is an instance of **list-panel** or **tree-view**.

The returned value *placement* indicates whether the user is dropping above, on or below the item.

There is also a setf expander that can be called with the values *new-item* and *new-placement* within the **:drag** stage of the operation, to adjust where the user will be allowed to drop the object.

Notes

drop-object-collection-item should only be called within a *drop-callback*. It is not supported on X11/Motif. See **simple-pane** for information about drop callbacks.

Examples

For an example illustrating the use of drag and drop in a **choice**, see:

```
(example-edit-file "capi/choice/drag-and-drop")
```

See also

drop-object-collection-index
17 Drag and Drop

drop-object-drop-effect *Accessor*

Summary

Reads or sets the current effect of a dropping operation.

Package

capi

Signature

drop-object-drop-effect *drop-object* **=>** *effect*

(setf drop-object-drop-effect) *effect drop-object* **=>** *effect*

Arguments

drop-object⇓ A *drop-object*, as passed to the *drop-callback*.

effect⇓ An effect keyword.

Values

effect⇓ An effect keyword.

Description

The accessor **drop-object-drop-effect** gets or sets the current effect of the dropping operation for *drop-object*. *effect* can be one of:

:copy The object will be copied. This is the most common value for operations between applications.

:move The object will be moved. This is usually triggered by the user dragging with a platform-specific modifier key pressed.

:link A link to the object will be created. This is usually triggered by the user dragging with a platform -specific modifier key pressed.

:none No dragging is possible.

Notes

drop-object-drop-effect should only be called within a *drop-callback*. It is not supported on X11/Motif. See **simple-pane** for information about drop callbacks.

Examples

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

See also

simple-pane
17 Drag and Drop

drop-object-get-object *Function*

Summary

Returns a dropped object in a given format.

Package

capi

Signature

drop-object-get-object *drop-object pane format* **&rest** *args* **=>** *object*

Arguments

drop-object⇓	A *drop-object*, as passed to the *drop-callback*.
pane⇓	A CAPI pane.
format⇓	A format keyword.
args⇓	Other arguments, currently ignored.

Values

object	An object in the given format.

Description

The function **drop-object-get-object** returns the dropped object in the dropping operation for *drop-object* over *pane* with format *format*. See **set-drop-object-supported-formats** for information on format keywords.

Other arguments in *args* are currently ignored.

Notes

1. When receiving an image (by calling **drop-object-get-object** with the **:image** format), the received image should also be freed when you finish with it. However, it will be freed automatically when the pane supplied to **drop-object-get-object** is destroyed, so normally you do not need to free it explicitly.

2. **drop-object-get-object** should only be called within a *drop-callback*, passing the supplied *drop-object* and *pane*. It is not supported on X11/Motif. See **simple-pane** for information about drop callbacks.

Examples

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

```
(example-edit-file "capi/choice/list-panel-drag-images")
```

See also

set-drop-object-supported-formats
simple-pane
17 Drag and Drop

drop-object-pane-x
drop-object-pane-y *Functions*

Summary

Gets the coordinates in the pane that an object is being dropped over.

Package

capi

Signatures

drop-object-pane-x *drop-object* **=>** *x-coord*

drop-object-pane-y *drop-object* **=>** *y-coord*

Arguments

drop-object⇓ A *drop-object*, as passed to the *drop-callback*.

Values

x-coord, *y-coord* Integers.

Description

The functions **drop-object-pane-x** and **drop-object-pane-y** return the x and y coordinates within the pane that the object is being dropped over in the dropping operation for *drop-object*. This information is only meaningful when the pane is an instance of **output-pane** or one of its subclasses.

Notes

drop-object-pane-x and **drop-object-pane-y** should only be called within a *drop-callback*. They are not supported on X11/Motif. See **simple-pane** for information about drop callbacks.

See also

simple-pane
17 Drag and Drop

drop-object-provides-format *Function*

Summary

Queries whether a dropping operation can provide an object in a given format.

Package

capi

Signature

drop-object-provides-format *drop-object* *format* **=>** *result*

Arguments

drop-object⇓ A *drop-object*, as passed to the *drop-callback*.

format⇓ A format keyword.

Values

result A boolean.

Description

The function **drop-object-provides-format** returns non-nil if the dropping operation can provide an object with format *format* in the dropping operation for *drop-object*. It returns **nil** if it cannot provide that format.

See **set-drop-object-supported-formats** for information on format keywords.

Notes

drop-object-provides-format should only be called within a *drop-callback*. It is not supported on X11/Motif. See **simple-pane** for information about drop callbacks.

Examples

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

See also

set-drop-object-supported-formats
simple-pane
17 Drag and Drop

echo-area-cursor-inactive-style *Variable*

Summary

The drawing style of the Echo Area cursor when the window is inactive.

Package

capi

Initial Value

:invisible

Description

The variable ***echo-area-cursor-inactive-style*** specifies the drawing style of the cursor in the Echo Area of an inactive window in the LispWorks IDE.

The allowed values are **:inverse**, **:outline**, **:underline** and **:invisible**.

echo-area-pane *Class*

Summary

The class of the Editor's echo area.

Package

capi

Superclasses

editor-pane

Description

The class **echo-area-pane** is used to implement the small window for user interaction, known as the Echo Area, which is at the bottom of Editor windows in the LispWorks IDE.

You should not normally need to work with this class directly. To add an Echo Area, pass **:echo-area t** when making the **editor-pane**.

editor-cursor-active-style *Variable*

Summary

The drawing style of the editor's cursor when the window is active.

Package

capi

Initial Value

:inverse

Description

The variable ***editor-cursor-active-style*** specifies the drawing style of an **editor-pane** cursor when the window is active.

The allowed values are **:inverse**, **:outline**, **:underline**, **:left-bar** and **:caret**.

See also

editor-pane-blink-rate

editor-cursor-color *Variable*

Summary

The background color of the cursor.

Package

capi

Initial Value

nil

Description

When non-nil, the value of the variable ***editor-cursor-color*** is a color spec or color alias determining the background color of the **editor-pane** cursor. See **15 The Color System** for information about color specs and aliases.

The value **nil** means that the cursor background color is the same as the *foreground* color of the editor pane. *foreground* is a slot inherited from **simple-pane**.

Examples

```
(setf capi:*editor-cursor-color* :red)
```

editor-cursor-drag-style *Variable*

Summary

The drawing style of the editor's cursor during a selection drag.

Package

`capi`

Initial Value

`:left-bar`

Description

The variable **`*editor-cursor-drag-style*`** specified the drawing style of an **`editor-pane`** cursor during a selection drag.

The allowed values are **`:inverse`**, **`:outline`**, **`:underline`**, **`:left-bar`** and **`:caret`**.

editor-cursor-inactive-style *Variable*

Summary

The drawing style of the editor's cursor when the window is inactive.

Package

`capi`

Initial Value

`:outline`

Description

The variable **`*editor-cursor-inactive-style*`** specifies the drawing style of an **`editor-pane`** cursor when the window is inactive.

The allowed values are **`:inverse`**, **`:outline`**, **`:underline`** or **`:invisible`**.

See also

`editor-pane`

editor-pane *Class*

Summary

An editor pane is an editor that has all of the functionality described in the *LispWorks Guide To The Editor*.

Package

`capi`

Superclasses

output-pane

Subclasses

interactive-pane
collector-pane

Initargs

:text	A string or **nil**.
:enabled	**t, nil** or **:read-only**.
:buffer-modes	A list specifying the modes of the editor buffer.
:buffer-name	A string, an editor buffer or the keyword **:temp**.
:buffer	A synonym for the initarg **:buffer-name**.
:change-callback	A function designator, or **nil**.
:before-input-callback	A function designator, or **nil**.
:after-input-callback	A function designator, or **nil**.
:echo-area	A flag determining whether the editor pane has an Echo Area.
:fixed-fill	An integer specifying the fill length, or **nil**.
:flag	A non-keyword symbol.
:line-wrap-marker	A **character**, or **nil**.
:line-wrap-face	An **editor:face** object, or a symbol naming a face, or **nil**.
:wrap-style	**nil**, **t** or the keyword **:split-on-space**.
:composition-face	Changes the editor face that is used by **editor-pane-default-composition-callback** to display the composition string. The default value is **:default**.

Accessors

editor-pane-text
editor-pane-change-callback
editor-pane-enabled
editor-pane-fixed-fill
editor-pane-line-wrap-marker
editor-pane-line-wrap-face
editor-pane-wrap-style
editor-pane-composition-face

Description

The class **editor-pane** is an editor that has all of the functionality described in the *LispWorks Guide To The Editor*.

enabled controls how user input affects the **editor-pane**. If enabled is **nil**, all input from the mouse and keyboard is ignored. When enabled is **t**, all input is processed according to the *input-model*. When *enabled* is **:read-only**, input to the pane by keyboard or mouse gestures cannot change the text. More accurately, input via the default *input-model* of editor-pane cannot change the text. The **Cut** and **Paste** menu entries are also disabled. When a user tries to change the text, the operation quietly aborts. Programmatic modifications of the text are still allowed (see Notes below for more detail).

The enabled state can be set by the accessor **`editor-pane-enabled`**. **`simple-pane-enabled`** has the same effect when applied to an **`editor-pane`**.

The pane stores text in buffers which are uniquely named, and so to create an **`editor-pane`** using an existing buffer you should pass the *buffer-name*. To create an **`editor-pane`** with a new buffer, use either *flag* or a non-empty *text* string or a *buffer-name* that does not match any existing buffer.

buffer-name can also be an editor buffer naming itself.

buffer-name can also be the keyword **`:temp`**. In this case the **`editor-pane`** will be created with a temporary buffer that will go away when the **`editor-pane`** is Garbage Collected (it is created by **`editor:make-buffer`** with **`:temporary t`**).

A non-empty string value of *text* specifies the initial text displayed and forces the creation of a new buffer. The accessor **`editor-pane-text`** is provided to read and write the text in the editor buffer.

buffer-modes allows you to specify the initial major mode and minor modes of the **`editor-pane`**'s buffer. It should be a list of the form **`(`***major-mode-name* **`.`** *minor-mode-names***`)`**. See the *Editor User Guide* for a description of major and minor modes in the LispWorks editor. *buffer-modes* is used only when the CAPI creates the buffer, and not when it reuses a buffer.

If *echo-area* is non-nil. then an Echo Area is added. *echo-area* defaults to **`nil`**.

If *fixed-fill* is non-nil, the editor pane tries to form lines of length close to, but no more than, *fixed-fill*. It does this by forcing line breaks at spaces between words. *fixed-fill* defaults to **`nil`**.

The cursor in an **`editor-pane`** blinks on and off by the mechanism described in **`editor-pane-blink-rate`**.

change-callback, if non-nil, should be a function which is called whenever the editor buffer under the **`editor-pane`** changes. For the details see **3.5.3.1 Editor pane callbacks**.

before-input-callback and *after-input-callback*, if non-nil, should be functions which are called when **`call-editor`** is called. For the details see **3.5.3.1 Editor pane callbacks**.

line-wrap-marker specifies the marker to display at the end of a line that is wrapped to the next line, or truncated if *wrap-style* is **`nil`**. The value must be a **`character`**, or **`nil`** (which is interpreted as **`#\Space`**). The default value is the value of **`*default-editor-pane-line-wrap-marker*`**. The value can be read by **`editor-pane-line-wrap-marker`**.

line-wrap-face specifies a face to use when displaying the *line-wrap-marker*. The argument can be **`nil`**, an **`editor:face`** object (the result of a call to **`editor:make-face`**), or a symbol naming a face (that is, the first argument to **`editor:make-face`**).

The default value of *line-wrap-face* is an internal symbol naming a face. The value can be accessed by **`editor-pane-line-wrap-face`**. The default face can be modified in the LispWorks IDE via **Tools > Preferences... > Environment > Styles > Colors and Attributes**, style name **Line Wrap Marker**.

wrap-style defines the wrapping of text lines that cannot be displayed in one line of the **`editor-pane`**. The argument can be one of:

`t`	Normal wrapping. Display as many characters as possible in the **`editor-pane`** line.
`nil`	Do not wrap. Text lines that are too long are truncated.
`:split-on-space`	Wrapping, but attempts to split lines on spaces. When the text reaches the end of a line, the code looks backwards for space, and wraps before it.

The default value of *wrap-style* is **`t`** and the value can accessed by **`editor-pane-wrap-style`**.

The input behavior of an **`editor-pane`** is determined by its *input-model* (inherited from **`output-pane`**). By default, an **`editor-pane`** has an *input-model* that implements the functionality of the Editor tool in the LispWorks IDE, and always does it via **`call-editor`**. You can replace this behavior by supplying **`:input-model`** when you call **`make-instance`** or by **`(setf capi:output-pane-input-model)`**, though this has an effect only if called before the pane is displayed. It is possible to achieve a minor modification to the default input behavior by prepending the modification (see the example below). Note that functions performing editor operations must do this via **call-editor**.

Editor panes support GNU Emacs keys on all platforms. Additionally on Microsoft Windows they support Windows editor keys, on GTK+ and Motif they support KDE/Gnome keys, and on Cocoa they support macOS editor keys. Exactly one style of emulation is active at any one time for each editor pane. By default, editor panes in the LispWorks IDE use Emacs emulation on all platforms. By default, editor panes in delivered applications use Windows emulation on Microsoft Windows, macOS editor emulation on Cocoa, and Emacs emulation on GTK+ and Motif. To alter the choice of emulation, see **interface-keys-style** or the **deliver** keyword **:editor-style**, described in the *Delivery User Guide*.

Notes

1. The **output-pane** initarg **:drawing-mode** controls anti-aliasing of the text displayed in an **editor-pane** on Microsoft Windows and GTK+.
2. For an **editor-pane** with *enabled* **:read-only**, Editor commands (predefined, and user-defined by **editor:defcommand**) may or may not be able to change the text, depending on how they are called. When executed by a key sequence they cannot change the text directly. However Editor commands can also be called via **editor:process-character** or **call-editor**, and then are programmatic input and so can change the text.
3. The effect of *enabled* **:read-only** is on the **editor-pane**. It does not affect the underlying Editor buffer, which can still be modified from other panes. The buffer that is displayed can be changed, and this does not affect the *enabled* state of the **editor-pane**.
4. Except when actually editing a file, it is normally best to use a temporary buffer when using an **editor-pane**, supplying **:buffer-name :temp** (or **:buffer-name** *tb*, where *tb* is created by **editor:make-buffer** with **:temporary t**). This prevents auto-saving and sharing buffers unintentionally.
5. To control whether the native input method is used to interpret keyboard input, you can supply the **output-pane** initarg **:use-native-input-method** or call **set-default-use-native-input-method**.
6. The default value of *composition-callback* (see **output-pane**) is **editor-pane-default-composition-callback**.

Compatibility note

In LispWorks 4.4 and previous versions **editor-pane** supports only fixed-width fonts.

On Cocoa, **editor-pane** supports only fixed-width fonts in LispWorks 6.1 and earlier versions.

In LispWorks 6.1 and later versions, variable-width fonts can be used on Microsoft Windows, GTK+ and Motif. In LispWorks 7.0 and later, variable-width fonts can also be used on Cocoa. Specify the font via the **:font** initarg (see **simple-pane**).

The initarg **:wrap-style** supersedes **editor:set-window-split-on-space**, which is deprecated.

Examples

```
(capi:contain (make-instance 'capi:editor-pane
                             :text "Hello world"
                             :buffer-name :temp))

(setq ed (capi:contain
          (make-instance 'capi:editor-pane
                         :text "Hello world"
                         :enabled nil
                         :buffer-name :temp)))
```

Note that you cannot type into the editor pane.

```
(capi:apply-in-pane-process
  ed #'(setf capi:editor-pane-enabled) t ed)
```

Now you can enter text into the editor pane interactively.

You can also change the text programmatically:

```
(capi:apply-in-pane-process
  ed #'(setf capi:editor-pane-text) "New text" ed)
```

In this example the callback modifies the buffer in the correct editor context so you that see the editor update immediately:

```
(capi:define-interface updating-editor ()
  ()
  (:panes
   (numbers capi:list-panel
            :items '(1 2 3)
            :selection-callback 'update-editor
            :callback-type :interface
            :visible-min-height '(:character 3))
   (editor capi:editor-pane
           :text
           "Select numbers in the list above."
           :visible-min-width
           (list :character 35)
           :buffer-name :temp)))

(defun update-editor (interface)
  (with-slots (numbers editor) interface
    (editor:process-character
     (list #'(setf capi:editor-pane-text)
           (format nil "~R"
                   (capi:choice-selected-item numbers))
           editor)
     (capi:editor-window editor))))

(capi:display (make-instance 'updating-editor))
```

This example illustrates the use of *buffer-modes* to specify a major mode:

```
(defclass my-lisp-editor (capi:editor-pane) ()
  (:default-initargs
   :buffer-modes '("Lisp")
   :echo-area t
   :text
   ";; Lisp mode functionality such as command bindings and
;; parenthesis balancing work in this window.

(list 1 2 3)
"
   :visible-min-width '(:character 60)
   :name "My Lisp Editor Pane"))

(capi:define-interface my-lisp-editor-interface ()
  ()
  (:panes
   (ed
    my-lisp-editor
    ))
  (:default-initargs
   :title "My Lisp Editor Interface"))

;; Ensure Emacs-like bindings regardless of platform
```

```
(defmethod capi:interface-keys-style
           ((self my-lisp-editor-interface))
  :emacs)

(capi:display
 (make-instance 'my-lisp-editor-interface))
```

This example makes an **editor-pane** with no input behavior:

```
(capi:contain
 (make-instance 'capi:editor-pane
                :input-model nil
                :buffer-name :temp))
```

This example makes an **editor-pane** with the default input behavior, except that pressing the mouse button displays a message rather than setting the point. It then displays the pane:

```
(progn
  (defun foo (self x y)
    (capi:display-message "Button-1 Press at ~a/~a"
                          x y))
  (let ((ep (make-instance 'capi:editor-pane
                           :buffer-name :temp)))
    (setf (capi:output-pane-input-model ep)
          (list* '((:button-1 :press) foo)
                 (capi:output-pane-input-model ep)))
    (capi:contain ep)))
```

Also see these examples:

```
(example-edit-file "capi/editor/")
```

See also

call-editor
default-editor-pane-line-wrap-marker
editor-pane-blink-rate
editor-cursor-active-style
editor-cursor-inactive-style
editor-cursor-color
editor-cursor-drag-style
editor-cursor-inactive-style
interface-keys-style
modify-editor-pane-buffer
output-pane
set-default-use-native-input-method
3.5 Displaying and entering text
10.6 In-place completion

editor-pane-blink-rate *Generic Function*

Summary

Returns the cursor blinking rate for an editor pane.

Package

capi

Signature

editor-pane-blink-rate *pane* **=>** *blink-rate*

Arguments

pane⇓ An **editor-pane**.

Values

blink-rate⇓ A non-negative real number, or **nil**.

Description

LispWorks calls the generic function **editor-pane-blink-rate** to determine the cursor blinking rate in milliseconds for *pane*. The pane uses the value *blink-rate* each time it gets the focus.

If *blink-rate* is a positive real number, then it is the blinking rate in milliseconds. If *blink-rate* is 0, then there is no blinking. If *blink-rate* is **nil**, then the default blinking rate is used.

The default method on **editor-pane-blink-rate** returns **nil**, which means use the default blinking rate. **set-default-editor-pane-blink-rate**.

You can define methods on **editor-pane-blink-rate** specializing on your own subclassses of **editor-pane**.

See also

editor-cursor-active-style
editor-pane
editor-pane-native-blink-rate
set-default-editor-pane-blink-rate
3.5 Displaying and entering text

editor-pane-buffer *Function*

Summary

Returns the editor buffer associated with an editor pane.

Package

capi

Signature

editor-pane-buffer *pane*

Arguments

pane⇓ An **editor-pane**.

Description

The function **editor-pane-buffer** returns the editor buffer associated with *pane*, which can be manipulated in the standard ways with the routines in the editor package.

Examples

```
(setq editor-pane
      (capi:contain (make-instance 'capi:editor-pane
                                   :text "Hello world")))


(setq buffer
      (capi:editor-pane-buffer editor-pane))


(editor:insert-string (editor:buffers-end buffer)
 (format nil "~%Here's some more text..."))
```

See also

editor-pane

editor-pane-composition-selected-range-face-plist *Variable*

Summary

Can modify the face of the default editor composition string.

Package

capi

Initial Value

(:inverse-p t)

Description

The variable ***editor-pane-composition-selected-range-face-plist*** is a plist that is used to modify the face of the composition string when **:selected-range** and **:selection-needs-face** are passed in the plist to **editor-pane-default-composition-callback**. The plist is merged into the plist that is passed into **editor-pane-default-composition-callback**, so keywords in it override the keywords in the face.

See also

editor-pane-default-composition-callback

editor-pane-default-composition-callback *Function*

Summary

The default composition callback of the editor. Composition here means composing input characters into other characters by an input method.

Package

capi

Signature

editor-pane-default-composition-callback *editor-pane what*

Arguments

editor-pane⇓	An **editor-pane**.
what⇓	One of **:start**, **:end** or a plist.

Description

The function **editor-pane-default-composition-callback** is the default *composition-callback* of **editor-pane**. It may also be called by your program.

editor-pane is the **editor-pane** that is currenltly being used for composition.

When *what* is **:start**, **editor-pane-default-composition-callback** sets the composition placement in the editor by calling **set-composition-placement**, and also makes it move the composition window following the user's mouse cursor movement.

When *what* is **:end**, it stops the following of the mouse cursor.

When *what* is a list (which needs to be a plist), **editor-pane-default-composition-callback** checks if it contains a keyword/value pair for **:string-face-lists**, and if it does displays it in the editor temporarily (until the next call to it). See the entry for **output-pane** for the description of the value *string-face-lists*.

By default, **editor-pane-default-composition-callback** uses the faces that are supplied in *string-face-lists*, but if the plist contains **:selection-needs-face** and **:selected-range**, it displays the selected range with a different face, by merging ***editor-pane-composition-selected-range-face-plist*** into the given face of the selected range.

This can be overridden by setting the *composition-face* in the **editor-pane**, or the global ***editor-pane-default-composition-face*** if the *composition-face* of the pane is **:default**. If *composition-face* is a true value then the exact behavior depends on its type:

A plist — This is appended to each face plist in the the *string-face-lists*. In other words, it provides default values for the attributes of the face.

An **editor:face** — Overrides the supplied face completely.

A function or a symbol

For *string-face-list*, funcalls it with two arguments, the pane and the supplied face plist, and uses the result (which may be an **editor:face** or a face plist).

editor-pane-default-composition-callback is the default value of *composition-callback* for **editor-pane**. This can be overridden by passing **:composition-callback** or using **output-pane-composition-callback** (see entry for **output-pane**).

The user-supplied callback may call **editor-pane-default-composition-callback** to do the actual display, potentially after modifying the argument when it is a plist.

See also

set-composition-placement

editor-pane-default-composition-face *Variable*

Summary

The default composition face for **editor-pane**.

Package

capi

Initial Value

nil

Description

The variable ***editor-pane-default-composition-face*** gives the default composition face for all **editor-pane**s where the *composition-face* is set to **:default**.

:default is the default value for *composition-face*, so normally setting this variable affects the *composition-face* of all **editor-pane**s.

See **editor-pane-default-composition-callback** for a description of how it is used.

See also

editor-pane-default-composition-callback

editor-pane-native-blink-rate *Function*

Summary

Returns the native cursor blinking rate for an **editor-pane**.

Package

capi

Signature

editor-pane-native-blink-rate *pane* **=>** *blink-rate*

Arguments

pane⇓ An **editor-pane**.

Values

blink-rate⇓ A non-negative real number, or **nil**.

Description

The function **editor-pane-native-blink-rate** returns the native cursor blinking rate for the **editor-pane** *pane*, that is the rate that the GUI library (Motif, Microsoft Windows, Cocoa) uses.

The value *blink-rate* is interpreted as a blinking rate as described in **editor-pane-blink-rate**.

See also

editor-pane-blink-rate
set-default-editor-pane-blink-rate

editor-pane-selected-text *Function*

Summary

Returns the selected text in an **editor-pane**.

Package

capi

Signature

editor-pane-selected-text *editor-pane* **=>** *result*

Arguments

editor-pane⇓ An **editor-pane**.

Values

result A string or **nil**.

Description

The function **editor-pane-selected-text** takes an instance of **editor-pane** as its argument and returns the selected text in *editor-pane*, or **nil** if there is no selection.

See also

editor-pane
editor-pane-selected-text-p

editor-pane-selected-text-p *Function*

Summary

The predicate for a current selection in an **editor-pane**.

Package

capi

Signature

editor-pane-selected-text-p *editor-pane* **=>** *result*

Arguments

editor-pane⇓ An **editor-pane**.

Values

result A boolean.

Description

The function **editor-pane-selected-text-p** takes an instance of **editor-pane** as its argument and returns **t** if there is text currently selected in *editor-pane*, or **nil** if there is no selection.

See also

editor-pane
editor-pane-selected-text

editor-pane-stream *Generic Function*

Summary

Returns the output stream associated with an editor pane.

Package

capi

Signature

editor-pane-stream *editor-pane* **=>** *stream*

Arguments

editor-pane⇓ An **editor-pane**.

Values

stream An output stream.

Description

The generic function **`editor-pane-stream`** returns the stream where the results of evaluation in the editor buffer currently associated with *editor-pane* are printed to.

See also

`editor-pane`

editor-window *Generic Function*

Summary

Returns the editor window object.

Package

`capi`

Signature

`editor-window` *editor* **`=>`** *editor-window*

Arguments

editor⇓ An **`editor-pane`** or an Editor interface in the LispWorks IDE.

Values

editor-window An editor window object.

Description

The generic function **`editor-window`** returns the editor window object associated with *editor*.

The functionality of editor windows is documented in the *Editor User Guide*.

See also

`editor-pane`

element *Class*

Summary

The class **`element`** is the superclass of all CAPI objects that appear in a window.

Package

`capi`

Superclasses

`capi-object`

Subclasses

`simple-pane`
`menu`

Initargs

`:parent` The element containing this element.

`:interface` The interface containing this element.

`:accepts-focus-p` Specifies that the element should accept input.

`:help-key` An object used for lookup of help. Default value **`t`**.

`:widget-name` A string designator.

`:initial-constraints`

Specifies constraints (geometry hints) that apply to the element during the creation of the element's interface, but not after the interface is displayed.

`:x` A geometry hint specifying the initial *x* position of the element in a pinboard.

`:y` A geometry hint specifying the initial *y* position of the element in a pinboard.

`:external-min-width`

A geometry hint specifying the minimum width of the element in its parent.

`:external-min-height`

A geometry hint specifying the minimum height of the element in its parent.

`:external-max-width`

A geometry hint specifying the maximum width of the element in its parent.

`:external-max-height`

A geometry hint specifying the maximum height of the element in its parent.

`:visible-min-width` A geometry hint specifying the minimum visible width of the element.

`:visible-min-height`

A geometry hint specifying the minimum visible height of the element.

`:visible-max-width` A geometry hint specifying the maximum visible width of the element.

`:visible-max-height`

A geometry hint specifying the maximum height of the element.

`:internal-min-width`

A geometry hint specifying the minimum width of the display region.

`:internal-min-height`

A geometry hint specifying the minimum height of the display region.

`:internal-max-width`

A geometry hint specifying the maximum width of the display region.

`:internal-max-height`

A geometry hint specifying the maximum height of the display region.

Accessors

`element-parent`
`element-widget-name`

Readers

`element-interface`
`help-key`

Description

The class **`element`** contains the slots *parent* and *interface* which contain the element and the interface that the element is contained in respectively. The writer method **`element-parent`** can be used to re-parent an element into another parent (or to remove it from a container entirely by setting its parent to **`nil`**). Note that an element should not be used in more than one place at a time.

The initarg *accepts-focus-p* specifies that the element can accept input. The default value is **`t`**. In some subclasses including **`display-pane`** and **`title-pane`** the default value of *accepts-focus-p* is **`nil`**. A pane accepts the input focus if and only if the function **`accepts-focus-p`** returns true.

accepts-focus-p also influences whether a pane is a tabstop on Microsoft Windows, where a pane acts as a tabstop if and only if the function **`accepts-focus-p`** returns true and the **`:accepts-focus-p`** initarg value is **`:force`**. On Motif and Cocoa, a pane acts as a tabstop if and only if the function **`accepts-focus-p`** returns true.

help-key is used to determine how help is displayed for the pane. The value **`nil`** means that no help is displayed. Otherwise, *help-key* is passed to the *help-callback*, except when *help-key* is **`t`**, when the name of the pane is passed to the *help-callback*. For details of *help-callback*, see **`interface`**.

widget-name specifies the widget name of the element. This is used to match resources on GTK+ and Motif. Note that this name will be in the path only if the element has a representation. **`tab-layout`** and **`pinboard-layout`** always have a representation, as do all elements that show anything on the screen. Other layouts may or may not have a representation and so you should not supply *widget-name* for these.

The actual widget name is the result of a call to **`cl:string`**, except when *widget-name* is a symbol, in which case the symbol name is downcased to derive the widget name.

If *widget-name* is not supplied, the system constructs a default widget name which is the name of the class of the widget (downcased), except for top level interfaces on GTK+ where the *application-class* is prepended followed by a dot.

Example GTK+ resource files are in **`lib/8-0-0-0/examples/gtk/`**.

Note: When *widget-name* is supplied, the GTK+ library does not prepend the *application-class*.

The accessor **`element-widget-name`** gets and (with **`setf`**) sets the *widget-name*. *widget-name* is used when the widget is created, that is when **`display`** is called on the top level interface of the element. Setting *widget-name* afterwards has no effect.

All elements accept initargs (listed above) representing hints as to the initial size and position of the element. By default elements have a minimum pixel size of one by one, and a maximum size of **`nil`** (meaning no maximum), but the hints can be specified to change these values. For the detailed interpretation of, and possible values for, these hints see **6.4.1 Width and height hints**.

Notes

1. Some classes have default initargs providing useful hints. For example, **`display-pane`** has **`:text-height`** as the default value of **`:visible-min-height`**, ensuring that the text is visible.

2. The *ratios*, *x-ratios* and *y-ratios* settings in some layouts (for example **`grid-layout`**) also control the actual size of the pane when the constraints are not specified. In particular, if **`nil`** is used in the ratios then the associated pane(s) will be fixed at their minimum size.

Examples

```
(capi:display (make-instance 'capi:interface
                              :title "Test"
                              :visible-min-width 300))
```

```
(capi:display (make-instance 'capi:interface
                              :title "Test"
                              :visible-min-width 300
                              :visible-max-height 200))
```

Here is a simple example that demonstrates the use of the **`element-parent`** accessor to place elements.

```
(setq pinboard (capi:contain
                (make-instance
                 'capi:pinboard-layout)
                :visible-min-width 520
                :visible-min-height 395))

(setq object
      (make-instance
       'capi:image-pinboard-object
       :x 10 :y 10
       :image
       (example-file "capi/graphics/Setup.bmp")
       :parent pinboard))

(capi:apply-in-pane-process
 pinboard #'(setf capi:element-parent) nil object)

(capi:apply-in-pane-process
 pinboard #'(setf capi:element-parent) pinboard object)
```

These final two examples illustrate the effect of *initial-constraints*.

Create a pane that starts at least 600 pixels high, but can be made shorter by the user:

```
(capi:contain
 (make-instance 'capi:output-pane
                :initial-constraints '(:visible-min-height 600)))
```

Compare with this, which creates a pane at least 600 pixels high but which cannot be made shorter.

```
(capi:contain
 (make-instance 'capi:output-pane
                :visible-min-height 600))
```

See also

`set-hint-table`
3.1.5 Focus
3.7 Hierarchy of panes
3.12 Tooltips
19.3.2 Matching resources for GTK+

6 Laying Out CAPI Panes

element-container *Function*

Summary

Returns the container of an element.

Package

capi

Signature

element-container *element* **=>** *container*

Arguments

element⇓ An **element**.

Values

container⇓ A **screen** or a **document-frame**.

Description

The function **element-container** returns the container of the element *element*.

If *element* is inside a standalone interface, then *container* is the **screen** object.

If *element* is inside an interface that is inside a MDI interface, then *container* is the **capi:container** object of that MDI interface. See **document-frame** for details.

See also

document-frame
element
3.7 Hierarchy of panes

element-interface-for-callback *Generic Function*

Summary

Returns the interface that is used in an element's callbacks.

Package

capi

Signature

element-interface-for-callback *element* **=>** *interface*

Arguments

element⇓ An **element**.

Values

interface An **interface**.

Description

The generic function **element-interface-for-callback** returns the interface that is passed to callbacks in *element*. Normally this is the interface that *element* is in, but that can be changed by **attach-interface-for-callback**.

See also

attach-interface-for-callback
element
3.4 Callbacks

element-screen *Function*

Summary

Returns the screen that an element is associated with.

Package

capi

Signature

element-screen *element* **=>** *screen*

Arguments

element⇓ An **element**.

Values

screen A **screen**.

Description

The function **element-screen** returns the screen that the element *element* is associated with.

See also

element
3.7 Hierarchy of panes

3.13 Screens

ellipse *Class*

Summary

A pinboard object that draws itself as an ellipse.

Package

capi

Superclasses

pinboard-object

Initargs

:filled A boolean.

Accessors

filled

Description

The class **ellipse** is a **pinboard-object** that draws itself as an ellipse.

If *filled* is true, then the ellipse is filled with the foreground color. *filled* defaults to **nil**.

See also

12.3 Creating graphical objects

ensure-area-visible *Function*

Summary

Ensures an area is visible in a scrollable pane.

Package

capi

Signature

ensure-area-visible *pane x y width height*

Arguments

pane⇓ A displayed **output-pane** or **layout**.

x⇓, *y*⇓ The coordinates of the origin of the area to make visible.

width⇓, *height*⇓ The dimensions of the area to make visible.

Description

The function **ensure-area-visible** ensures that the area of *pane* specified by *x*, *y*, *width* and *height*, or at least part of it, is visible.

This function can be used only for instances of **output-pane** of **layout** which have at least one scroll bar.

ensure-interface-screen *Function*

Summary

Ensures that a top level interface is displayed on a given screen.

Package

capi

Signature

ensure-interface-screen *interface* **&key** *screen*

Arguments

interface⇓ An **interface**.

screen⇓ A **screen**, or any argument accepted by **convert-to-screen**.

Description

The function **ensure-interface-screen** ensures that the top level interface *interface* is displayed on the given *screen* (or the default) if **display** is called later without a *screen* argument. This allows the querying of font and color information associated with a particular screen. It returns the screen that is used.

See also

screen
display
interface

execute-with-interface *Function*

Summary

Allows functions to be executed in the event process of a given interface.

Package

capi

Signature

`execute-with-interface` *interface function* **`&rest`** *args*

Arguments

interface⇓	An **`interface`**.
function⇓	A function designator.
args⇓	Arguments passed to *function*.

Description

The function **`execute-with-interface`** is a useful way of operating on an **`interface`** owned by another process. It takes a top-level interface, a function designator *function* and some arguments *args* and queues the function to be run by that process when it next enters its event loop (for an interface owned by the current process, it calls the function immediately).

Notes

1. **`execute-with-interface`** applies *function* even if *interface* does not have a screen representation, for example when it is destroyed. To call *function* only if *interface* has a representation, use **`execute-with-interface-if-alive`**.
2. All accesses (reads as well as writes) on a CAPI interface and its sub-elements should be performed in the interface process. Within a callback on the interface this happens automatically, but **`execute-with-interface`** is a useful utility in other circumstances.
3. **`execute-with-interface`** calls *function* on the current process if *interface* does not have a process.
4. **`apply-in-pane-process`** and **`apply-in-pane-process-if-alive`** are other ways to call a function in the appropriate CAPI process. They takes panes of all classes, not merely **`interface`**.

Examples

```
(setq a (capi:display (make-instance 'capi:interface)))
```

```
(capi:execute-with-interface
 a 'break
 "Break inside the interface process")
```

```
(example-edit-file "capi/elements/progress-bar-from-background-thread")
```

See also

`apply-in-pane-process`
`apply-in-pane-process-if-alive`
`execute-with-interface-if-alive`
4.1 The correct thread for CAPI operations
7 Programming with CAPI Windows

execute-with-interface-if-alive *Function*

Summary

Executes a function in the event process of a given interface if it is alive.

Package

`capi`

Signature

`execute-with-interface-if-alive` *interface function* **`&rest`** *args* **`=>`** *alivep*

Arguments

interface⇓	An **`interface`**.
function⇓	A function designator.
args⇓	Arguments passed to *function*.

Values

alivep⇓	A boolean.

Description

The function **`execute-with-interface-if-alive`** applies the function *function* to the arguments *args* in the process of the interface *interface*, if the interface is "alive". An interface become alive during the creation process before **`interface-display`** is called (and before **`display`** returns). It stops being alive once it is destroyed, either programmatically or by the user.

If *interface* is not alive, *function* is not applied. This is in contrast to **`execute-with-interface`**, which in this case applies the function in the current process.

The return value *alivep* is true if *interface* was alive while **`execute-with-interface-if-alive`** executed. It does not guarantee that *function* is going to be called.

`execute-with-interface-if-alive` is useful for automatic updating of interfaces that may be destroyed by the user, where the update is redundant if the interface is not alive.

Notes

1. The return value is useful for checking whether the interface has gone away (for example closed by the user), in which case the caller may want to do something, most typically stop calling **`execute-with-interface-if-alive`** on the dead interface. It should be checked only when the caller knows that the interface is already displayed (**`display`** returned, or **`interface-display`** was called on it), otherwise it may be **`nil`** because it is not displayed yet.
2. All accesses (reads as well as writes) on a CAPI interface and its sub-elements should be performed in the interface process. Using **`execute-with-interface-if-alive`** is one way of ensuring this.

See also

apply-in-pane-process-if-alive
execute-with-interface
4.1 The correct thread for CAPI operations
7 Programming with CAPI Windows

exit-confirmer *Function*

Summary

Called by the **OK** button on a dialog created with **popup-confirmer**.

Package

capi

Signature

exit-confirmer &rest *dummy-args*

Arguments

dummy-args⇓ Ignored.

Description

The function **exit-confirmer** is called by the **OK** button on a dialog created using **popup-confirmer**, and it is provided as an entry point so that other callbacks can behave in the same way. There is a full description of the **OK** button in **popup-confirmer**.

All of the arguments in *dummy-args* are ignored.

Examples

This example demonstrates the use of **exit-confirmer** to make the dialog exit when pressing **Return** in the text input pane. It also demonstrates the use of *value-function* as a means of deciding the return value from **popup-confirmer**.

```
(capi:popup-confirmer (make-instance
                        'capi:text-input-pane
                        :callback 'capi:exit-confirmer)
                       "Enter some text:"
                       :value-function
                       'capi:text-input-pane-text)
```

See also

popup-confirmer
display-dialog
interface
10 Dialogs: Prompting for Input

exit-dialog *Function*

Summary

Exits the current dialog.

Package

capi

Signature

exit-dialog *value*

Arguments

value⇓ A Lisp object.

Description

The function **exit-dialog** is the means to successfully return a value *value* from the current dialog. Hence, it might be called from an **OK** button so that pressing the button would cause the dialog to return successfully, while the **Cancel** button would call the counterpart function **abort-dialog**.

If there is no current dialog then **exit-dialog** does nothing and returns **nil**. If there is a current dialog then **exit-dialog** either returns non-nil or does a non-local exit. Therefore code that depends on **exit-dialog** returning must be written carefully - see the discussion under **abort-dialog** for details.

Examples

```
(capi:display-dialog
  (capi:make-container
    (make-instance 'capi:text-input-pane
                   :callback-type :data
                   :callback 'capi:exit-dialog)
    :title "Test Dialog"))
```

There is another example in:

```
(example-edit-file "capi/dialogs/simple-dialog")
```

See also

abort-dialog
display-dialog
popup-confirmer
interface
10 Dialogs: Prompting for Input

expandable-item-pinboard-object *Class*

Summary

A class used to implement nodes in **graph-pane**.

Package

capi

Superclasses

item-pinboard-object

Description

The class **expandable-item-pinboard-object** is a **pinboard-object** that **graph-pane** uses by default to implement nodes in a graph.

expandable-item-pinboard-object draws itself with a small circle to indicate that the node has children.

See also

graph-pane
12.3 Creating graphical objects

extended-selection-tree-view *Class*

Summary

A pane that displays a hierarchical list of items which (unlike **tree-view**) allows extended selection.

Package

capi

Superclasses

tree-view

Description

The class **extended-selection-tree-view** is like **tree-view** but allows more than one item to be selected at once.

Notes

1. Although **extended-selection-tree-view** is a subclass of **collection**, it does its own items handling and you must not access its *items* and related slots directly. In particular for **extended-selection-tree-view** do not pass **:items**, **:items-count-function**, **:items-get-function** or **:items-map-function**, and do not use the corresponding accessors.

2. The delete item callback (see *delete-item-callback* in **tree-view**) is called in **extended-selection-tree-view** with the second argument being a list of the selected items, unless *interaction* is **:single-selection**, in which case it behaves the same as in **tree-view**.

See also

tree-view
5 Choices - panes with items

filtering-layout *Class*

Summary

A layout that can be used for filtering.

Package

capi

Superclasses

row-layout

Initargs

:callback-object	The argument for the callbacks. If it is **nil** the top-level-interface of the layout is used.
:change-callback	A function of one argument (the *callback-object*).
:callback	A function of one argument (the *callback-object*).
:gesture-callbacks	Additional *gesture-callbacks* to the **text-input-pane** inside the layout.
:text	A string specifying the initial text of the filter, or **nil**.
:matches-title	A string, **t** or **nil**.
:help-string	A string, **t** or **nil**.
:added-filters	A list of additional filter specifications.
:label-style	**:short**, **:medium** or **:long**.

Accessors

filtering-layout-state
filtering-layout-matches-text

Description

The class **filtering-layout** can be used to display a filter pane for some other pane, such as a **list-panel**.

The main part of a filtering layout is a **text-input-pane** which allows the user to enter a string, which is intended to be used for filtering. The user can control how it is used by a menu (or special keystroke) that allows her to specify whether:

- The string is used as a regular expression or plain string (**Control+R**).
- The filter excludes matches or includes matches (**Control+E**).
- Filtering is case-sensitive or case-insensitive (**Control+C**).

The filtering layout defines the parameters to use, and calls the callbacks to perform the filtering. It does not do any filtering itself.

change-callback is called whenever the text in the filter changes. Also if *callback* is not supplied, then *change-callback* is called instead.

callback is called whenever there is any change in the state of the filter: the user presses **Return**, makes a selection from the menu, clicks the **Confirm** button or changes the selection in any of the added filters. If *callback* is not supplied, then *change-callback* is called instead.

To actually do the filtering, the using code needs to call **filtering-layout-match-object-and-exclude-p**, which returns as multiple values a precompiled regexp and a flag specifying whether to exclude matches. The regexp should be used to perform the filtering, typically by using **lispworks:find-regexp-in-string**. Note that **filtering-layout-match-object-and-exclude-p** returns **nil** when there is no string in the **text-input-pane**, and that even when the filter is set to plain match it returns a regexp (which matches a plain string).

You supply a **filtering-layout** amongst the *panes* of your interface definition (not its *layouts*). The description of a **filtering-layout** is set by the **initialize-instance** method of the class, and therefore the description cannot be passed as an initarg and should not be manipulated.

filtering-layout-state returns a "state" object which can be used later to set the state of any **filtering-layout** by **(setf capi:filtering-layout-state)**. When setting the state, the value can also be a string or **nil**. A string means setting the filter string to it and making the filtering state be plain string, includes matches, and case-insensitive. **nil** means the same as the empty string.

matches-title controls whether the **filtering-layout** contains a **display-pane** (the "matches pane") showing the number of matches. If *matches-title* is a string, it provides the title of the matches pane. If *matches-title* is **t** the title is **Matches:**. Note that the actual text in the matches pane must be set by the caller by **(setf capi:filtering-layout-matches-text)**.

If *help-string* is non-nil then the filter has a Help button which raises a default help text if *help-string* is **t**, or the text of *help-string* if it is a string.

If *label-style* is **:short** the filter menu has a short title. For example if the filter is set for case-sensitive plain inclusive matching the short label is **PMC**. If *label-style* is **:medium** then this label would be **Filter:C**. Any other value of *label-style* would make a long label **Plain Match Cased**.

When *added-filters* is non-nil, it adds panes (**check-button** or **option-pane**) to the **filtering-layout**. Each element of *added-filters* must be one of:

A cons of a string and some object.

> This specifies a **check-button**, with the string as its text, plus an associated object.

A list of conses, where each cons is a cons of a string and some object.

> This specifies an **option-pane**, where the string of each cons specifies the text of an item in the **option-pane**, plus an associated object for the item.

The **check-button** and **option-pane** panes are displayed in the same row as the filter.

The third return value of **filtering-layout-match-object-and-exclude-p** contains the associated objects from each selected **check-button** (but not from any unselected **check-button**) and from the selected item of each **option-pane**.

Notes

A **filtering-layout** is used when a **list-panel** is made with the **:filter** initarg.

Examples

```
(defvar *things* (list "Foo" "Bar" "Baz" 'car 'cdr))

(capi:define-interface my-interface ()
  ((things :reader my-things
           :initform *things*))
  (:panes
   (my-things-list-panel
    capi:list-panel
    :reader my-interface-list-panel
    :items things
    :visible-min-height `(:character ,(length *things*)))
   (my-filtering
    capi:filtering-layout
    :change-callback 'update-my-interface
    :reader my-interface-filtering))
  (:layouts
   (a-layout
    capi:column-layout
    '(my-filtering my-things-list-panel)))
  (:default-initargs :title "Filtering example")
  )

(defun update-my-interface (my-interface)
  (let* ((things (my-things my-interface))
         (filtered-things
          (multiple-value-bind (regexp excludep)
              (capi:filtering-layout-match-object-and-exclude-p
               (my-interface-filtering my-interface)
               nil)
            (if regexp
                (loop for thing in things
                      when (if (find-regexp-in-string
                                regexp
                                (string thing))
                               (not excludep)
                             excludep)
                      collect thing)
              things))))
    (setf (capi:collection-items
           (my-interface-list-panel my-interface))
          filtered-things)))
```

See also

filtering-layout-match-object-and-exclude-p

filtering-layout-match-object-and-exclude-p *Function*

Summary

Returns filtering parameters for a **filtering-layout**.

Package

capi

Signature

`filtering-layout-match-object-and-exclude-p` *filtering-layout display-message* **`=>`** *regexp*, *excludep*, *added-filters-values*

Arguments

filtering-layout⇓	A **`filtering-layout`**.
display-message⇓	A generalized boolean.

Values

regexp	A precompiled regular expression.
excludep⇓	A boolean.
added-filters-values⇓	A list of objects.

Description

The function **`filtering-layout-match-object-and-exclude-p`** returns a regexp to use for filtering in *filtering-layout*.

The second returned value *excludep* specifies whether the filter should be used to exclude or include matches.

The third returned value *added-filters-values* is non-nil when *filtering-layout* has filters added by the initarg **`:added-filters`** (see the documentation for **`filtering-layout`**). *added-filters-values* is a list containing the associated object from each selected **`check-button`** and from the selected item of each **`option-pane`** that were added. Note that *added-filters-values* does not contain anything for any added **`check-button`** that is currently unselected.

display-message is a generalized boolean controlling whether a message is displayed to the user if there is an error when compiling the regexp.

See **`filtering-layout`** for details.

See also

`filtering-layout`

find-graph-edge *Generic Function*

Summary

Finds and returns an edge in a graph given two items.

Package

`capi`

Signature

`find-graph-edge` *graph from to* **`=>`** *edge*

Arguments

graph⇓ A **graph-pane**.

from⇓ An item in *graph*.

to⇓ An item in *graph*.

Values

edge A graph edge, or **nil**.

Description

The generic function **find-graph-edge** finds the edge in *graph* that goes from the node corresponding to *from* to the node corresponding to *to*.

If there is no such edge, **find-graph-edge** returns **nil**.

See also

find-graph-node
graph-pane

find-graph-node *Generic Function*

Summary

Finds and returns a node in a graph corresponding to an item.

Package

capi

Signature

find-graph-node *graph object* **=>** *node*

Arguments

graph⇓ A **graph-pane**.

object⇓ An item in *graph*.

Values

node A node of *graph*, or **nil**.

Description

The generic function **find-graph-node** finds the node in *graph* that corresponds to the item *object*.

If there is no such node, **find-graph-node** returns **nil**.

See also

find-graph-edge
graph-pane

find-interface *Generic Function*

Summary

Displays an interface of a given class, making it if necessary.

Package

capi

Signature

find-interface *class-name* **&rest** *initargs* **&key** *screen* **&allow-other-keys =>** *interface*

Arguments

class-name⇓	A specifier for a subclass of **interface**.
initargs⇓	Initialization arguments for *class-name*.
screen⇓	A **screen** or **nil**.

Values

interface	An interface of class *class-name*.

Description

The generic function **find-interface** finds and displays an interface of the given class *class-name* that matches *initargs* and *screen*.

class-name can be the name of a suitable class, the class itself, or an instance of the class.

screen can be a CAPI object as accepted by **convert-to-screen**. *screen* defaults to the default screen.

find-interface calls **locate-interface** to locate an existing interface:

1. If an interface of the class specified by *class-name* matching *initargs* exists already on *screen*, then this interface is activated and returned.

2. Otherwise, if an interface of the class specified by *class-name* exists already on *screen*, then **reinitialize-interface** is applied to this interface which is then activated and returned.

If no instance of class *class-name* exists on *screen*, then **find-interface** creates one by passing *class-name* and *initargs* to **make-instance**, and displays the result on *screen*.

Notes

There are many uses of **find-interface** in the LispWorks IDE.

See also

locate-interface
reinitialize-interface

find-string-in-collection *Generic Function*

Summary

Returns the next item whose printed representation matches a given string.

Package

capi

Signature

find-string-in-collection *collection string* **&optional** *set*

Arguments

collection⇓	A **collection**.
string⇓	A string.
set⇓	A generalized boolean.

Description

The generic function **find-string-in-collection** returns the next item in *collection* whose printed representation matches *string*. If *set* is true, 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.

See also

collection
collection-find-string
collection-find-next-string
collection-last-search

force-screen-update *Function*

Summary

Ensures a screen is up to date.

Package

capi

Signature

force-screen-update &key *screen*

Arguments

screen⇓ A **screen**.

Description

The function **force-screen-update** makes sure that the **screen** specified by *screen* is up to date.

screen can be a CAPI object as accepted by **convert-to-screen**. The default value of *screen* is **nil**.

Notes

On GTK+, **force-screen-update** does not work when it is called inside the *display-callback* of an **output-pane** or a sub-class, including drawing of **pinboard-object**s inside a **pinboard-layout**.

See also

force-update-all-screens

force-update-all-screens *Function*

Summary

Ensures a screen is up to date.

Package

capi

Signature

force-update-all-screens

Description

The function **force-update-all-screens** makes sure that all screens are up to date.

See also

force-screen-update

foreign-owned-interface *Class*

Summary

Allows another application to own a CAPI dialog.

Package

capi

Superclasses

interface

Description

The class **foreign-owned-interface** allows another application's window to be the owner of a CAPI dialog. Instances should be created by calling **make-foreign-owned-interface**.

foreign-owned-interface is implemented only on Microsoft Windows.

See also

make-foreign-owned-interface

form-layout *Class*

Summary

The class **form-layout** lays its children out in a form.

Package

capi

Superclasses

layout

Initargs

:vertical-gap	The gap between rows in the form.
:vertical-adjust	The adjustment made to the rows.
:title-gap	The gap between the two columns.
:title-adjust	The adjustment made to the left column.

Accessors

form-vertical-gap
form-vertical-adjust
form-title-gap
form-title-adjust

Description

The form layout lays its children out in two columns, where the children in the left column (which are usually titles) are right adjusted while the children in the right column are left adjusted.

Compatibility note

This class has been superseded by **grid-layout**, and will probably be removed at some point in the future. The examples below demonstrate the use of grid layouts as an alternative to forms.

Examples

```
(setq children (list
                "Button:"
                (make-instance 'capi:push-button
                               :text "Press Me")
                "Enter Text:"
                (make-instance 'capi:text-input-pane)
                "List:"
                (make-instance 'capi:list-panel
                               :items '(1 2 3))))
```

```
(capi:contain (make-instance
               'capi:grid-layout
               :description children
               :x-adjust '(:right :left)
               :y-adjust :center))
```

See also

grid-layout
layout

free-metafile *Function*

Summary

Frees a metafile.

Package

capi

Signature

free-metafile *metafile*

Arguments

metafile⇓ A metafile.

Description

The function **free-metafile** releases the window system storage used by *metafile*.

free-metafile must be called when the metafile is no longer needed, to avoid memory leaks.

free-metafile is supported on GTK+ only where Cairo is supported (GTK+ 2.8 and later).

Notes

free-metafile is not implemented on X11/Motif.

Examples

```
(example-edit-file "capi/graphics/metafile")
```

See also

clipboard
draw-metafile
draw-metafile-to-image

free-sound *Function*

Summary

Frees a loaded sound object on Microsoft Windows and Cocoa.

Package

capi

Signature

free-sound *sound*

Arguments

sound⇓	An array returned by **load-sound**.

Description

The function **free-sound** unloads (frees) the loaded sound object *sound*.

Notes

free-sound is not implemented on GTK+ and Motif.

See also

load-sound
read-sound-file
18.2.1 Sound API

get-collection-item *Generic Function*

Summary

Returns the item at a specified position in a collection.

Package

capi

Signature

get-collection-item *collection index*

Arguments

collection⇓	A **collection**.
index⇓	A non-negative integer.

Description

The generic function **get-collection-item** returns the item at position *index* from *collection*. It achieves this by calling the *items-get-function* of *collection*. There is also a complementary function, **search-for-item** which finds the index for a given item in a collection.

See also

collection
search-for-item

get-constraints *Function*

Summary

Returns the external constraints for an element.

Package

capi

Signature

get-constraints *element* **=>** *min-width***,** *min-height***,** *max-width***,** *max-height*

Arguments

element⇓	An instance of **simple-pane** (or one of its subclasses), or an instance of **pinboard-object** (or one of its subclasses).

Values

min-width, *min-height*	Integers specifying the minimum external dimensions of *element*.
max-width, *max-height*	Integers specifying the maximum external dimensions of *element*.

Description

The function **get-constraints** returns the external constraints for *element* as multiple values.

The values are the minimum width, the minimum height, the maximum width and the maximum height of the element including borders. A containing layout will use these values when laying out its children.

get-constraints calls the generic function **calculate-constraints** to calculate these sizes initially, but then just uses the values in the geometry cache for the element. To force an element to take account of its new constraints, call the function

invalidate-pane-constraints.

See also

calculate-constraints
define-layout
element
invalidate-pane-constraints
6 Laying Out CAPI Panes

get-horizontal-scroll-parameters
get-vertical-scroll-parameters

Functions

Summary

Queries the scroll parameters of a horizontal or vertical scroll bar.

Package

capi

Signatures

get-horizontal-scroll-parameters *self* **&rest** *keys* **=>** *parameter***

get-vertical-scroll-parameters *self* **&rest** *keys* **=>** *parameter***

Arguments

self⇓	A displayed **output-pane** or **layout**.
keys⇓	Keywords as below.

Values

*parameter**	The parameters are returned as multiple values, one for each key passed in *keys* and in the same order as the arguments.

Description

The functions **get-horizontal-scroll-parameters** and **get-vertical-scroll-parameters** retrieve the specified parameters of the horizontal or vertical scroll bar of *self*.

self should be a displayed instance of a subclass of **output-pane** (such as **editor-pane**) or **layout** and have a scroll bar.

The valid *keys* are:

:min-range	The minimum data coordinate.
:max-range	The maximum data coordinate.
:slug-position	The current scroll position.
:slug-size	The length of the scroll bar slug.

:page-size The scroll page size.

:step-size The scroll step size.

Notes

For the other pane classes, such as **list-panel**, the underlying widget determines what the scroll range and units are.

Examples

See the following CAPI example files:

```
(example-edit-file "capi/output-panes/scrolling-without-bar")

(example-edit-file "capi/output-panes/fixed-origin-scrolling")

(example-edit-file "capi/output-panes/coordinate-origin-fixed")
```

See also

get-scroll-position
scroll
set-horizontal-scroll-parameters
set-vertical-scroll-parameters
simple-pane
12.4 output-pane scrolling

get-page-area *Function*

Summary

Calculates the dimensions of suitable rectangles for use with **with-page-transform**.

Package

capi

Signature

get-page-area *printer* **&key** *scale* *dpi* *screen*

Arguments

printer⇓ A printer.

scale⇓ A real or **nil**.

dpi⇓ An integer, a list of two integers or **nil**.

screen⇓ A **screen**.

Description

The function **`get-page-area`** is provided to simplify the calculation of suitable rectangles for use with **`with-page-transform`**. It calculates and returns the width and height of the rectangle in the user's coordinate space that corresponds to one printable page on *printer*, based on the logical resolution of the user's coordinate space in dpi.

For example, if a logical resolution of 72 dpi was specified, this means that each unit in user space would map onto 1/72 of an inch on the printed page, assuming that no *scale* is specified.

If *dpi* is **`nil`** (the default), the logical resolution of *screen* is used, or the logical resolution of the default screen if *screen* is **`nil`**. *dpi* can be a number, or a list of two elements representing the logical resolution of the coordinate spaces in the x and y directions respectively.

If *scale* is specified the rectangle is calculated so that the image is scaled by this factor when printed. It defaults to 1.0.

Examples

```
(example-edit-file "capi/printing/fit-to-page")
```

```
(example-edit-file "capi/printing/multi-page")
```

```
(example-edit-file "capi/printing/page-on-demand")
```

See also

`printer-metrics`
`with-page-transform`
16 Printing from the CAPI—the Hardcopy API

get-printer-metrics *Function*

Summary

Returns the metrics for a printer.

Package

`capi`

Signature

`get-printer-metrics` *printer* **`=>`** *metrics*

Arguments

printer⇓ A printer.

Values

metrics⇓ A **`printer-metrics`** object.

Description

The function **get-printer-metrics** returns the metrics of *printer*.

The metrics values in *metrics* should be accessed by the **printer-metrics** readers.

See also

set-printer-metrics
printer-metrics
with-page-transform
16 Printing from the CAPI—the Hardcopy API

get-scroll-position *Generic Function*

Summary

Returns the current scroll position of a pane such as **list-panel**, **display-pane** or **tree-view**.

Package

capi

Signature

get-scroll-position *pane dimension* **=>** *position*

Arguments

pane⇓	A pane with built-in scrolling.
dimension⇓	A keyword, either **:horizontal** or **:vertical**.

Values

position⇓	An integer or **nil**.

Description

The generic function **get-scroll-position** returns the scroll position of the pane *pane* in the given *dimension*.

pane should be an instance of a pane class that has built-in scrolling. That is, the scrolling is implemented by the underlying widget. Examples include **list-panel**, **display-pane** and **tree-view**.

In general, the units in the returned value *position* are unspecified, but they can be passed to the generic function **scroll** with *operation* **:move** to restore the position.

For a **list-panel**, the vertical units are items.

position is **nil** if *pane* is not displayed on the screen, for example if **get-scroll-position** is called after *pane* is destroyed.

See also

get-horizontal-scroll-parameters

get-vertical-scroll-parameters
scroll

graph-edge *Class*

Summary

The class of objects that represent edges in a graph.

Package

capi

Superclasses

graph-object

Initargs

:from	The node where the edge starts.
:to	The node where the edge ends.

Accessors

graph-edge-from
graph-edge-to

Description

The class **graph-edge** represent edges in a **graph-pane**.

from and *to* are the nodes that the edge connects.

See also

graph-pane

graph-node *Class*

Summary

The class of objects that represent nodes in a graph.

Package

capi

Superclasses

graph-object

Readers

graph-node-x
graph-node-y
graph-node-width
graph-node-height
graph-node-in-edges
graph-node-out-edges

Description

The class **graph-node** is the default class of nodes in a **graph-pane**.

The **graph-pane** generates a graph of **graph-node** and **graph-edge** objects.

See also

graph-edge
graph-pane

graph-node-children *Generic Function*

Summary

Returns the children of a graph node.

Package

capi

Signature

graph-node-children *node* **=>** *result*

Arguments

node⇓	A **graph-node**.

Values

result	A list.

Description

The generic function **graph-node-children** returns a list of all the 'children' of the node *node*. These children are the nodes which are at the other end of some edge in the **graph-node-out-edges** of the **graph-node** *node*.

See also

graph-node

graph-object *Abstract Class*

Summary

The superclass of node and edge objects.

Package

capi

Superclasses

standard-object

Subclasses

graph-edge
graph-node

Readers

graph-object-element
graph-object-object

Description

The abstract class **graph-object** is the superclass of **graph-edge** and **graph-node**.

The reader **graph-object-element** returns the CAPI object that is displayed.

The reader **graph-object-object** returns the user object associated with the graph object.

graph-pane *Class*

Summary

A graph pane is a pane that displays a hierarchy of items in a graph.

Package

capi

Superclasses

simple-pinboard-layout
choice

Subclasses

simple-network-pane

Initargs

:roots The roots of the graph.

:children-function Returns the children of a node.

:layout-function A keyword denoting how to layout the nodes.

:layout-x-adjust The adjust value for the *x* direction.

:layout-y-adjust The adjust value for the *y* direction.

:node-pinboard-class
The class of pane to represent nodes.

:edge-pinboard-class
The class of pane to represent edges.

:node-pane-function
A function to return an element for each node.

:edge-pane-function
A function to return an element for each edge.

Accessors

graph-pane-layout-function
graph-pane-roots

Description

The class **graph-pane** is a pane that displays a hierarchy of items in a graph.

The **graph-pane** calculates the items of the graph by calling the *children-function* on each of its *roots*, and then calling it again on each of the children recursively until no more children are found. The *children-function* gets called with an item of the graph and should return a list of the children of that item.

Each item is represented by a node in the graph.

The *layout-function* tells the graph pane how to lay out its nodes. It can be one these values:

:left-right Lay the graph out from the left to the right.

:top-down Lay the graph out from the top down.

:right-left Lay the graph out from the right to the left.

:bottom-up Lay the graph out from the bottom up.

layout-x-adjust and *layout-y-adjust* act on the underlying layout to decide where to place the nodes. The values should be a keyword or a list of the form **(***keyword n***)** where *n* is an integer. These values of *adjust* are interpreted as by **pane-adjusted-position**. **:top** is the default for *layout-y-adjust* and **:left** is the default for *layout-x-adjust*.

When a graph pane wants to display nodes and edges, it creates instances of *node-pinboard-class* and *edge-pinboard-class* which default to **item-pinboard-object** and **line-pinboard-object** respectively. These classes must be subclasses of **simple-pane** or **pinboard-object**, and there are some examples of the use of these keywords below.

The *node-pane-function* is called to create an element for each node, and by default it creates an instance of *node-pinboard-class*. It gets passed the graph pane and the item corresponding to the node, and should return an instance of a subclass of **simple-pane** or **pinboard-object**. Note that the name of the initarg is a little misleading, as in most cases you should return a **pinboard-object** rather than a pane. If you use your own class which has its own geometry requirements, you should define a **calculate-constraints** method for it, which should use **with-geometry** on the object to set

%min-width% and **%width%** to the desired width, and **%height%** and **%min-height%** to the desired height. See the example in:

```
(example-edit-file "capi/graphics/circled-graph-nodes")
```

edge-pane-function is called to create an element for an edge. The default creates an object of the class specified by *edge-pinboard-class*. If *edge-pane-function* is supplied, it must be a function that takes three arguments: the pane and the two items that are connected by the edge, and must return an element (a **simple-pane** or a **pinboard-object**).

To expand or contract a node, the user clicks on the circle next to the node. An expandable node has an unfilled circle and a collapsible node has a filled circle.

graph-pane is a subclass of **choice**, so for details of its selection handling, see **choice**.

The highlighting of the children is controlled as described for **pinboard-layout**, but for **graph-pane** the default value of *highlight-style* is **:standard**.

Notes

The **output-pane** initarg **:drawing-mode** controls quality of drawing in a **graph-pane**, including anti-aliasing of any text displayed on Microsoft Windows and GTK+.

Compatibility note

In LispWorks 4.3 the double click gesture on a **graph-pane** node always calls the *action-callback*, and the user gesture to expand or collapse a node is to click on the circle drawn alongside the node.

In LispWorks 4.2 and previous versions, the double click gesture was used for expansion and contraction of nodes and the *action-callback* was not always called.

Examples

```
(defun node-children (node)
  (when (< node 16)
    (list (* node 2)
          (1+ (* node 2)))))


(setq graph
      (capi:contain
       (make-instance 'capi:graph-pane
                      :roots '(1)
                      :children-function
                      'node-children)
       :best-width 300 :best-height 400))


(capi:apply-in-pane-process
 graph #'(setf capi:graph-pane-roots) '(2 6) graph)


(capi:contain
 (make-instance 'capi:graph-pane
                :roots '(1)
                :children-function
                'node-children
                :layout-function :top-down)
 :best-width 300 :best-height 400)
```

```
(capi:contain
 (make-instance 'capi:graph-pane
                :roots '(1)
                :children-function
                'node-children
                :layout-function :top-down
                :layout-x-adjust :left)
 :best-width 300 :best-height 400)
```

This example demonstrates a different style of graph output with right-angle edges and parent nodes being adjusted towards the top instead of at the center.

```
(capi:contain
 (make-instance
  'capi:graph-pane
  :roots '(1)
  :children-function 'node-children
  :layout-y-adjust '(:top 10)
  :edge-pinboard-class
  'capi:right-angle-line-pinboard-object)
 :best-width 300
 :best-height 400)
```

This example demonstrates the use of **`:node-pinboard-class`** to specify that the nodes are drawn as push buttons.

```
(capi:contain
 (make-instance
  'capi:graph-pane
  :roots '(1)
  :children-function 'node-children
  :node-pinboard-class 'capi:push-button)
 :best-width 300
 :best-height 400)
```

There are more examples here:

```
(example-edit-file "capi/graphics/*graph*")
```

See also

find-graph-edge
find-graph-node
graph-edge
graph-node
graph-node-children
graph-pane-add-graph-node
graph-pane-delete-object
graph-pane-delete-objects
graph-pane-delete-selected-objects
graph-pane-direction
graph-pane-edges
graph-pane-nodes
graph-pane-object-at-position
graph-pane-select-graph-nodes
graph-pane-update-moved-objects
maximum-moving-objects-to-track-edges
output-pane
1.2.1 CAPI elements
5 Choices - panes with items

12 Creating Panes with Your Own Drawing and Input

graph-pane-add-graph-node *Generic Function*

Summary

Adds a node to a graph.

Package

capi

Signature

graph-pane-add-graph-node *graph-pane object parent-node* **=>** *new-node*

Arguments

graph-pane⇓	A **graph-pane**.
object⇓	An object.
parent-node⇓	A **graph-node**.

Values

new-node	A **graph-node**.

Description

The generic function **graph-pane-add-graph-node** adds a new node in the graph *graph-pane* corresponding to *object*, and links it as a child of *parent-node*.

See also

graph-node
graph-pane

graph-pane-delete-object *Generic Function*

Summary

Removes a node from a graph.

Package

capi

Signature

graph-pane-delete-object *graph-pane object*

Arguments

graph-pane⇓ A **graph-pane**.

object⇓ An object.

Description

The generic function **graph-pane-delete-object** deletes the node corresponding to *object* in the graph *graph-pane*.

See also

graph-node
graph-pane
graph-pane-add-graph-node
graph-pane-delete-objects

graph-pane-delete-objects *Generic Function*

Summary

Removes nodes from a graph.

Package

capi

Signature

graph-pane-delete-objects *graph-pane objects*

Arguments

graph-pane⇓ A **graph-pane**.

objects⇓ A list of objects.

Description

The generic function **graph-pane-delete-objects** deletes the node in the graph *graph-pane* corresponding to each object in the list *objects*.

See also

graph-node
graph-pane
graph-pane-delete-object

graph-pane-delete-selected-objects *Generic Function*

Summary

Removes selected nodes from a graph.

Package

capi

Signature

graph-pane-delete-selected-objects *graph-pane*

Arguments

graph-pane⇓ A **graph-pane**.

Description

The generic function **graph-pane-delete-selected-objects** deletes the currently selected nodes in the graph *graph-pane*.

See also

graph-node
graph-pane
graph-pane-delete-object

graph-pane-direction *Accessor*

Summary

Returns or sets the direction of a graph.

Package

capi

Signature

graph-pane-direction *graph-pane* **=>** *direction*

(setf graph-pane-direction) *direction graph-pane* **=>** *direction*

Arguments

graph-pane⇓ A **graph-pane**.

direction⇓ One of **:forwards** or **:backwards**.

Values

direction⇓ One of **:forwards** or **:backwards**.

Description

The accessor **graph-pane-direction** accesses the direction of the graph *graph-pane*. If the *layout-function* of *graph-pane* is **:top-down** or **:left-right** then *direction* is **:forwards**. Otherwise *direction* is **:backwards**.

The setter **(setf graph-pane-direction)** maintains the dimension of the *layout-function* but potentially reverses its direction.

Examples

```
(setf gp
      (make-instance 'capi:graph-pane
                     :layout-function :top-down))
=>
#<CAPI:GRAPH-PANE [0 items] 20603294>

(setf (capi:graph-pane-direction gp)
      :backwards)
=>
NIL

(capi:graph-pane-layout-function gp)
=>
:TOP-DOWN
```

See also

graph-pane

graph-pane-edges *Function*

Summary

Returns the edges of a graph.

Package

capi

Signature

graph-pane-edges *graph-pane* **=>** *edges*

Arguments

graph-pane⇓ A **graph-pane**.

Values

edges A list.

Description

The function **graph-pane-edges** returns a list of all the **graph-edge** objects in the graph *graph-pane*.

See also

graph-edge
graph-pane

graph-pane-nodes *Function*

Summary

Returns the nodes of a graph.

Package

capi

Signature

graph-pane-nodes *graph-pane* **=>** *nodes*

Arguments

graph-pane⇓ A **graph-pane**.

Values

nodes A list.

Description

The function **graph-pane-nodes** returns a list of all the **graph-node** objects in the graph *graph-pane*.

See also

graph-node
graph-pane

graph-pane-object-at-position *Function*

Summary

Returns the graph object at a given position in a graph.

Package

capi

Signature

`graph-pane-object-at-position` *graph-pane x y* **`=>`** *object*

Arguments

graph-pane⇓ A **`graph-pane`**.

x⇓, *y*⇓ Non-negative numbers.

Values

object A **`graph-object`**, or **`nil`**.

Description

The function **`graph-pane-object-at-position`** returns the **`graph-object`** (either a **`graph-edge`** or a **`graph-node`**) at the coordinates *x, y* in the graph *graph-pane*.

If there is no **`graph-object`** at position *x,y* then **`graph-pane-object-at-position`** returns **`nil`**.

See also

`graph-pane`

graph-pane-select-graph-nodes *Generic Function*

Summary

Selects nodes in a graph according to a predicate.

Package

`capi`

Signature

`graph-pane-select-graph-nodes` *graph-pane predicate*

Arguments

graph-pane⇓ A **`graph-pane`**.

predicate⇓ A function of one argument with boolean result.

Description

The generic function **`graph-pane-select-graph-nodes`** applies *predicate* to all of the **`graph-node`**s in *graph-pane*, and sets the *selected-items* to be the objects corresponding to those nodes for which *predicate* returns a true value.

See also

`choice-selected-items`
`graph-node`

graph-pane

graph-pane-update-moved-objects *Generic Function*

Summary

Updates a graph after the user moves objects.

Package

capi

Signature

graph-pane-update-moved-objects *graph-pane* *objects*

Arguments

graph-pane⇓	A **graph-pane**.
objects⇓	A list.

Description

The generic function **graph-pane-update-moved-objects** is called after some objects in the graph *graph-pane* were moved by a user gesture.

objects is a list containing the objects that were moved.

The primary method updates the geometry of edges connected to the moved objects. You can add non-primary methods to perform other operations at that point.

See also

graph-pane

grid-layout *Class*

Summary

A layout which positions its children on a two dimensional grid.

Package

capi

Superclasses

x-y-adjustable-layout

Subclasses

row-layout
column-layout

Initargs

:columns	The number of columns in the grid.
:has-title-column-p	A boolean specifying whether the first column is a title column.
:orientation	The orientation of the children.
:rows	The number of rows in the grid.
:x-ratios	The ratios between the columns.
:y-ratios	The ratios between the rows.
:x-gap	The gap between each column.
:y-gap	The gap between each row.
:x-uniform-size-p	If **t**, make each of the columns the same size.
:y-uniform-size-p	If **t**, make each of the rows the same size.
:min-column-width	**nil**, or a real number which provides a minimum of the width of each column.
:min-row-height	**nil**, or a real number which provides a minimum of the height of each row.

Accessors

layout-x-ratios
layout-y-ratios
layout-x-gap
layout-y-gap

Description

The class **grid-layout** is a layout which positions its children on a two dimensional grid.

The row and column sizes are controlled by the constraints on their children. For example, the *visible-min-width* of any column is the maximum of the *visible-min-width* in of the children in the column. The size of the layout is controlled by the constraints on the rows and columns.

For **grid-layout** *description* is either a two dimensional array or a list in the order specified by *orientation* (which defaults to **:row**). In the case of a list, one of *columns* or *rows* can be supplied to specify the dimensions (the default is two columns). As well as panes, slot names and strings, *description* may contain the element **nil**, which is interpreted as a special dummy pane with suitable geometry for resizable gaps. This special interpretation of **nil** in the *description* is specific to **grid-layout** and its subclasses.

x-ratios and *y-ratios* control the sizes of the elements in a grid layout.

If *x-ratios* (or *y-ratios*) is a list, then each of its elements control the size of each child relative to the others. If an element in *x -ratios* (or *y-ratios*) is **nil** then the child is fixed at its minimum size. Otherwise the size is calculated as follows:

```
(round (* total ratio) ratio-sum)
```

where *ratio-sum* is the sum of the non-nil elements of *x-ratios* (or *y-ratios*) and *ratio* is the element of ratios corresponding to the child. If this ideal ratio size does not fit the maximum or minimum constraints on the child size, and the constraint means that changing the ratio size would not assist the sum of the child sizes fitting the total space available, then the child is fixed at its constrained size, the child is removed from the ratio calculation, and the calculation is performed again. If *x-ratios* (or *y-ratios*) has fewer elements than the number of children, 1 is used for each of the missing ratios. Leaving *x-ratios* (or *y-ratios*) **`nil`** (the default) causes all of the children to be the same size.

The positions of each pane in the layout can be specified using *x-adjust* and *y-adjust* like every other **`x-y-adjustable-layout`**, except that if there is one value then it is used for all of the panes, whereas if it is a list then each value in the list refers to one row or column. If the list does not contain a value for every row or column then the last value is taken to refer to all of the remaining panes.

Normally, the items in a **`grid-layout`** are arranged to look like a set of columns that are joined horizontally and rows that are joined vertically. All the cells in each column have the same width and all the cells in each row have the same height. The keyword **`:right-extend`** (or **`:bottom-extend`**) can be used to allow an item to span more than one column (or row). The keyword should be placed in the cell of the *description* that you want the item to expand into. For **`:right-extend`**, the cell immediately to the left will be extended to fill both columns in that row. For **`:bottom-extend`**, the cell immediately above will be extended to fill both rows in that column. Note that the item can only be extended if its constraints allow this. Fosr example, a **`push-button-panel`** will not extend by default with **`:bottom-extend`** because its constraints fix its height at its min-height.

If *has-title-column-p* is true, then the items in the description which correspond to the first column are treated specially:

A string — Equivalent to specifying **`(:title`** *string***`)`**

A list of the form **`(:title`** *string* **`.`** *options***`)`**.

Make a title using the given list as initargs. *options* is a plist of options, which can include the keys **`:title-font`**, **`:title-args`**, **`:mnemonic`** or **`:mnemonic-escape`**. See **`titled-object`** for how these are processed.

A list of the form **`(:mnemonic-title`** *string* **`.`** *options***`)`**.

Make a title using the given list as initargs. *string* can contain the mnemonic escape. *options* is a plist of options, which can include the keys **`:title-font`**, **`:title-args`**, or **`:mnemonic-escape`**. See **`titled-object`** for how these are processed.

Notes

Mnemonics are not supported on all platforms.

Examples

```
(capi:contain (make-instance
               'capi:grid-layout
               :description '("1" "2" "3"
                              "4" "5" "6"
                              "7" "8" "9")
               :columns 3))
```

```
(capi:contain (make-instance
               'capi:grid-layout
               :description (list "List:"
                                  (make-instance
                                   'capi:list-panel
                                   :items '(1 2 3))
                                   "Buttons:"
                                  (make-instance
                                   'capi:button-panel
                                   :items '(1 2 3)))))
```

```
(capi:contain (make-instance
               'capi:grid-layout
               :description (list "List:"
                                  (make-instance
                                   'capi:list-panel
                                   :items '(1 2 3))
                                  "Buttons:"
                                  (make-instance
                                   'capi:button-panel
                                   :items '(1 2 3)))
               :x-adjust '(:right :left)
               :y-adjust '(:center :bottom)))
```

```
(capi:contain (make-instance
               'capi:grid-layout
               :description (list "List:"
                                  (make-instance
                                   'capi:list-panel
                                   :items '(1 2 3))
                                  "Buttons:"
                                  (make-instance
                                   'capi:button-panel
                                   :items '(1 2 3)))
               :orientation :column))
```

This example illustrates the special interpretation of **nil** in the *description*:

```
(capi:contain
 (make-instance
  'capi:grid-layout
  :description
  (cdr
   (loop for i below 5
         appending
         (list
          nil
          (make-instance 'capi:simple-pane
                         :background :red
                         :visible-min-width 50
                         :visible-max-width t
                         :visible-min-height 50
                         :visible-max-height t))))
  :columns 3)
 :height 150 :width 150 :title "Resize Me")
```

This example illustrates the use of **:right-extend** and **:bottom-extend** to make cells span multiple columns and rows:

```
(example-edit-file "capi/layouts/extend")
```

There are more examples here:

```
(example-edit-file "capi/applications/")
```

This example is a grid with **:has-title-column-p t** :

```
(example-edit-file "capi/layouts/titles-in-grid")
```

See also

layout

1.2.1 CAPI elements
3.1.4.1 Controlling Mnemonics
6 Laying Out CAPI Panes

hide-interface *Function*

Summary

The function **hide-interface** hides the interface containing a specified pane.

Package

capi

Signature

hide-interface *pane* **&optional** *iconify*

Arguments

pane⇓	A **simple-pane**.
iconify⇓	A generalized boolean.

Description

The function **hide-interface** hides the interface containing *pane* from the screen. If *iconify* is non-nil then it will iconify it, else it will just remove it from the screen. To show it again, use **show-interface**.

The default value of *iconify* is **t**.

See also

interface
show-interface
quit-interface
7.7 Manipulating top-level windows

hide-pane *Function*

Summary

Hides the specified pane.

Package

capi

Signature

hide-pane *pane* **=>** *pane*

Arguments

pane⇓ An instance of **simple-pane** or a subclass.

Values

pane An instance of **simple-pane** or a subclass.

Description

The function **hide-pane** hides the pane *pane*, removing it from the screen. *pane*'s children, if any, are hidden too.

To restore *pane* to the screen, use **show-pane**.

See also

hide-interface
show-pane

highlight-pinboard-object *Function*

Summary

Highlights a specified pinboard object.

Package

capi

Signature

highlight-pinboard-object *pinboard object* **&key** *redisplay* **=>** *was-unhighlighted-p*

Arguments

pinboard⇓ A **pinboard-layout**.

object⇓ A **pinboard-object**.

redisplay⇓ A generalized boolean.

Values

was-unhighlighted-p⇓ A boolean.

Description

The function **highlight-pinboard-object** causes the pinboard object *object* to become highlighted until **unhighlight-pinboard-object** is called on it.

The pinboard object highlighting is drawn according to the *highlight-style* of the **pinboard-layout** *pinboard*.

If *redisplay* is non-nil the highlighting is drawn immediately. The default value for *redisplay* is **t**.

The returned value *was-unhighlighted-p* is true if *object* was unhighlighted before the call.

See also

unhighlight-pinboard-object
draw-pinboard-object-highlighted
pinboard-object
pinboard-layout

image-list *Class*

Summary

An object used to manage the images displayed by tree views and list views.

Package

capi

Superclasses

capi-object

Initargs

:image-width	The width of the images in this image list.
:image-height	The height of the images in this image list.
:image-sets	A list of images or image sets.

Description

The class **image-list** is used to manage the images displayed by **tree-view** and **list-view**.

The initarg **:image-sets** specifies a list. Each item in the list *image-sets* may be one of the following:

A pathname or string — This specifies the filename of a file suitable for loading with **load-image**.

A symbol — The symbol must be a predefined image identifier, or have been registered by means of a call to **register-image-translation**.

An **image** object — For example, as returned by **load-image**.

An **image-set** object — See **image-set** for further details.

Note that image sets are added in their entirety; it is not possible to use image-locators to extract a single image from an image set.

The images added to the image list are numbered in order, starting from zero. An **image-set** containing *n* images contributes *n* images to the image list, and hence consumes *n* consecutive integer indices.

Examples

```
(example-edit-file "capi/choice/tree-view")
```

```
(example-edit-file "capi/choice/extended-selection-tree-view")
```

See also

image-set
load-image
register-image-translation
5.10.4 image-list, image-set and image-locator

image-locator *Type*

Summary

The type of the object that **make-image-locator** creates.

Package

capi

Signature

image-locator

Description

The type **image-locator** is the type of the object that **make-image-locator** creates.

See **make-image-locator** for the details.

See also

make-image-locator
5.10.4 image-list, image-set and image-locator

image-pinboard-object *Class*

Summary

A pinboard object that displays itself as an image.

Package

capi

Superclasses

pinboard-object
titled-object

Initargs

:image	The image to be displayed.

Accessors

image-pinboard-object-image

Description

The class **image-pinboard-object** is a **pinboard-object** that displays itself as an image.

The *image* initarg for an **image-pinboard-object** should either be an **external-image** or any other object accepted by **load-image**. The image displayed in the object can be changed read or changed dynamically using the accessor **image-pinboard-object-image**.

Examples

```
(cd (example-file "capi/"))

(setf image
      (capi:contain
       (make-instance
        'capi:image-pinboard-object
        :image "applications/images/info.bmp")))

(capi:apply-in-pane-process
 (capi:element-parent image)
 #'(setf capi:image-pinboard-object-image)
 "graphics/Setup.bmp" image)

(capi:apply-in-pane-process
 (capi:element-parent image)
 #'(setf capi:image-pinboard-object-image)
 "applications/images/info.bmp" image)

(capi:contain
 (make-instance
  'capi:image-pinboard-object
  :image "graphics/Setup.bmp"
  :title "LispWorks Splashscreen"
  :title-adjust :right
  :title-position :bottom))
```

See also

pinboard-layout
12.3 Creating graphical objects
13.10 Working with images

image-set *Class*

Summary

An object that identifies the location of image.

Package

capi

Superclasses

t

Description

An instance of the class **image-set** is an object that identifies the location of an image. The image is typically a large image to be broken down into sub-images. The sub-images must all have the same size and be positioned side by side.

The following functions are available to create image set objects:

See also

make-general-image-set
make-icon-resource-image-set
make-scaled-image-set
make-scaled-general-image-set
make-resource-image-set
5.10.4 image-list, image-set and image-locator
9 Adding Toolbars

installed-libraries *Function*

Summary

Returns the installed libraries.

Package

capi

Signature

installed-libraries => *libraries*

Values

libraries⇓	A list of library names.

Description

The function **installed-libraries** returns the list of installed CAPI libraries.

A library name is a keyword naming a library.

On Linux, FreeBSD and x86/x64 Solaris platforms, libraries is initially **(:gtk)** but may also include **:motif** if the deprecated "capi-motif" module is loaded.

On Microsoft Windows platforms, currently *libraries* is always **(:win32)**.

On macOS platforms, in the native GUI image *libraries* is always **(:cocoa)**. In the macOS/GTK+ image, libraries is initially **(:gtk)** but may also include **:motif** if the deprecated "capi-motif" module is loaded.

See also

default-library
19.5 CAPI communication with host window system - libraries

install-postscript-printer *Function*

Summary

Installs or modifies a Postscript printer definition.

Package

capi

Signature

install-postscript-printer *name* **&key** *if-exists default savep ppd-file description use-jcl command use-file always-print-to-file orientation installed-options*

Arguments

name⇓	A string.
if-exists⇓	One of **:supersede**, **:error** or **nil**.
default⇓	One of **t**, **nil** or **:when-none**.
savep⇓	A boolean.
ppd-file⇓	A string or pathname.
description⇓	A string, or **:preserve**.
use-jcl⇓	A boolean, or **:preserve**.
command⇓	A string, or **:preserve**.
use-file⇓	A boolean, or **:preserve**.
always-print-to-file⇓	A boolean, or **:preserve**.
orientation⇓	One of **:landscape**, **:portrait** or **:preserve**.
installed-options⇓	An association list, or **:preserve**.

Description

The function **install-postscript-printer** installs or modifies a Postscript printer definition for the given printer name.

This applies only on Motif.

name is a string naming the printer.

if-exists controls what happens if the named printer is already known. The default value is **:supersede**.

default controls whether the default printer is set. The value **t** forces the default printer to be set. The value **:when-none** causes the default printer to be set if there is currently no default. The default value of *default* is **nil**.

savep, if true, causes the printer to be saved for subsequent sessions, by writing a file to the path specified by the first item of

`*printer-search-path*`.

ppd-file, if non-nil, should be a pathname or string specifying the name of a PPD file (PostScript Printer Description File) which comes with the printer and specifies the printer properties. *ppd-file* must be supplied when installing a new printer. The default value is **`nil`**.

All the other arguments provide optional printer information. Each defaults to the value **`:preserve`**, which means that appropriate defaults are used. These correspond to the settings on the dialog displayed by **`printer-configuration-dialog`**. Non-default values are as follows:

description is a string describing the printer.

use-jcl controls whether to use Job Control Language (JCL).

command is the command to execute to print with the printer.

use-file controls how to pass data to the printer. A true value means a file is used, **`nil`** means a pipe is used.

always-print-to-file controls whether printing always goes to a file.

orientation controls the orientation of the output.

installed-options is an association list, with pairs of strings where the **`car`** is an option name and the **`cdr`** is its value. Which options are available and their potential values is defined by the *OpenUI/*CloseUI and *JCLOpenUI/*JCLCloseUI entries in the PPD file.

See also

`printer-configuration-dialog`
`*ppd-directory*`
`*printer-search-path*`
`uninstall-postscript-printer`
16.7 Printing on Motif

interactive-pane *Class*

Summary

An editor with a process reading and processing input, and that collects any output into itself. We are considering deprecating this class - please contact Lisp Support if you use it.

Package

`capi`

Superclasses

`editor-pane`

Subclasses

`listener-pane`
`shell-pane`

Initargs

:top-level-function

The input processing function.

Readers

interactive-pane-stream
interactive-pane-top-level-function

Description

An instance of the class **interactive-pane** is an editor with a process reading and processing input, and that collects any output into itself.

interactive-pane contains its own GUI stream. The *top-level-function* is called once, when the interactive pane is created: it needs to repeatedly take input from the GUI stream and write output to it. The *top-level-function* is called on a separate process from the process that displays the pane and does editor interaction. If the *top-level-function* wants to invoke CAPI functionality, it needs to use **apply-in-pane-process** to ensure it is done on the right process. If the *top-level-function* returns, the process just exits, but the pane itself stays and continues to function as an **editor-pane**.

Note that because the pane is a fully functional **editor-pane**, the user can perform complex operations, and the *top-level-function* should try to cope with it. For example, the user may yank a very large amount of text, or may delete half of the buffer.

The first argument to *top-level-function* is the interface containing the interactive pane. The second argument is the interactive pane itself. The third argument is the GUI stream. The default for *top-level-function* is a function which runs a Lisp listener top-loop.

Notes

The class **listener-pane** is built upon **interactive-pane**. **listener-pane** adds functionality for handling Lisp forms and handles complexities involved with the interaction with the Editor, so it is much easier to use. If you use **interactive-pane** directly please contact Lisp Support.

Compatibility note

This class was named **interactive-stream** in LispWorks 3.2 but has been renamed to avoid confusion (as this class is not a stream but a pane that contains a stream). **interactive-stream** and its accessors **interactive-stream-top-level-function** and **interactive-stream-stream** have now been removed.

Examples

This example assumes there is just one line of output from each command sent to the pipe:

```
(capi:contain
 (make-instance
  'capi:interactive-pane
  :top-level-function
  #'(lambda (interface pane stream)
      (declare (ignore interface pane))
      (with-open-stream (s (sys:open-pipe
                            '("/usr/local/bin/bash")
                            :direction :io))
        (loop
         (progn
           (format stream "primitive xterm$ ")
           (let ((input (read-line stream nil nil)))
```

```
                (if input
                    (progn
                      (write-line input s)
                      (force-output s))
                  (return))))
            (let ((output (read-line s nil nil)))
              (if output
                  (progn
                    (write-line output stream)
                    (force-output stream))
                (return)))))))
     :best-height 300
     :best-width 300)
```

See also

collector-pane
3.9.6 Stream panes

interactive-pane-execute-command *Generic Function*

Summary

Simulates user entry of commands in an **interactive-pane**.

Package

capi

Signature

interactive-pane-execute-command *interactive-pane command* **&key** *command-modification-function editp* **&allow-other-keys**

Arguments

interactive-pane⇓ An **interactive-pane**.

command⇓ A Lisp form.

command-modification-function⇓
A function or **nil**.

editp⇓ A generalized boolean.

Description

The generic function **interactive-pane-execute-command** has the same effect as the user typing the Lisp form *command* into the **interactive-pane** *interactive-pane*, and pressing **Return**.

interactive-pane-execute-command may be called from any process.

If *command-modification-function* is non-nil, it is a function of one argument. It is called with argument *command* in the process in which *interactive-pane* runs. The result of this call is used as the command to enter. The default value of *command-modification-function* is **nil**.

If *editp* is true then the command is left at the end of the pane for the user to edit before pressing **Return**. If *editp* is **nil** then

interactive-pane-execute-command simulates the user pressing **Return**. The default value of *editp* is **nil**.

See also

interactive-pane
listener-pane-insert-value

interface *Class*

Summary

The class **interface** is the top level window class, which contains both menus and a hierarchy of panes and layouts. Interfaces can also themselves be contained within a layout, in which case they appear without their menu bar.

Package

capi

Superclasses

simple-pane
titled-object

Initargs

:title	A string, the title of the interface.
:layout	The layout of the interface.
:menu-bar-items	The items on the menu bar.
:auto-menus	A flag controlling the automatic addition of menu objects.
:create-callback	A callback done on creating the window, before display and user interaction.
:destroy-callback	A callback done on closing the window.
:confirm-destroy-function	A function to verify closing of the window.
:best-x	The best *x* position for the interface.
:best-y	The best *y* position for the interface.
:best-width	The best width of the interface.
:best-height	The best height of the interface.
:geometry-change-callback	A function called when the interface geometry changes.
:activate-callback	A function called when the interface is activated or deactivated.
:iconify-callback	A function called when the interface is iconified or restored.
:override-cursor	A cursor that takes precedence over the cursors of panes inside the interface. Not supported on Cocoa and ignored by **text-input-pane** on GTK+.
:message-area	A boolean determining whether the interface has a message area.
:enable-pointer-documentation	A boolean determining whether Pointer Documentation is enabled. Supported only on Motif.

:enable-tooltips	A boolean determining whether Tooltip Help is enabled.
:help-callback	A function called when a user gesture requests help.
:top-level-hook	A function called around the top level event handler.
:external-border	An integer or **nil**.
:initial-focus	A pane, a symbol naming a pane, or **nil**.
:display-state	One of the keywords **:normal**, **:maximized**, **:iconic** and **:hidden**.
:color-mode	**nil** (the default), a keyword or a string. Only effective on Cocoa.
:color-mode-callback	A function that takes a single argument or **nil**.
:transparency	A real number in the inclusive range [0,1], used on Cocoa, later versions of Microsoft Windows, and GTK+.
:window-styles	A list of keywords, or **nil**.
:toolbar-items	A list of items for the toolbar.
:toolbar-states	A toolbar state plist.
:default-toolbar-states	A toolbar state plist.
:pathname	A pathname designator.
:drag-image	**nil**, **t** or an image specifier (that is, a value acceptable as the *id* argument of **load-image**).

Accessors

interface-title
pane-layout
interface-menu-bar-items
interface-create-callback
interface-destroy-callback
interface-confirm-destroy-function
interface-geometry-change-callback
interface-activate-callback
interface-iconify-callback
interface-override-cursor
interface-message-area
interface-pointer-documentation-enabled
interface-tooltips-enabled
interface-help-callback
top-level-interface-color-mode-callback
top-level-interface-external-border
top-level-interface-transparency
interface-toolbar-items
interface-toolbar-states
interface-default-toolbar-states
interface-pathname
interface-drag-image

Readers

interface-window-styles

Description

Every interface can have a title *title* which when it is a top level interface is shown as a title on its window, and when it is contained within another layout is displayed as a decoration (see the class **titled-object** for more details).

The argument *layout* specifies a layout object that contains the children of the interface. To change this layout you can either use the writer **pane-layout**, or you can use the layout **switchable-layout** which allows you to easily switch the currently visible child.

The argument *menu-bar-items* specifies a list of menus to appear on the interface's menu bar.

auto-menus defaults to **t**, which means that an interface may have some automatic menus created by the environment in which it is running (for example the **Works** menu in the LispWorks IDE). To switch off these automatic menus, pass **:auto-menus nil**.

Note: On Cocoa, certain system menu commands such as **Edit > Start Dictation** are added automatically. *auto-menus* does not control this.

When you have an instance of an interface, you can display it either as an ordinary window or as a dialog using respectively **display** and **display-dialog**. The CAPI calls *create-callback* (if supplied) with the interface as its single argument, after all the widgets have been created but before the interface appears on screen. Then to remove the interface from the display, you use **quit-interface** and either **exit-dialog** or **abort-dialog** respectively. When the interface is about to be closed, the CAPI calls the *confirm-destroy-function* (if there is one) with the interface, and if this function returns non-nil the interface is closed as if by calling **destroy**. Once the interface is closed, the *destroy-callback* is called with the interface. Therefore, neither *confirm-destroy-function* nor *destroy-callback* should call **destroy**.

Note: *create-callback* should be used only for operations that must be done with the interface already created and cannot be done in **interface-display**. Otherwise they should be either done in **initialize-instance** or between your calls to **make-instance** and **display**. An operation that needs to run after the interface is created but just before displaying the interface as an ordinary window (typical cases are font queries and loading images) can be put in the **interface-display :before** method. An operation that needs to run just after displaying the interface as an ordinary window can be put in the **interface-display :after** method.

The interface also accepts a number of hints as to the size and position of the interface for when it is first displayed. The arguments *best-x* and *best-y* specify the position, while the arguments *best-width* and *best-height* specify the size. The values can be any hints accepted by **:visible-max-width** and **:visible-max-height** for elements (see **6.4.2 Hint values formats**), except for the character, string or text based hints. If *best-width* or *best-height* is **nil** or not specified, then the interface is displayed at its minimum size based on its constraints.

Whether or not an interface window is resizable is indicated as allowed by the window system. For non-resizable windows on Cocoa the interface window's maximize button is disabled and the resize indicator is not shown, and on Microsoft Windows the maximize box is disabled.

geometry-change-callback may be **nil**, meaning there is no callback. This is the default value. Otherwise *geometry-change-callback* is a function of five arguments: the interface and the geometry. Its signature is:

geometry-change-callback *interface x y width height*

x and *y* are measured from the top-left of the screen rectangle representing the area of the primary monitor (the primary screen rectangle).

activate-callback may be **nil**, meaning there is no callback. This is the default value. Otherwise *activate-callback* is a function of two arguments: the interface and a boolean *activatep* which is true on activation and false on deactivation. Its signature is:

activate-callback *interface activatep* *iconify-callback* may be **nil**, meaning there is no callback. This is the default value. Otherwise *iconify-callback* is a function of two arguments: the interface and a boolean *iconify* which is true when *interface* is iconified and false when it is restored. Its signature is:

iconify-callback *interface iconifyp*

override-cursor, if non-nil, specifies a cursor that is used instead of the cursor of each pane inside the interface. The default value of *override-cursor* is **nil**. See below for an example of setting and unsetting the override cursor. *override-cursor* is not supported on Cocoa. *override-cursor* is ignored by **text-input-pane** on GTK+.

If *message-area* is true, then the interface is created with a message area at the bottom. The text of the message area can be accessed using the **titled-object** accessor **titled-object-message**. The default value of *message-area* is **nil**.

enable-pointer-documentation is a boolean controlling whether Pointer Documentation is enabled. The default value is **t**. The actual action is done by the *help-callback*. *enable-pointer-documentation* is supported only on Motif. It is possible to implement equivalent functionality for **output-pane** and subclasses such as **pinboard-layout** by using the *focus-callback* of **output-pane**.

enable-tooltips is a boolean controlling whether Tooltip Help is enabled. The default value is **t**. The actual action is done by the *help-callback*.

help-callback may be **nil**, meaning there is no callback. This is the default value. Otherwise *help-callback* is a function of four arguments: the interface, the pane inside interface where help is requested, the type of help requested, and the help key of the pane. Its signature is:

help-callback *interface pane type help-key*

Here *type* can be one of:

:tooltip
A tooltip is requested. The function needs to return a string to display in the tooltip, or **nil** if no tooltip should be displayed.

:help
The function should display a detailed, asynchronous help. This value is passed when the user presses the **F1** key (not implemented on Cocoa). **:help** is also passed when the user clicks the '?' box in the title bar of a Microsoft Windows dialog with window style **:contexthelp** (see *window-styles* below).

:pointer-documentation-enter
The cursor entered the pane. The function should set the pointer documentation. This is only supported on Motif.

:pointer-documentation-leave
The cursor left the pane. The function needs to reset the pointer documentation. This is only supported on Motif.

help-key is the *help-key* of *pane*, as described in **element**. There is an example illustrating *help-callback* in:

```
(example-edit-file "capi/elements/help")
```

and there is another example below.

top-level-hook can be used on Microsoft Windows and Motif to specify a hook function that is called around the interface's top level event handler. The hook is passed two arguments: a continuation function (with no arguments) and the interface. The hook must call the continuation, which normally does not return. *top-level-hook* is designed especially for error handling (see below for an example). It can also be used for other purposes, for instance to bind special variables around the top level function. **:top-level-hook** is not supported on Cocoa.

external-border controls how close to the edge of the screen the interface can be placed with explicit positioning using the *best-x*, *best-y*, *best-height* and *best-width* initargs or implicit positioning when a dialog is centered within its owner. The value **nil** allows the window to be anywhere, on or off the screen. The value 0 allows the window can be anywhere on the screen. If *external-border* is a positive integer then the window can be anywhere within *external-border* pixels from the edge of the screen. If *external-border* is a negative integer then the window be anywhere on the screen or up to *external-border* pixels off the edge of the screen. This does not affect whether the use can move the window after it has been displayed. It also does not affect the default positioning of interfaces, where the window system chooses the position.The default value of *external-border* is 0.

initial-focus specifies a pane which has the input focus when the interface is first displayed. See **pane-initial-focus** for more information about the initial focus pane.

display-state controls the initial display of the interface window, as described for **top-level-interface-display-state**.

color-mode controls the visual appearance the interface window, as described for **top-level-interface-color-mode**. Only effective on Cocoa.

If *color-mode-callback* is non-nil, it is called, with the interface as a single argument, when the global color mode (the Appearance on Cocoa, the Theme on Windows and GTK+) may have changed. It may take any action that is useful. Note that *color-mode-callback* may be called sometimes where there is no actual change.

transparency is the overall transparency of the whole interface, where 0 is fully transparent and 1 is fully opaque. This has no effect on whether the user can click on the window. This is implemented for Cocoa and Microsoft Windows. It also works on GTK+, provided that GTK+ and the X server support it. On GTK+ it is supported in version 2.12 and later. The X server needs compositing manager to do it.**:transparency** should only be used for top-level interfaces.

window-styles is a list of keywords controlling various aspects of the top level window's appearance and behavior. Each keyword is supported only on the Window systems explicitly mentioned below.

The following keywords apply to ordinary windows:

:no-geometry-animation

Cocoa: Programmatic changes to window geometry happen without animation.

:hides-on-deactivate-window

Cocoa: The window is only visible when the application is the current application.

Microsoft Windows and GTK+: The window is only visible when it is the active window.

:toolbox Cocoa, Microsoft Windows and GTK+: A window with a small title bar. This window style is used in **docking-layout**.

:borderless Cocoa, Microsoft Windows, GTK+ and Motif: A window with no external decoration or frame.

:internal-borderless

Cocoa and Motif: Remove the default border between the window's edge and its contents.

Microsoft Windows: Remove the default border between the window's edge and its contents for dialogs.

:never-iconic Cocoa, Microsoft Windows, GTK+ and Motif: The window cannot be minimized.

:movable-by-window-background

Cocoa and Microsoft Windows: The user can move the window by grabbing at any point not in an inner pane.

:shadowed Cocoa: Force a shadow on windows with window style **:borderless**. (Other windows have a shadow by default.)

Windows XP (and later): The window has a shadow.

:shadowless Cocoa: The window has no shadow.

:textured-background

Cocoa: The window has a textured background (like the Finder).

:always-on-top Cocoa, Microsoft Windows and GTK+: The window is always above all other windows. Such a window is also known as a windoid.

:ignores-keyboard-input

Cocoa and GTK+: The window cannot be given the focus for keyboard input.

:no-character-palette

Cocoa: The **Special Characters...** menu item is not inserted automatically. (This menu item is added to the **Edit** menu by default.)

:motion-events-without-focus

Cocoa: **output-pane**s in the window will see **:motion** input model events even if the output pane does not have the focus. This is the same behavior as on Microsoft Windows.

:can-full-screen Cocoa: The window can be made full screen (only supported on macOS 10.7 and later).

The following keywords are supported in *window-styles* when the interface is displayed as a dialog:

:resizable Microsoft Windows: The dialog has a border to allow resizing. (Generally Windows dialogs do not allowing resizing.)

:contexthelp Microsoft Windows: A '?' box appears in the window's title bar that sends *help-callback* type **:help**.

If *toolbar-items* is non-nil, then the interface will have a toolbar, which is typically displayed at the top of the window. The value of *toolbar-items* is a list of objects of type **toolbar-button**, **toolbar-component** or **simple-pane**, which are items that might be shown on the toolbar. The set of visible items, their order and their appearance is determined by the current *toolbar-state*, which can be changed if the user customizes the toolbar interactively. Each **toolbar-button** or **simple-pane** in the *toolbar-items* list (including those within a **toolbar-component**) should have a *name* that is not **cl:eql** to any other item in the list. Each **toolbar-button** should have *image* and *text* specified, to control the image and title that is shown for the item. Each **simple-pane** should have *toolbar-title* specified, to control the title that is shown for the item.

toolbar-states is a plist containing information about the state of the toolbar. The user can also change this by customizing the toolbar, so you cannot assume that the value will be the same each time you read it. See **interface-toolbar-state** for a description of the keys and values in this plist.

default-toolbar-states is a plist containing information about the default state of the toolbar, which you can provide as the suggested toolbar state for the interface. The key **:items** will be used in the **Customize Toolbar** dialog as the "default" set of toolbar buttons. If both *default-toolbar-states* and *toolbar-states* are supplied, then the value of any key in *toolbar-states* takes precedence over that of the same key in *default-toolbar-states*. See **interface-toolbar-state** for a description of the keys and values in this plist.

pathname specifies the interface pathname. You can get and set this with the accessor **interface-pathname**. The pathname may be displayed in some way to the user, depending on the GUI library.

Currently, only Cocoa uses *pathname*, in two ways:

- It makes the interface display a drag image on the title bar (This is the same image that is set by **interface-drag-image**, and the *drag-image* takes precedence if it not **nil**). The user can drag from the drag image, and if there is no *drag-callback* or if the *drag-callback* returns **:default** it will drag the pathname as a one item in a **:filenames-list**. For information about *drag-callback*, see **simple-pane**'s description of **:drag-callback** and **simple-pane-drag-callback**.
- The context menu (invoked by right-mouse-click) on the drag image or on the title raises a menu containing the components of the path. Selecting a component opens the Finder with it.

drag-image is currently only effective on Cocoa. A non-nil value specifies that the **interface** should have a drag image, which on Cocoa is a small image (16x16px) to the left of the window title.

When the user drags this image, if the **interface** has a *drag-callback* it is called and if this returns non-nil LispWorks performs drag-and-drop with the image. See **simple-pane** for details of the *drag-callback*.

It is possible to have the image for aesthetic purposes only by supplying *drag-image* and not specifying a *drag-callback*. When *drag-callback* is non-nil, it can dynamically decide whether to allow a dragging, or to disallow dragging (by returning **nil**).

The image specification can be an already converted image (made by **load-image**, **convert-external-image**, **make-sub-image** or **make-image-from-port**). The image will be freed automatically when the interface is destroyed or when *drag-image* is set by **(setf interface-drag-image)**. Otherwise the system uses **load-image** to create a new image, which is also freed automatically.

The value **t** for *drag-image* is interpreted specially: it means display some image. If *drag-image* is set to **t** after an image has already been set, it just displays the previous image. This is useful if an image was displayed but then removed by **(setf interface-drag-image)** with **nil**. If there was no previous image, a default image is displayed.

Notes

1. *create-callback* can only be used for actions that are part of the creation of the pane, that is preparing the pane for display. The *create-callback* is called before the pane is actually displayed, and therefore cannot interact with the user.
2. On Microsoft Windows **F1** always calls *help-callback* if it is non-nil.
3. **(setf capi:interface-message-area)** has an effect only before display. After display, this writer has no effect unless the interface is destroyed and re-created.
4. Even though **interface** is a subclass of **titled-object**, the accessor **titled-object-message-font** cannot be used to get and set the font of the interface's message.
5. On Cocoa in the presence of a **cocoa-default-application-interface**, an **interface** with no menus of its own and with **:auto-menus nil** uses the menu bar from the application interface.

Compatibility note

interface-iconize-callback is deprecated. Use the synonym **interface-iconify-callback** instead.

Examples

```
(capi:display (make-instance 'capi:interface
                             :title "Test Interface"))


(capi:display (make-instance
                'capi:interface
                :title "Test Interface"
                :destroy-callback
                  #'(lambda (interface)
                              (capi:display-message
                               "Quitting ~S"
                               interface))))


(capi:display (make-instance
                'capi:interface
                :title "Test Interface"
                :confirm-destroy-function
                #'(lambda (interface)
                            (capi:confirm-yes-or-no
                             "Really quit ~S"
                             interface))))


(capi:display (make-instance
                'capi:interface
                :menu-bar-items
                  (list
                    (make-instance 'capi:menu
                                   :title "Menu"
                                   :items '(1 2 3)))
                :title "Menu Test"))


(setq interface
      (capi:display
       (make-instance
        'capi:interface
        :title "Test Interface"
        :layout
        (make-instance 'capi:simple-layout
                       :description
                       (list (make-instance
                              'capi:text-input-pane
                              :text "Text Pane"))))))

(capi:execute-with-interface interface
 #'(setf capi:pane-layout) (make-instance
                            'capi:simple-layout
                            :description
                            (list (make-instance
                                   'capi:editor-pane
                                   :text "Editor Pane")))


interface)
(capi:display
(make-instance
 'capi:interface
 :title "Test"
 :best-x 200
 :best-y 200
 :best-width '(/ :screen-width 2)
 :best-height 300))
```

The following forms illustrate the use of *help-callback*:

```
(capi:define-interface my-interface ()
  ()
  (:panes
   (a-pane
    capi:text-input-pane
    :help-key 'input)
   (another-pane
    capi:display-pane
    :help-key 'output
    :text "some text"))
  (:menu-bar a-menu)
  (:menus
   (A-menu
    "A menu"
    (("An item" :help-key "item 1")
     ("Another item" :help-key "item 2"))
    :help-key "a menu"))
  (:layouts
   (main-layout
    capi:column-layout
    '(a-pane another-pane)))

  (:default-initargs
   :help-callback 'my-help-callback
   :message-area t))

(defun do-detailed-help (interface)
  (capi:contain
   (make-instance
    'capi:display-pane
    :text "Detailed help for my interface")
   :title
   (format nil "Help for ~a"
           (capi:capi-object-name interface))))

(defun my-help-callback (interface pane type key)
  (declare (ignore pane))
  (case type
    (:tooltip (if (eq key 'input)
                  "enter something"
                (when (stringp key) key)))
    (:pointer-documentation-enter
     (when (stringp key)
       (setf (capi:titled-object-message interface)
             key)))
    (:pointer-documentation-leave
     (setf (capi:titled-object-message interface)
           "Something else"))
    (:help (do-detailed-help interface ))))

(capi:display
 (make-instance 'my-interface :name "Helpful"))
```

The following forms illustrate the use of *override-cursor* to set and then remove an override cursor.

Create an interface with panes that have various different cursors. Move the pointer across each pane.

```
(setf interface
      (capi:element-interface
       (car
        (capi:contain
         (loop for cursor
               in '(:crosshair :hand :v-double-arrow)
```

```
                     collect
                     (make-instance 'capi:editor-pane
                                    :cursor cursor
                                    :text
                                    (format nil "~A CURSOR"
                                            cursor)))))))
```

Override the pane cursors by setting the override cursor on the interface, and move the pointer across each pane again.

```
(setf (capi:interface-override-cursor interface)
      :i-beam)
```

Remove the override cursor.

```
(setf (capi:interface-override-cursor interface)
      :default)
```

This example illustrates *top-level-hook*. Evaluate this form and then get an error by the interrupt gesture in the editor pane. (For example, the interrupt gesture is **Meta+Control+C** on Motif and **Control+Break** on Microsoft Windows). Then select the Destroy Interface restart.

```
(capi:display
 (capi:make-container
  (make-instance
   'capi:editor-pane)
  :top-level-hook
  #'(lambda (func interface)
      (restart-case (funcall func)
        (nil ()
             :report
             (list "Destroy Interface ~a" interface)
             (capi:destroy interface))))))
```

For an example of using *color-mode* and *color-mode-callback*, see:

```
(example-edit-file "capi/applications/interface-color-mode")
```

This example illustrates the use of *toolbar-items*:

```
(example-edit-file "capi/applications/simple-symbol-browser")
```

See also

layout
switchable-layout
menu
display
display-dialog
interface-display
quit-interface
define-interface
activate-pane
titled-object
interface-document-modified-p
interface-toolbar-state
interface-customize-toolbar
top-level-interface-display-state
top-level-interface-color-mode

top-level-interface-dark-mode-p
1.2.1 CAPI elements
2 Getting Started
3.3.2.1 Window titles
3.12.2 Tooltips for collections, elements and menu items
6 Laying Out CAPI Panes
9 Adding Toolbars
11 Defining Interface Classes - top level windows
12 Creating Panes with Your Own Drawing and Input
13 Drawing - Graphics Ports
17 Drag and Drop

interface-customize-toolbar *Function*

Summary

Displays a window that allows the user to customize an interface toolbar.

Package

capi

Signature

interface-customize-toolbar *interface*

Arguments

interface⇓ A CAPI interface.

Description

The function **interface-customize-toolbar** displays a window owned by the interface *interface* that allows the user to customize the interface toolbar of *interface*.

See **9 Adding Toolbars** for information on how to specify an interface toolbar.

Notes

interface must be displayed at the time **interface-customize-toolbar** is called.

See also

interface
9 Adding Toolbars

interface-display *Generic Function*

Summary

The function called to display an interface on screen.

Package

capi

Signature

interface-display *interface*

Arguments

interface⇓ An instance of a subclass of **interface**.

Description

The generic function **interface-display** is called by **display** to display an interface on screen.

The primary method for **interface** actually does the work. You can add **:before** methods on your own interface classes for code that needs to be executed just before the interface appears, and **:after** methods for code that needs to be executed just after the interface appears.

interface-display is useful when you need to make changes to the interface which require it to be already be created. Font queries and loading images are typical cases.

Notes

1. **interface-display** is called in the process of *interface*.
2. **interface-display** is not called when *interface* is displayed as a dialog. Another way to run code before it appears on screen is to supply a *create-callback* for *interface*.

Examples

This example shows how **interface-display** can be used to set the initial selection in a choice whose items are computed at display-time:

```
(capi:define-interface my-tree ()
  ((favorite-color :initform :blue))
  (:panes
   (tree
    capi:tree-view
    :roots '(:red :blue :green)
    :print-function
    'string-capitalize))
  (:default-initargs
   :width 200
   :height 200))

(defmethod capi:interface-display :after
  ((self my-tree))
```

```
    (with-slots (tree favorite-color) self
      (setf (capi:choice-selected-item tree)
            favorite-color)))

(capi:display (make-instance 'my-tree))
```

See also

display
interface
7 Programming with CAPI Windows
13 Drawing - Graphics Ports

interface-display-title *Function*

Summary

Returns the interface title to use on screen.

Package

capi

Signature

interface-display-title *interface* **=>** *string*

Arguments

interface⇓ A CAPI **interface**.

Values

string A string.

Description

The function **interface-display-title** returns the title to use when displaying the interface *interface* on screen.

This is equivalent to:

```
(capi:interface-extend-title
 interface
 (capi:interface-title interface))
```

See also

interface-extend-title
set-default-interface-prefix-suffix

interface-document-modified-p *Accessor*

Summary

Gets and sets the document-modified flag in the interface.

Package

capi

Signature

interface-document-modified-p *interface* **=>** *value*

(setf interface-document-modified-p) *value interface* **=>** *value*

Arguments

interface⇓	A CAPI interface.
value	A boolean.

Values

value	A boolean.

Description

The accessor **interface-document-modified-p** gets and sets the document-modified flag in the interface *interface*.

Currently this only has a visible effect on Cocoa, where an interface whose document is modified is flagged by adding a dark dot in the middle of its Close button (the red button at top-left of the window).

On other platforms the document-modified state is merely remembered.

See also

interface
11.5.3 Indicating a changed document

interface-editor-pane *Generic Function*

Summary

Finds an **editor-pane** in an interface.

Package

capi

Signature

interface-editor-pane *interface* **=>** *pane*

Arguments

interface⇓ An instance of a subclass of **interface**.

Values

pane An **editor-pane** or **nil**.

Description

The generic function **interface-editor-pane** finds the first pane of *interface* that is an **editor-pane**, and returns it.

If there is no **editor-pane**, then **interface-editor-pane** returns **nil**.

interface-editor-pane may be useful when you need to apply an editor command in the process of some "random" interface, in which case you can use **call-editor** with the result of **interface-editor-pane** (if it is not **nil**).

See also

call-editor
editor-pane
interface

interface-extend-title *Generic Function*

Summary

Calculates the complete interface title.

Package

capi

Signature

interface-extend-title *interface title* **=>** *string*

Arguments

interface⇓ A CAPI **interface**.
title⇓ A string.

Values

string⇓ A string.

Description

The generic function **interface-extend-title** is called by LispWorks with an interface *interface* and its title *title* before actually displaying the title on the screen. The result must be a string, which is actually displayed. There is no requirement for any relation between *title* and the result.

The return value *string* is the title to display on the screen.

The default method uses the values set by **set-default-interface-prefix-suffix**. You can specialize **interface-extend-title** to get other effects.

See also

interface-display-title
set-default-interface-prefix-suffix
3.3.2.1 Window titles
11.5 Controlling the appearance of the top level window

interface-geometry *Generic Function*

Summary

Returns the geometry of an interface. This function is deprecated. Use **top-level-interface-geometry** instead.

Package

capi

Signature

interface-geometry *interface* **=>** *geometry*

Arguments

interface⇓ An instance of a subclass of **interface**.

Values

geometry A list.

Description

The generic function **interface-geometry** returns a list representing the geometry of *interface* in pixel values.

This function is deprecated. Use **top-level-interface-geometry** instead.

See also

top-level-interface-geometry

interface-iconified-p *Function*

Summary

The predicate for whether an interface is iconified.

Package

capi

Signature

interface-iconified-p *pane* **=>** *iconifiedp*

Arguments

pane⇓	A CAPI element.

Values

iconifiedp	A boolean.

Description

The function **interface-iconified-p** returns **t** if the top level interface containing *pane* is iconified. An interface is iconified when its display state as returned by **top-level-interface-display-state** is **:iconic**. This means that the window is visible as an icon, also referred to as minimized.

If the top level interface is not iconified, then **interface-iconified-p** returns **nil**.

See also

hide-interface
top-level-interface
top-level-interface-display-state

interface-keys-style *Generic Function*

Summary

Determines the emulation for an interface.

Package

capi

Signature

interface-keys-style *interface* **=>** *keys-style*

Arguments

interface⇓ An instance of a subclass of **interface**.

Values

keys-style⇓ A keyword, **:pc**, **:emacs** or **:mac**.

Description

The generic function **interface-keys-style** returns a keyword indicating a keys style, or *emulation*. It is called when *interface* starts running in a new process, and *keys-style* determines how user input is interpreted by output panes (including **editor-pane**) in *interface*.

The editor (that is, instances of **editor-pane** and its subclasses) responds to user input gestures according to one of three basic models.

When *keys-style* is **:emacs**, the editor emulates GNU Emacs. This value is allowed on all platforms.

When *keys-style* is **:pc**, the editor emulates standard Microsoft Windows keys on Windows, and KDE/Gnome keys on GTK+ and Motif. This value is allowed in the Windows, GTK+ and X11/Motif implementations.

When *keys-style* is **:mac**, the editor emulates macOS editor keys. This value is allowed only in the macOS Cocoa implementation.

The most important differences between the styles are in the handling of the **Alt** key on Microsoft Windows, selected text, and accelerators:

:emacs **Alt** is interpreted on Microsoft Windows as the Meta key (used to access many Emacs commands).

The modifier **:meta** is used in an **output-pane** *input-model* gesture specification.

Control characters such as **Ctrl+S** are not interpreted as accelerators.

The selection is not deleted on input.

:pc **Alt** is interpreted as **Alt** on Microsoft Windows and can be used for shortcuts.

The modifier **:meta** is not used in an **output-pane** *input-model* gesture specification.

Control keystrokes are interpreted as accelerators. Standard accelerators are added for standard menu commands, for example **Ctrl+S** for **File > Save**. For the full set of standard accelerators see **8.7.1 Standard default accelerators**.

The selection is deleted on input, and movement keys behave like a typical Microsoft Windows or KDE/Gnome editor.

:mac Emacs **Control** keys are available, since they do not clash with the Macintosh **Command** key.

The selection is deleted on input, and movement keys behave like a typical macOS editor.

By default *keys-style* is **:pc** on Microsoft Windows platforms and **:emacs** on other platforms. You can supply methods for **interface-keys-style** on your own interface classes that override the default methods.

In the Cocoa implementation, **Command** keystrokes such as **Command+X** are available if there is a suitable **Edit** menu, regardless of the Editor emulation.

See the chapter "Emulation" in the *Editor User Guide* for more detail about the different styles.

Notes

On Motif the code to implement accelerators and mnemonics clashes with the LispWorks meta key support. Therefore the keyboard must be configured so that none of the keysyms connected to mod1 (see xmodmap) are listed in the variable **capi-motif-library:*meta-keysym-search-list***, which must be also be non-nil. Note also that Motif requires Alt to be on mod1.

See also

editor-pane

---

interface-match-p *Generic Function*

Summary

Determines whether an interface is suitable for displaying initargs.

Package

capi

Signature

interface-match-p *interface* **&rest** *initargs* **&key &allow-other-keys =>** *matchp*

Arguments

interface⇓	An instance of a subclass of **interface**.
initargs⇓	Initargs for *interface*.

Values

matchp	A boolean.

Description

The generic function **interface-match-p** returns a true value if *interface* is suitable for displaying the initargs *initargs*.

interface-match-p is used by **locate-interface**. When there is an existing interface for which **interface-match-p** returns true, then **locate-interface** returns it.

The default method for **interface-match-p** always returns **nil**. You can add methods for your own interface classes.

See also

locate-interface

interface-menu-groups *Generic Function*

Summary

Used when an embedded document sets the *menu-bar-items* to its menus, on Microsoft Windows.

Package

`capi`

Signature

`interface-menu-groups` *interface* **`=>`** *result*

Arguments

interface⇓ A CAPI **`interface`**.

Values

result A list.

Description

The generic function **`interface-menu-groups`** is called when an embedded document sets the menu bar of its containing interface *interface*. It is called when an embedded object uses the **`IOleInPlaceFrame::InsertMenus`** method to add menus from the interface to its own composite menu, which is used as the menubar while the embedded object is active.

The menu bar for the embedded document includes three groups of menus that are supplied by the container (file-group, view-group, windows-group). **`interface-menu-groups`** is used to define these groups of menus.

`interface-menu-groups` should return a list of length 3. Each element is a list of menus. In this list, each item is either a menu object, or a cons. When it is a cons, the car is a menu object and the cdr is a string, which overrides the the title of the menu.

The default method, specialied on **`interface`**, simply returns **`(nil nil nil)`**.

Notes

`interface-menu-groups` is implemented only in LispWorks for Windows. Load the functionality by **`(require "embed")`**.

See also

`ole-control-pane`

interface-preserve-state *Generic Function*

Summary

Called before an interface is destroyed during session saving.

Package

capi

Signature

interface-preserve-state *interface*

Arguments

interface⇓ An **interface**.

Description

The generic function **interface-preserve-state** is called by **hcl:save-current-session** just before it destroys *interface*. It is called in the process that runs *interface*. You can specialize this for your own interface classes. Your methods should not interact with the user or other external sources, and should not interact with other processes, because it is called after **hcl:save-current-session** already started to destroy interfaces.

The return value is not used.

The default method does nothing.

See also

interface-preserving-state-p
7.7.6 Preserving information when saving an IDE session

interface-preserving-state-p *Function*

Summary

The predicate for whether an interface is in "preserving-state" context.

Package

capi

Signature

interface-preserving-state-p *interface* **=>** *result*

Arguments

interface⇓ An **interface**.

Values

result⇓ **nil**, **t**, **:different-invocation** or **:keeping-processes**.

Description

The function **interface-preserving-state-p** returns information about whether *interface* is in the "preserving-state" context. An interface enters "preserving-state" context just before it is destroyed by **hcl:save-current-session**, and exits the context just after **interface-display** returns.

If the interface *interface* is in "preserving-state" context, then *result* is either **t** or **:different-invocation**. The value **t** means that the current invocation of LispWorks is still the same invocation. The value **:different-invocation** means it is a different invocation, in other words it is the saved image that is restarted.

In other circumstances **interface-preserving-state-p** can return **:keeping-processes**, which means that the interfaces are destroyed but processes that are not associated with *interface* are not killed. That currently happens only on Microsoft Windows when the programmer changes the arrangement of IDE windows via **Preferences... > Environment > General > Window Options**.

Otherwise *result* is **nil**.

interface-preserving-state-p is typically used in the *destroy-callback* of an interface or a pane to decide whether really to destroy the information, and in the *create-callback* or **interface-display** to decide whether the existing information can be used. Note that if it is a pane, it needs to find the **top-level-interface**.

Information that is made entirely of Lisp objects can be preserved in all cases. Information that is associated with external objects is invalid when the image is restarted. So when **interface-preserving-state-p** is used inside the *create-callback* or **interface-display**, external information can be preserved only if it returns **t**. When **interface-preserving-state-p** returns **t**, the external information may be preserved, unless it is tied to the lightweight process.

See also

interface
interface-display
interface-preserve-state
7.7.6 Preserving information when saving an IDE session

interface-reuse-p *Generic Function*

Summary

Determines whether an interface is suitable for re-use.

Package

capi

Signature

interface-reuse-p *interface* **&rest** *initargs* **&key &allow-other-keys =>** *reusep*

Arguments

interface⇓ An instance of a subclass of **interface**.

initargs⇓ Initargs for *interface*.

Values

reusep A boolean.

Description

The generic function **interface-reuse-p** returns a true value if *interface* is suitable for reuse with *initargs*.

interface-reuse-p is used by **locate-interface** if no matching interface is found first by **interface-match-p**. In this case, when there is an interface for which **interface-reuse-p** returns true, then **locate-interface** reinitializes it by **reinitialize-interface** and returns it.

Notes

interface-reuse-p should not be confused with **reuse-interfaces-p**, which determines the global re-use state.

See also

interface-match-p
locate-interface

interface-toolbar-state *Accessor*

Summary

Reads or changes the properties of an interface toolbar that give information about its state.

Package

capi

Signature

interface-toolbar-state *interface key* **=>** *value*

(setf interface-toolbar-state) *value interface key* **=>** *value*

Arguments

interface⇓ An instance of **interface** or a subclass.

key⇓ One of the *toolbar-states* plist keys.

value The value associated with the *toolbar-states* plist key.

Values

value The value associated with the *toolbar-states* plist key.

Description

The accessor **interface-toolbar-state** reads or changes the properties of the interface toolbar of *interface* that give information about its state. The user can also change these properties by customizing the toolbar, so you cannot assume that the value will be the same each time you read it.

See **9 Adding Toolbars** for information on how to specify an interface toolbar.

key can be one of the following, with the corresponding value:

:visible — *visible* is true if the toolbar is visible and false if it is hidden. The default is true.

:items — *items* is a list of the names of the *toolbar-items* which are shown on the toolbar, in the order they are shown. The built-in names **:separator**, **:space** and **:flexible-space** represent various kinds of gap between items. On Microsoft Windows, an item can be a list of the form **(:titled-separator** *title***)** which starts a dockable group of items that displays *title* when it is undocked. The default *items* includes all items in *toolbar-items*, with **:separator** between each **toolbar-component**.

:display — *display* is a keyword describing what is displayed for each item. It can be **:image** (just shows an image), **:title** (just shows the title), **:image-and-title** (shows both title and image) or **:image-and-title-horizontal** (shows title and image horizontally, only supported on GTK+). The default is platform-specific.

:size — *size* is a keyword describing the size of the items. It can be one of **:small**, **:normal** or **:large**. Some of these sizes might be the same as others. The default is platform-specific.

You can set all of the keys simultaneously by setting the **interface-toolbar-state** accessor or providing the *toolbar-states* initarg.

Notes

The value **:separator** in *items* may or may not actually be visible, depending on the windowing system. On macOS Lion it is zero width.

See also

interface
interface-customize-toolbar
9 Adding Toolbars

interface-visible-p *Function*

Summary

The predicate for whether the interface containing a pane is visible.

Package

capi

Signature

interface-visible-p *pane* **=>** *visiblep*

Arguments

pane⇓ A CAPI pane.

Values

visiblep A boolean.

Description

The function **interface-visible-p** returns **nil** if one of the following is true:

1. *pane* is not associated with any interface.
2. *pane* is associated with an interface which is not displayed.
3. *pane* is associated with an interface which is minimized or iconified.
4. *pane* is known to be fully obscured by other windows. This can happen on Motif, but is not detected on Microsoft Windows.

An error is signalled if *pane* is not a CAPI pane (that is, it is not an instance of a subclass of **element**, **collection** or **pinboard-object**).

Otherwise **interface-visible-p** returns **t**.

Notes

On Microsoft Windows, **interface-visible-p** may return **t** even though the interface is entirely obscured by another window.

interpret-description *Generic Function*

Summary

Converts an abstract description of a layout's children into a list of objects.

Package

capi

Signature

interpret-description *layout description interface* **=>** *result*

Arguments

layout⇓ A **layout**.

description⇓ A list, or other Lisp object accepted for some layout class.

interface⇓ An **interface**.

Values

result A list, each element being a **simple-pane**, a **pinboard-object** or a geometry object.

Description

The generic function **interpret-description** is used by the layout mechanism to translate an abstract description of *layout*'s children (supplied by the initarg **:description** or **(setf layout-description)**) into a list of objects to actually use. Each object must be either an element (an object of type **simple-pane** or of type **pinboard-object**) or a geometry object (the result of the default method of **parse-layout-descriptor**). *interface* is the interface of *layout*.

The default method specialized on **layout** expects *description* to be a list, and returns a list of the values returned by **parse-layout-descriptor** for each element. Some built-in subclasses of **layout** have their own methods, which allow different values of *description*. In these cases the manual page for the layout class describes what *description* can be.

For example, **column-layout** expects as its description a list of items where each item in the list is either the slot-name of the child or a string which should be turned into a title pane. This is the default handling of a layout's description, which is done by calling the generic function **parse-layout-descriptor** to do the translation for each item.

You can define a method for your own layout class. The elements in the returned list must not be returned more than once for layouts that are displayed at the same time.

See also

parse-layout-descriptor
define-layout
layout
6 Laying Out CAPI Panes

invalidate-pane-constraints *Function*

Summary

Causes the resizing of a pane if its minimum and maximum size constraints have changed. It returns **t** if resizing was necessary.

Package

capi

Signature

invalidate-pane-constraints *pane*

Arguments

pane⇓ A **simple-pane**.

Description

The function **invalidate-pane-constraints** informs the CAPI that the constraints (its minimum and maximum size) of *pane* may have changed. The CAPI then checks this, and if the pane is no longer within its constraints it resizes it so that it is and then makes the pane's parent layout lay its children out and display them again at their new positions and sizes. If the pane is resized, then **invalidate-pane-constraints** returns **t**.

See also

get-constraints
layout
element
define-layout
6 Laying Out CAPI Panes

invoke-command *Function*

Summary

Invokes a command in the input model for a specified output pane.

Package

capi

Signature

invoke-command *command* *output-pane* **&rest** *event-args*

Arguments

command⇓	A Lisp object defined as a command by **define-command**.
output-pane⇓	An **output-pane**.
event-args⇓	A list of appropriate arguments for the event that *command* is associated with.

Description

The function **invoke-command** invokes *command* with arguments *event-args* in the input model of *output-pane*, with the translator being called to process the gesture information. To avoid the translation, use **invoke-untranslated-command**.

See also

invoke-untranslated-command
define-command
output-pane
12.2.2 Commands - aliases

invoke-untranslated-command *Function*

Summary

Invokes a command in the input model for a specified output pane, without the translator being called.

Package

capi

Signature

invoke-untranslated-command *command output-pane* **&rest** *event-args*

Arguments

command⇓	A Lisp object defined as a command by **define-command**.
output-pane⇓	An **output-pane**.
event-args⇓	A list of appropriate arguments for the event that *command* is associated with.

Description

The function **invoke-untranslated-command** invokes *command* with arguments *event-args* in the input model of *output-pane*, without the translator being called to process the gesture information. To perform the translation, use **invoke-command**.

See also

invoke-command
define-command
output-pane
12.2.2 Commands - aliases

item *Class*

Summary

The class **item** groups together a title, some data and some callbacks into a single object for use in collections and choices.

Package

capi

Superclasses

callbacks
capi-object

Subclasses

menu-item
button
item-pinboard-object
popup-menu-button
toolbar-button

Initargs

:collection	The collection in which item is displayed.
:data	The data associated with the item.
:text	The text to appear in the item (or **nil**).
:print-function	If *text* is **nil**, this is called to print the data.

:selected If **t** the item is selected.

Accessors

item-collection
item-data
item-text
item-print-function
item-selected

Description

An item can provide its own callbacks to override those specified in its enclosing *collection*, and can also provide some data to get passed to those callbacks.

An item is printed in the collection by **print-collection-item**. By default this returns a string using the item's *text* if specified, or else calls a print function on the item's *data*. The *print-function* will either be the one specified in the item, or else the *print-function* for its parent collection.

The *selected* slot in an item is non-nil if the item is currently selected. The accessor **item-selected** is provided to access and to set this value.

Examples

```
(defun main-callback (data interface)
  (capi:display-message "Main callback: ~S"
                        data))

(defun item-callback (data interface)
  (capi:display-message "Item callback: ~S"
                        data))

(capi:contain (make-instance
               'capi:list-panel
               :items (list
                        (make-instance
                         'capi:item
                         :text "Item"
                         :data '(some data)
                         :selection-callback
                          'item-callback)
                        "Non-Item 1"
                        "Non-Item 2")
               :selection-callback 'main-callback))
```

See also

itemp
collection
choice
print-collection-item
9 Adding Toolbars

itemp *Function*

Summary

A predicate for **item**.

Package

capi

Signature

itemp *object* **=>** *result*

Arguments

object⇓ A Lisp object.

Values

result A boolean.

Description

The function **itemp** returns true if *object* is an **item** and false otherwise. It is equivalent to:

```
(typep object 'capi:item)
```

See also

item
collection

item-pane-interface-copy-object *Generic Function*

Summary

Determines what **pane-interface-copy-object** returns from a **choice**.

Package

capi

Signature

item-pane-interface-copy-object *item choice interface* **=>** *object*, *string*, *plist*

Arguments

item⇓ A Lisp object that is one of the items of *choice*.

choice⇓ A **choice** within *interface*.

interface⇓ An **interface**.

Values

object A Lisp object.

string A string.

plist⇓ A plist.

Description

The generic function **item-pane-interface-copy-object** is used by the method of **pane-interface-copy-object** that specializes on **choice** to decide what to return.

item is one of the items of *choice*, which is a **choice** within *interface*.

If only one item is selected, the **pane-interface-copy-object** method for **choice** returns what **item-pane-interface-copy-object** returns for this item. In this case all three of the return values are used.

If multiple items are selected, **pane-interface-copy-object** applies **item-pane-interface-copy-object** to each one, and returns a list of the returned objects as the first value, and a concatenation of returned strings (separated by newlines) as the second value. *plist* is ignored if the there more than one element.

The default method returns the item and its print representation (using the *print-function* of the **choice**), and no third return value.

You can define your own methods for **item-pane-interface-copy-object**. This is useful to make **active-pane-copy** work properly for a **choice**, in cases where the actual items in the choice are not the objects that are displayed in the choice as far as the user is concerned. For example, you may have a structure:

```
(defstruct my-item
  real-object
  color)
```

To give different colors to different lines in a **list-panel**. In this case **pane-interface-copy-object** (and hence **active-pane-copy** when the **list-panel** is active) will return the **my-item** structure, while the user will expect the real object. This can be fixed by adding a method:

```
(defmethod item-pane-interface-copy-object
           ((item my-item) pane interface)
  (let ((real-object (my-item-real-object item)))
    (values real-object
            (print-a-real-object real-object))))
```

See also

pane-interface-copy-object
active-pane-copy
7.6 Edit actions on the active element

item-pinboard-object *Class*

Summary

A **pinboard-object** that displays a single piece of text.

Package

capi

Superclasses

pinboard-object
item

Description

The class **item-pinboard-object** displays an item on a pinboard layout. It displays the text specified by the item in the usual way (either by the text field, or through printing the data with the print function).

Examples

```
(capi:contain (make-instance
                'capi:item-pinboard-object
                :text "Hello World"))
```

```
(capi:contain (make-instance 'capi:item-pinboard-object
                             :data :red
                             :print-function
                               'string-capitalize))
```

See also

image-pinboard-object
pinboard-layout
12.3 Creating graphical objects

labelled-arrow-pinboard-object *Class*

Summary

A **pinboard-object** that displays an arrow with a label on it.

Package

capi

Superclasses

arrow-pinboard-object
labelled-line-pinboard-object

Description

The class **`labelled-arrow-pinboard-object`** displays an arrow with a label on it on a **`pinboard-layout`**.

Examples

See **`labelled-line-pinboard-object`**.

See also

`pinboard-layout`
12.3 Creating graphical objects

labelled-line-pinboard-object *Class*

Summary

A subclass of **`pinboard-object`** which draws a labelled line.

Package

`capi`

Superclasses

`item-pinboard-object`
`line-pinboard-object`

Subclasses

`labelled-arrow-pinboard-object`

Initargs

`:text-foreground`	A valid color specification, as defined for the **`graphics-state`** parameter *foreground*.
`:text-background`	A valid color specification, as defined for the **`graphics-state`** parameter *foreground*, or the keyword **`:background`**, or **`nil`**.

Accessors

`labelled-line-text-foreground`
`labelled-line-text-background`

Description

The class **`labelled-line-pinboard-object`** displays a line on a **`pinboard-layout`** and draws a label in the middle of it.

Note that the label text is inherited from **`item`**.

text-foreground defines the color of the label text.

text-background defines the background for the text, which is the color used to draw a filled rectangle in the area of the text before drawing the text. The value **`:background`** means use the *background* of the **`pinboard-layout`** of the object. The value **nil** means do not draw a background rectangle. The default value of *text-background* is **:background**.

Notes

For a description of color specifications, see **15.1 Color specs**.

Examples

```
(capi:contain
 (make-instance
  'capi:pinboard-layout
  :description
  (list (make-instance
         'capi:labelled-line-pinboard-object
         :text "Labelled Line"
         :start-x 10 :start-y 10
         :end-x 80 :end-y 60)
        (make-instance
         'capi:labelled-arrow-pinboard-object
         :text "Labelled Arrow"
         :start-x 10 :start-y 70
         :end-x 80 :end-y 120
         :head-direction :both))))
```

See also

graphics-state
pinboard-layout
12.3 Creating graphical objects

layout *Class*

Summary

A pane that positions one or more child panes within itself according to a layout policy.

Package

capi

Superclasses

titled-object
simple-pane

Subclasses

simple-layout
grid-layout
pinboard-layout

Initargs

:default	A flag to mark the default layout for an interface.
:description	The list of the layout's children.

:initial-focus A child of the layout, or its *name*, specifying where the input focus should be, or **nil**.

Accessors

layout-description

Description

The class **layout** is a pane that positions one or more child panes within itself according to a layout policy.

description is an abstract description of the children of the layout, and each layout defines its format. Generally, *description* is a list, each element of which is one of:

- An element, that is an object of type **simple-pane** or **pinboard-object**.
- A slot name, where the name refers to a slot in the layout's interface containing an element.
- A string, where the string gets converted to a **title-pane** or an **item-pinboard-object**.

Note that **pinboard-object**s can be used only when the hierarchy contains **pinboard-layout**.

Some subclasses of **layout** have different syntax for *description*, for example **grid-layout** (and its subclasses **row-layout** and **column-layout**) allows arrays too, and it also accepts **nil** in the *description* list.

Setting the layout description causes the layout to translate it, and then to layout the new children, adjusting the size of its parent if necessary. The actual translation is done by **interpret-description**.

A number of default layouts are provided which provide the majority of layout functionality that is needed. They are as follows:

simple-layout A layout for one child.

row-layout Lays its children out in a row.

column-layout Lays its children out in a column.

grid-layout Lays its children out in an n by m grid.

pinboard-layout Places its children where the user specifies.

switchable-layout Keeps only one of its children visible.

initial-focus specifies which child of the layout has the input focus when the layout is first displayed. Panes are compared by **cl:eq** or **capi-object-name**. See **pane-initial-focus** for more information about the initial focus pane..

Notes

In most cases, a **layout** does not have its own native GUI object. You can force it to have its own native GUI object by supplying the initargs **:background :background**. You need to do that if you want to make a **layout** without a background initially, and change it later using **(setf simple-pane-background)**.

See also

define-layout
interpret-description
6 Laying Out CAPI Panes

line-pinboard-object *Class*

Summary

A subclass of **pinboard-object** which displays a line drawn between two corners of the area enclosed by the pinboard object.

Package

capi

Superclasses

pinboard-object

Subclasses

arrow-pinboard-object
right-angle-line-pinboard-object

Initargs

:start-x	The x coordinate of the start of the line.
:start-y	The y coordinate of the start of the line.
:end-x	The x coordinate of the end of the line.
:end-y	The y coordinate of the end of the line.

Description

The class **line-pinboard-object** displays a line drawn between two corners of the area enclosed by the pinboard object.

start-x, *start-y*, *end-x* and *end-y* default to values computed from the *x*, *y*, *width* and *height*. They are used to compute the size of the object, and the proper value of *x* and *y*. Note that *width* and *height* may be larger, for example to accommodate the label in a **labelled-line-pinboard-object**, and the *x* and *y* are adjusted for that.

To change the end points of the line, call **move-line**.

A complementary class **right-angle-line-pinboard-object** is provided which draws a line around the edge of the pinboard object.

Examples

```
(capi:contain
 (make-instance
  'capi:line-pinboard-object
  :start-x 0 :end-x 100
  :start-y 100 :end-y 0))
```

See also

move-line
pinboard-layout

12.3 Creating graphical objects

line-pinboard-object-coordinates *Function*

Summary

Returns the coordinates of a **line-pinboard-object**.

Package

capi

Signature

line-pinboard-object-coordinates *object* **=>** *start-x*, *start-y*, *end-x*, *end-y*

Arguments

object⇓	A **line-pinboard-object**.

Values

start-x	An integer.
start-y	An integer.
end-x	An integer.
end-y	An integer.

Description

The function **line-pinboard-object-coordinates** returns the start and end coordinates of the **line-pinboard-object** *object*.

See also

move-line

listener-pane *Class*

Summary

An editor that accepts Lisp forms, entered by the user at a prompt, which it then evaluates and displays any output and results.

Package

capi

Superclasses

interactive-pane

Description

An instance of the class **`listener-pane`** is an editor that accepts Lisp forms, entered by the user at a prompt, which it then evaluates. All of the output that is sent to **`*standard-output*`** is sent to the listener, and finally the results of the evaluation are displayed.

Examples

```
(capi:contain (make-instance 'capi:listener-pane)
              :best-width 300 :best-height 200)
```

See also

`collector-pane`
`interactive-pane`
3.9.6 Stream panes

listener-pane-insert-value *Function*

Summary

Evaluates a form and inserts the result in a **`listener-pane`**.

Package

`capi`

Signature

`listener-pane-insert-value` *pane form*

Arguments

pane⇓	A **`listener-pane`**.
form⇓	A Lisp form.

Description

The function **`listener-pane-insert-value`** evaluates the form *form* and inserts the result in the **`listener-pane`** *pane*, as if it resulted from user input. The result is printed, and the values of the history variables **`*`**, **`**`**, **`***`**, **`/`**, **`//`**, and **`///`** are set.

`listener-pane-insert-value` may be called in any process.

Multiple values in the result of evaluating *form* are not supported: only the first value is inserted in *pane*.

See also

`interactive-pane-execute-command`

list-panel *Class*

Summary

A pane that displays a group of items and provides support for selecting items and performing actions on them. Each item may optionally have an image.

Package

capi

Superclasses

choice
simple-pane
sorted-object
titled-object

Subclasses

list-view
multi-column-list-panel

Initargs

:right-click-selection-behavior
A keyword or **nil**. Controls the behavior on a right mouse button click.

:color-function A function designator or **nil**. Controls item text color on Microsoft Windows, Cocoa and GTK+.

:alternating-background
A boolean influencing the use of alternating background color on Cocoa and GTK+.

:filter A boolean. The default value is **nil**. Only used when *filter* is non-nil.

:the

:filter-automatic-p
A boolean. The default value is **t**. Only used when *filter* is non-nil.

:filter-callback A function designator or the keyword **:default**, which is the default value. Only used when *filter* is non-nil.

:filter-change-callback-p
A boolean. Only used when *filter* is non-nil.

:filter-short-menu-text
A boolean. The default value is **nil**. Only used when *filter* is non-nil.

:filter-matches-title
A string, **t** or **nil**. Only used when *filter* is non-nil.

:filter-help-string
A string, **t** or **nil**. Only used when *filter* is non-nil.

:filter-added-filters
A list of additional filter specifications.

:keyboard-search-callback
A function that is used to search for an item when the user types ordinary characters.

:image-function Returns an image for an item.

:state-image-function
Returns a state image for an item.

:image-lists A plist of keywords and **image-list** objects.

:use-images Flag to specify whether items have images. Defaults to **t**.

:use-state-images Flag to specify whether items have state images. Defaults to **nil**.

:image-width Defaults to 16.

:image-height Defaults to 16.

:state-image-width Defaults to *image-width*.

:state-image-height
Defaults to *image-height*.

:separators One of **nil** (the default), **:horizontal**, **:vertical**, **:both** or **t**.

Accessors

list-panel-right-click-selection-behavior
list-panel-keyboard-search-callback
list-panel-image-function
list-panel-state-image-function

Description

The class **list-panel** gains much of its behavior from **choice**, which is an abstract class that handles items and their selection. By default, a list panel has both horizontal and vertical scrollbars.

list-panel does not support the **:no-selection** interaction style. For a non-interactive list use a **display-pane**.

To scroll a **list-panel**, call **scroll** with *scroll-operation* **:move**.

mnemonic-title is interpreted as for **menu**.

color-function allows you to control the text colors on Microsoft Windows, Cocoa and GTK+. If *color-function* is non-nil, then it is a function used to compute the text color of each item, with signature:

color-function *list-panel item state* **=>** *result*

When *alternating-background* is true, the list panel is drawn with alternating background on Cocoa. On GTK+ it provides a hint, which the theme can override. Experience suggests that theme may draw with alternating background even when *alternating-background* is false, but when it is true they tend to draw it always. The default value of *alternating-background* is **nil**.

state is a keyword representing the state of the item. It can be one of **:normal**, **:selected** or **:disabled**. The value *result* should be a value suitable for the function **convert-color**. The pane uses the converted color as the foreground color for the item *item*. *color-function* is called while *list-panel* is being drawn, so it should not do heavyweight computations.

Description: Filter

If *filter* is non-nil, the system automatically adds a **filtering-layout** above the list. The items in the **list-panel** are filtered by the value in the **filtering-layout**. Filtering displays only those items whose print representation matches the filter. (The print representation is the result of **print-collection-item**, and is what the user sees.) Only the items that match, or those that do not match if **Exclude** is set, are displayed in the **list-panel**.

Here filtering means mapping over the unfiltered items, collecting each item that matches the current setting in the filter, and then setting the items of the **list-panel** to the collected items.

For a **list-panel** with a filter, **collection-items** returns only the filtered items, and the selection (that is, the result of **choice-selection** and the argument to **(setf choice-selection)**) index into the filtered items.

Calling **(setf collection-items)** on a filtered **list-panel** sets an internal unfiltered list, and then clears the filtering so that all items are visible.

To get and set the unfiltered items, use the accessor **list-panel-unfiltered-items**. To access the filter-state, use **list-panel-filter-state**. To access both the unfiltered items and the filter simultaneously, which is especially useful when setting both of them at the same time, use **list-panel-items-and-filter**.

filter-automatic-p controls whether the filter automatically does the filtering whenever the text in the filter changes, and *filter-callback* defines the callback of the **filtering-layout**.

If *filter-automatic-p* is **t**, whenever a change occurs in the filter the list is refreshed against the new value in the filter. The *filter-callback* (if non-nil) is called with two arguments, the **filtering-layout** and the **list-panel** itself, when the user "confirms" (that is, she presses **Return** or clicks the **Confirm** button). If *filter-automatic-p* is false and *filter-callback* is **:default**, then the **filtering-layout** is given a callback that does the filtering when the user "confirms". If *filter-automatic-p* is false and *filter-callback* is non-nil, then no filtering is done explicitly, and it is the responsibility of the callback to do any filtering that is required.

filter-matches-title (default **t**) and *filter-help-string* (default **t**) are passed down to the filtering layout using the **filtering-layout** initargs **:matches-title** and **:help-string** respectively. See **filtering-layout** for a description of these initargs.

If *filter-short-menu-text* is true, the filter menu has a short title. For example if the filter is set for case-sensitive plain inclusive matching the short label is **PMC**. If *filter-short-menu-text* were false then this label would be **Filter:C**.

When *filter-added-filters* is non-nil, it adds additional filters that apply to the items of the **list-panel**. Each element of *filter-added-filters* must be one of:

A cons of a string and a function.

> This specifies a **check-button**, with the string as its text, plus an associated function.

A list of conses, where each cons is a cons of a string and a function.

> This specifies an **option-pane**, where the string of each cons specifies the text of an item in the **option-pane**, plus an associated function for the item. The function can also be **nil**, which means no filtering.

The **check-button** and **option-pane** panes are displayed in the same row as the filter.

Before checking if an item in the **list-panel** matches the filter's text, the filter passes the item to the associated function from each selected **check-button** and from the selected item of each **option-pane** (unless the associated function is **nil**). If any of these functions returns **nil**, then the item is excluded (so it is not displayed). Note that the **Exclude** setting of the filter does not apply to the added filters, and the functions are called with the item in the **list-panel**, rather than its printed representation.

Any change in the selection of any of the **check-button** and **option-pane** panes causes the filter to be applied, which recomputes the displayed items.

There is a simple example of using *filter-added-filters* in:

```
(example-edit-file "capi/choice/filter-added-filters")
```

Notes: Filter

If you use *filter*:

1. You should not rely on the **element-parent** of the **list-panel**, because it is implemented by wrapping some layouts around the **list-panel**.
2. The filter is actually a filtering layout, so it has the same interactive semantics as **filtering-layout**.

Description: Keyboard search

keyboard-search-callback should be a function with signature:

keyboard-search-callback *pane string position* **=>** *index*, *last-match*, *last-match-reset-time*

pane is the **list-panel**, *string* is a string to match and *position* is the item index from which the system thinks that the search should start.

string contains the character that the user typed, appended to the "last match", if there is one. There is a "last match" if the previous call to *keyboard-search-callback* returned it (see below).

index is an index in the **collection-items** to move to. Apart from an integer inside the items range of the **list-panel**, this can be **nil**, which means do nothing, or **:no-change**, which selects the current item.

last-match is a string that should be recorded as the "last match" (if it is not a string, the "last match" is reset). This is prepended to the character in the next call, if the character is typed before the "last match" is reset.

last-match-reset-time is the time to wait before resetting the "last match", in seconds. Once this time passes, the last match is reset to **nil**. If *last-match-reset-time* is **nil**, the default value (which defaults to 1) is used. This default value can be changed by **set-list-panel-keyboard-search-reset-time**.

You can simplify the implementation of *keyboard-search-callback* by using **list-panel-search-with-function**.

As a special case, passing **:keyboard-search-callback t** tells CAPI to use its own internal search mechanism in preference to the native one. That can be useful on GTK+, where the default is to use the native search mechanism (for GTK+ versions after 2.4).

Notes: Keyboard search

keyboard-search-callback is intended for searching, but it is not limited to doing a search, and in fact can be used for implementing other functionality. However, since the system waits for the result, if the callback does something heavy or interacts with the user, it should schedule it in some way and return, for example:

```
(defun my-keyboard-search-callback (pane string pos)
  (declare (ignore pane pos))
  ;; cause a call to display-message in event loop
  (mp:current-process-send
   (list 'capi:display-message
         (format nil "You pressed  ~a" string)))
  nil ; return nil so do nothing
  )
```

Description: Images

The *image-function* is called on an item to return an image associated with the item. It can return one of the following:

A pathname or string — This specifies the filename of a file suitable for loading with **load-image**. Currently this must be a bitmap file.

A symbol — The symbol must have been previously registered by means of a call to **register-image-translation**. It can also one of the following symbols, which map to standard images: **:std-cut**, **:std-copy**, **:std-paste**, **:std-undo**, **:std-redo**, **:std-delete**, **:std-file-new**, **:std-file-open**, **:std-file-save**, **:std-print**, **:std-print-pre**, **:std-properties**, **:std-help**, **:std-find** and **:std-replace**.

On Microsoft Windows, the following symbols are also recognized. They map to view images: **:view-large-icons**, **:view-small-icons**, **:view-list**, **:view-details**, **:view-sort-name**, **:view-sort-size**, **:view-sort-date**, **:view-sort-type**, **:view-parent-folder**, **:view-net-connect**, **:view-net-disconnect** and **:view-new-folder**.

Also on Microsoft Windows, these symbols are recognized. They map to history images: **:hist-back**, **:hist-forward**, **:hist-favorites**, **:hist-addtofavorites** and **:hist-viewtree**.

An **image** object — For example, as returned by **load-image**.

An image locator object

This allowing a single bitmap to be created which contains several button images side by side. See **make-image-locator** for more information. On Microsoft Windows, it also allows access to bitmaps stored as resources in a DLL.

An integer — This is a zero-based index into the list panel's image lists. This is generally only useful if the image list is created explicitly. See **image-list** for more details.

The *state-image-function* is called on an item to determine the state image: an additional optional image used to indicate the state of an item. It can return one of the above, or **nil** to indicate that there is no state image.

If *image-lists* is specified, it should be a plist containing the following keywords as keys. The corresponding values should be **image-list** objects.

:normal — Specifies an **image-list** object that contains the item images. The *image-function* should return a numeric index into this **image-list**.

:state — Specifies an **image-list** object that contains the state images. The *state-image-function* should return a numeric index into this **image-list**.

Description: Right-click selection behavior

right-click-selection-behavior can take the following values:

nil — Corresponds to the behavior in LispWorks 4.4 and earlier. The data is not passed.

All non-nil values pass the clicked item as data to the *pane-menu*:

:existing-or-clicked/restore/discard

If the clicked item is not already selected, make it be the entire selection while the menu is displayed. If the clicked item is already selected, do not change the selection. If the menu is cancelled, the original selection is restored. If the user chooses an item from the menu, the selection is not restored.

:temporary-selection

A synonym for **:existing-or-clicked/restore/discard**.

:existing-or-clicked/restore/restore

If the clicked item is not already selected, make it be the entire selection while the menu is displayed. If the clicked item is already selected, do not change the selection. If the user chooses an item from the menu and the item's callback does not set the selection then the original selection is restored after the callback. If the callback sets the selection, then this selection remains. The original selection is restored if the user cancels the menu.

:temporary-restore A synonym for **:existing-or-clicked/restore/restore**.

:clicked/restore/discard

Make the clicked item be the entire selection while the menu is displayed. If the menu is cancelled, the original selection is restored. If the user chooses an item from the menu, the selection is not restored.

:temporary-always A synonym for **:clicked/restore/discard**.

:clicked/restore/restore

Make the clicked item be the entire selection while the menu is displayed. If the user chooses an item from the menu and the item's callback does not set the selection then the original selection is restored after the callback. If the callback sets the selection, then this selection remains. The original selection is restored if the user cancels the menu.

:existing-or-clicked/discard/discard

If the clicked item is not already selected, make it be the entire selection while the menu is displayed. If the clicked item is already selected, do not change the selection. The original selection is never restored, regardless of whether the user chooses an item from the menu or cancels the menu.

:discard-selection A synonym for **:existing-or-clicked/discard/discard**.

:clicked/discard/discard

Make the clicked item be the entire selection. The original selection is never restored, regardless of whether the user chooses an item from the menu or cancels the menu.

:discard-always A synonym for **:clicked/discard/discard**.

:no-change Does not affect the selection, but the clicked item is nonetheless passed as the data.

The default value of *right-click-selection-behavior* is **:no-change**.

separators controls whether there are separators. Horizontal separators means that each row is separated from the previous row by a horizontal line. Vertical separators are applicable only in **multi-column-list-panel**, and means that by default each column is separated by a vertical line from the previous column. This can be overridden by the **:separator** option in the column specification (see entry for **multi-column-list-panel**). If *separators* is **nil** (the default), there are no separators. **:both** and **t** are equivalent, and specify both horizontal and vertical separators. **:horizontal** and **:vertical** specify separators for one direction.

Examples

```
(setq list (capi:contain
            (make-instance 'capi:list-panel
                           :items '(:red :blue :green)
```

```
                              :selected-item :blue
                              :print-function
                              'string-capitalize)))

(capi:apply-in-pane-process
 list #'(setf capi:choice-selected-item) :red list)

(capi:apply-in-pane-process
 list #'(setf capi:choice-selected-item) :green list)


(capi:contain (make-instance
               'capi:list-panel
               :items '(:red :blue :green)
               :print-function 'string-capitalize
               :selection-callback
                 #'(lambda (data interface)
                           (capi:display-message
                            "~S" data))))
```

This example illustrates the use of **:right-click-selection-behavior**:

```
(capi:define-interface click ()
  ((keyword :initarg :right-click-selection-behavior))
  (:panes
   (list-panel
    capi:list-panel
    :items '("foo" "bar" "baz" "quux")
    :visible-min-height '(:character 4)
    :pane-menu 'my-menu
    :interaction :multiple-selection
    :right-click-selection-behavior keyword)))

(defun my-menu (pane data x y)
  (declare (ignore pane x y))
  (make-instance 'capi:menu
                 :items (list "Hi There"
                              ""
                              "Here's the data:"
                              data)))

(capi:display
   (make-instance 'click
                  :right-click-selection-behavior
                  :clicked/restore/restore))
```

See also this example:

```
(example-edit-file "capi/choice/list-panel-pane-menu")
```

There are further examples here:

```
(example-edit-file "capi/choice/")
```

This example illustrates the use of *color-function*:

```
(example-edit-file "capi/applications/simple-symbol-browser")
```

There are further examples in **20 Self-contained examples**.

See also

button-panel
double-list-panel
1.2.1 CAPI elements
3.1.4.1 Controlling Mnemonics
19.3.2 Matching resources for GTK+
5 Choices - panes with items
7 Programming with CAPI Windows
10.2.3 Prompting for an item in a list
13.10 Working with images
17 Drag and Drop

list-panel-enabled *Accessor Generic Function*

Summary

Gets or sets the enabled state of a **list-panel**. This accessor is deprecated.

Package

capi

Signature

list-panel-enabled *list-panel* **=>** *enabledp*

(setf list-panel-enabled) *enabledp list-panel* **=>** *enabledp*

Arguments

list-panel⇓	A **list-panel**.
enabledp	A boolean.

Values

enabledp	A boolean.

Description

The accessor generic function **list-panel-enabled** gets or sets the enabled state of *list-panel*.

Notes

list-panel-enabled is deprecated because it is equivalent to the **simple-pane** accessor **simple-pane-enabled**. Use **simple-pane-enabled** instead.

See also

simple-pane

list-panel-filter-state *Accessor Generic Function*

Summary

Accesses the state of the filter in a filtered **list-panel**.

Package

capi

Signature

list-panel-filter-state *list-panel* **=>** *filter-state*

(setf list-panel-filter-state) *filter-state list-panel* **=>** *filter-state*

Arguments

list-panel⇓ A **list-panel**.

filter-state⇓ A "state" of a **filtering-layout** or **nil**.

Values

filter-state⇓ A "state" of a **filtering-layout** or **nil**.

Description

The accessor generic function **list-panel-filter-state** accesses the state of the filter in a filtered **list-panel** (that is, a **list-panel** created with *filter* **t**).

list-panel-filter-state returns the state of the filter in *list-panel*. The return value *filter-state* is the same type as the state that is used in **filtering-layout**.

(setf list-panel-filter-state) sets the filter in *list-panel*, filters the unfiltered items and displays those that match the *new-state*. The *new-state* has the same semantics as the *new-value* of **(setf filtering-layout-state)**. It can be a result of a call to **list-panel-filter-state** or to **filtering-layout-state** (on a **filtering-layout**), or a string (meaning plain match, case-insensitive), or **nil** (meaning match everything).

On an unfiltered **list-panel** **list-panel-filter-state** returns **nil**, and **(setf list-panel-filter-state)** does nothing.

See also

list-panel
list-panel-unfiltered-items
filtering-layout

list-panel-items-and-filter *Accessor*

Summary

Accesses the unfiltered items and filter in a **list-panel**.

Package

capi

Signature

list-panel-items-and-filter *list-panel* **=>** *unfiltered-items***,** *filter-state*

setf (list-panel-items-and-filter *list-panel***) (values** *unfiltered-items filter-state***) =>** *unfiltered-items***,** *filter-state*

Arguments

list-panel⇓	A **list-panel**.
unfiltered-items	A sequence.
filter-state⇓	A "state" for a **filtering-layout**.

Values

unfiltered-items	A sequence.
filter-state⇓	A "state" for a **filtering-layout**.

Description

The accessor **list-panel-items-and-filter** accesses the unfiltered items and the state of the filter in the list panel *list-panel* simultaneously. It is especially useful for setting the filter state and the items without flickering.

list-panel-items-and-filter returns the items and filter state in *list-panel* as multiple values. It is equivalent to:

```
(values (list-panel-unfiltered-items list-panel)
        (list-panel-filter-state list-panel))
```

but is more efficient.

The return value *filter-state* is the same type as the state that is used in **filtering-layout**.

The **setf** form of **list-panel-items-and-filter** takes the items and new filter state as two values and sets them in *list-panel*:

These two forms:

```
(setf (list-panel-items-and-filter list-panel)
      (values new-items new-filter-state))
```

```
(progn
  (setf (list-panel-unfiltered-items list-panel) new-items)
```

```
(setf (list-panel-filter-state list-panel) new-filter-state))
```

have the same ultimate effect on *list-panel*, but the latter form will filter *new-items* with the old filter and display the result and then filter *new-items* again with *new-filter-state*, whereas the **setf** form of **list-panel-items-and-filter** filters *new-items* just once, with *new-filter-state*.

See also

list-panel
list-panel-filter-state
list-panel-unfiltered-items

list-panel-search-with-function *Function*

Summary

Searches a **list-panel**.

Package

capi

Signature

list-panel-search-with-function *list-panel function arg* **&key** *start-index wrap-around reset-time*

Arguments

list-panel⇓	A **list-panel**.
function⇓	A function taking two arguments. The first is *arg*, the second is an item in *list-panel*.
arg⇓	Any Lisp object.
start-index⇓	An integer, default 0.
wrap-around⇓	A boolean, default **t**.
reset-time⇓	A real number. The default is an internal value which can be set by **set-list-panel-keyboard-search-reset-time**.

Description

The function **list-panel-search-with-function** searches *list-panel* using *function*. **list-panel-search-with-function** is intended to simplify the implementation of the *keyboard-search-callback* of **list-panel**.

list-panel-search-with-function searches *list-panel* for a match. It applies *function* to each item and *arg*, until *function* returns non-nil.

When *function* returns non-nil, *list-panel-search-with-function* returns three values: the index of the item, *arg*, and *reset-time*.

The search starts at *start-index* if supplied, and at 0 otherwise. When the search reaches the end of the list panel and it did not start from 0, it wraps around to the beginning, unless *wrap-around* is supplied as **nil**. The default value of *wrap-around* is **t**.

Examples

```
(defun string-equal-prefix (string item)
  (let* ((start 0)
         (len (length item))
         (end (+ start (length string))))
    (and (>= len end )
         (string-equal string item
                       :start2 start
                       :end2 end))))

(capi:contain
 (make-instance
  'capi:list-panel
  :items '("ae" "af" "bb" "cc")
  :keyboard-search-callback
  #'(lambda (pane string position)
      (capi:list-panel-search-with-function
       pane
       'string-equal-prefix ; or 'string-not-greaterp
       string
       :start position
       :reset-time 1
       :wrap-around t))))
```

Pressing "a" slowly cycles between "ae" and "af". Running the same example with **string-not-greaterp** instead causes "a" to cycle around all of the items.

See also

list-panel
set-list-panel-keyboard-search-reset-time
5.3.9 Searching by keyboard input

list-panel-unfiltered-items *Accessor Generic Function*

Summary

Accesses the unfiltered items of a filtered **list-panel**.

Package

capi

Signature

list-panel-unfiltered-items *list-panel* **=>** *unfiltered-items*

setf (list-panel-unfiltered-items *list-panel***)** *unfiltered-items* **=>** *unfiltered-items*

Arguments

list-panel⇓	A **list-panel**.
unfiltered-items	A sequence.

Values

unfiltered-items A sequence.

Description

The accessor generic function **`list-panel-unfiltered-items`** accesses the unfiltered items of a filtered **`list-panel`** (that is, a **`list-panel`** created with **`:filter t`**).

`list-panel-unfiltered-items` returns the unfiltered items of *list-panel* (that is all of them, as opposed to the accessor **`collection-items`**, which returns only those items that match the filter).

`(setf list-panel-unfiltered-items)` sets the items of *list-panel* without affecting the filter (as opposed to **`(setf collection-items)`** which resets the filter). The items are then filtered, and only those that match the filter are displayed.

`list-panel-unfiltered-items` behaves the same as **`collection-items`** when called on an unfiltered **`list-panel`**.

See also

`list-panel`
`list-panel-items-and-filter`
`list-panel-filter-state`

list-view *Class*

Summary

The list view pane is a **`choice`** that displays its items as icons and text in a number of formats. Not implemented on Cocoa.

Package

`capi`

Superclasses

`list-panel`

Initargs

`:view` Specifies which view the list view pane shows. The default is **`:icon`**.

`:subitem-function` Returns additional information to be displayed in report view.

`:subitem-print-functions`
Used in report view to print the additional information.

`:image-function` Returns an image for an item.

`:state-image-function`
Returns a state image for an item.

`:image-lists` A plist of keywords and **`image-list`** objects.

`:columns` Defines the columns used in report view.

`:auto-reset-column-widths`
Determines whether columns automatically resize. Defaults to **`:all`**.

:auto-arrange-icons
Determines whether icons are automatically arranged to fit the size of the window.

:use-large-images Indicates whether large icons will be used (generally only if the icon view will be used). Defaults to **t**.

:use-small-images Indicates whether small icons will be used. Defaults to **t**.

:use-state-images Indicates whether state images will be used. Defaults to **nil**.

:large-image-width Width of a large image. Defaults to 32.

:large-image-height
Height of a large image. Defaults to 32.

:small-image-width Width of a small image. Defaults to 16.

:small-image-height
Height of a small image. Defaults to 16.

:state-image-width Width of a state image. Defaults to *small-image-width*.

:state-image-height
Height of a state image. Defaults to *small-image-height*.

Accessors

list-view-view
list-view-subitem-function
list-view-subitem-print-functions
list-view-image-function
list-view-state-image-function
list-view-columns
list-view-auto-reset-column-widths
list-view-auto-arrange-icons

Description

The class **list-view** displays items as icons and text in a number of formats.

list-view inherits its functionality from **choice**. In many ways it may be regarded as a kind of enhanced list panel, although its behavior is not identical. It supports single selection and extended selection interactions.

The list view displays its items in one of four ways, determined by the value in the *view* slot. An application may use the list view pane in just a single view, or may change the view between all four available views using **(setf list-view-view)**.

See the notes below on using both large and small icon views.

In all views, the text associated with the item (the label) is returned by the *print-function*, as with any other choice.

- The icon view — **:icon**.

  In this view, large icons are displayed, together with their label, positioned in the space available. See also *auto-arrange-icons*, below.

- The small icon view — **:small-icon**.

  In this view, small icons are displayed, together with their label, positioned in the space available. See also *auto-arrange -icons*, below.

- The list view — **:list**.

  In this view, small icons are displayed, arranged in vertical columns.

- The report view — **:report**.

  In this view, multiple columns are displayed. A small icon and the item's label is displayed in the first column. Additional pieces of information, known as subitems, are displayed in subsequent columns.

  To use the view **:report**, *columns* must specify a list of column specifiers. Each column specifier is a **plist**, in which the following keywords are valid:

  :title The column heading.

  :width The width of the column in pixels. If this keyword is omitted or has the value **nil**, the width of the column is automatically calculated, based on the widest item to be displayed in that column.

  :align May be **:left**, **:right** or **:center** to indicate how items should be aligned in this column. The default is **:left**. Only left alignment is available for the first column.

If *auto-arrange-icons* is true, then the icons are automatically arranged to fit the size of the window when the view is showing **:icon** or **:small-icon**. The default value of *auto-arrange-icons* is **nil**.

The *subitem-function* is called on the item to return subitem objects that represent the additional information to be displayed in the subsequent columns. Hence, *subitem-function* should normally return a list, whose length is one less than the number of columns specified. Each subitem is then printed in its column using the appropriate subitem print function. *subitem-print-function* may be either a single print function, to be used for all subitems, or a list of functions: one for each subitem column.

Note that the first column always contains the item label, as determined by the *choice-print-function*.

The *image-function* is called on an item to return an image associated with the item. It can return one of the following:

A pathname or string: This specifies the filename of a file suitable for loading with **load-image**. Currently this must be a bitmap file.

A symbol: The symbol must have been previously registered by means of a call to **register-image-translation**.

An **image** object: For example, as returned by **load-image**.

An image locator object: Allowing a single bitmap to be created which contains several button images side by side. See **make-image-locator** for more information. On Microsoft Windows, this also allows access to bitmaps stored as resources in a DLL.

An integer: This is a zero-based index into the list view's image list. This is generally only useful if the image list is created explicitly. See **image-list** for more details.

The *state-image-function* is called on an item to determine the state image, an additional optional image used to indicate the state of an item. It can return one of the above, or **nil** to indicate that there is no state image. State images may be used in any view, but are typically used in the report and list views.

If *image-lists* is supplied, it should be a plist containing the following keywords as keys. The corresponding values should be **image-list** objects.

:normal Specifies an **image-list** object that contains the large item images. The *image-function* should return a numeric index into this image-list.

:small Specifies an **image-list** object that contains the small item images. The *image-function* should return a numeric index into this image-list.

:state Specifies an **image-list** object that contains the state images. The *state-image-function* should return a numeric index into this image-list.

If both the large icon view (icon view) and one or more of the small icon views (small icon view, list view, report view) are to be used, special considerations apply.

The image lists must be created explicitly, using the **:image-lists** initarg, and the *image-function* must return an integer. Take care to ensure that corresponding images in the **:normal** and **:small** image lists have the same numeric index.

Returning pathnames, strings or image-locators from the image function cause the CAPI to create the image-lists automatically; however, if large and small icon views are mixed, this will lead to incorrect icons (or no icons) being displayed in one or other view.

Notes

1. **list-view** is not implemented on Cocoa.
2. For some applications **multi-column-list-panel** will suffice instead of **list-view**.

See also

image-list
list-panel
make-image-locator
multi-column-list-panel
5.10.4 image-list, image-set and image-locator
13.10 Working with images

load-cursor *Function*

Summary

Loads a cursor.

Package

capi

Signature

load-cursor *filename-or-list* **=>** *cursor*

Arguments

filename-or-list⇓ A string or a list.

Values

cursor A cursor object.

Description

The function **load-cursor** loads a cursor from your cursor file, or loads a built-in cursor. It returns a cursor object which can be supplied as the value of the **simple-pane** **:cursor** initarg.

The cursor object can also be set with **(setf simple-pane-cursor)** to change a pane's cursor. This must be done in the process of the pane's interface.

If *filename-or-list* is a string, then it names a file which should be in a suitable format for the platform, as follows:

Microsoft Windows	**.cur** or **.ani** format.
Cocoa	TIFF format.
GTK+	Any image format that **load-image** supports. **Note:** The image can be of any dimension, but it will be clipped to what the server thinks is an appropriate size, 32x32 or 16x16. Using large images would waste space, because the image would still be in memory.

The file is loaded at the time **load-cursor** is called, so the cursor object does not require the file at the time the cursor is displayed. The cursor object survives saving and delivering the image.

If *filename-or-list* is a list then it names a file or a built-in cursor to be loaded for a particular library, optionally together with arguments to be passed to the library. It should be of the form:

```
((libname_1  filename_1  arg_1a  arg_1b  ...)
 (libname_2  filename_2  arg_2a  arg_2b  ...)
 ...
 )
```

where *libname_n* is a keyword naming a supported library such as **:cocoa**, **:win32** or **:gtk** (see **default-library** for the values) and *filename_n* is either a string naming the cursor file to load for this library or a keyword naming one of the built-in cursors. *arg_na*, *arg_nb* and so on are library-specific arguments. Currently these are not used on Microsoft Windows. Hotspot keyword arguments **:x-hot** and **:y-hot** are supported on Cocoa and GTK+ as in the example below. They specify the hotspot of the cursor. The values must be integers inside the image dimensions, that is they satisfy:

```
(and (> image-width  x-hot  -1)
     (> image-height  y-hot  -1))
```

On GTK+ the library-specific arguments also include the keywords **:transparent-color-index** and **:type**, which are passed to **read-external-image**. Note that supplying the *transparent-color-index* allows making a useful cursor with a simple format image file which does not have transparency.

Examples

This example loads a standard Microsoft Windows cursor file:

```
(setq cur1 (capi:load-cursor "arrow_l"))
```

This example loads a standard Windows cursor file, and on Motif uses one of the built-in cursors:

```
(setq cur2
      (capi:load-cursor '((:win32 "3dwns")
                          (:motif :v-double-arrow))))
```

This example loads a horizontal double-arrow on Windows, and a vertical double-arrow on Motif:

```
(setq cur3
      (capi:load-cursor '((:win32 :h-double-arrow)
                          (:motif :v-double-arrow))))
```

This example loads a custom .cur file:

```
(setq cur4
```

```
(capi:load-cursor "C:/Temp/Animated_Cursors/1a.cur"))
```

In this extended example, firstly we load a custom cursor for two platforms:

```
(setq cur
      (capi:load-cursor
       '((:win32
          "c:/WINNT40/Cursors/O_CROSS.CUR")
         (:cocoa
          "/Applications/iPhoto.app/Contents/Resources/retouch-cursor.tif"
          :x-hot 2
          :y-hot 2))))
```

Now we display a pane with the custom cursor loaded above:

```
(setq oo
      (capi:contain
       (make-instance
        'capi:output-pane
        :cursor cur
        :input-model
        `(((:button-1 :press)
           ,(lambda (&rest x)
              (print x)))))))
```

We can remove the custom cursor:

```
(capi:apply-in-pane-process
 oo
 (lambda ()
   (setf (capi:simple-pane-cursor oo)
         :default)))
```

And we can restore the custom cursor:

```
(capi:apply-in-pane-process
 oo
 (lambda ()
   (setf (capi:simple-pane-cursor oo)
         cur)))
```

See also

simple-pane

load-sound *Function*

Summary

Converts data to a loaded sound object on Microsoft Windows and Cocoa.

Package

capi

Signature

`load-sound` *source* **`&key`** *owner* **`=>`** *sound*

Arguments

source⇓ A pathname designator or an array returned by **`read-sound-file`**.

owner⇓ A CAPI **`interface`**, or **`nil`**.

Values

sound⇓ An array of element type **`(unsigned-byte 8)`**.

Description

The function **`load-sound`** converts *source* into a loaded sound which can be played by **`play-sound`**.

source can be a pathname designator or an array returned by **`read-sound-file`**.

owner should be a CAPI **`interface`** object, or **`nil`** which means that the sound's owner is the current top level interface.

The loaded sound *sound* will be unloaded (freed) automatically when its owner is destroyed. To create a sound that is never unloaded, pass the **`screen`** as the argument *owner*.

Notes

1. The array *sound* contains the contents of the file. Its bytes are interpreted by the OS functions, so the format can be whatever they can deal with, for example WAV on Microsoft Windows. The fact that this date is represented as an **`(unsigned-byte 8)`** array in Lisp does not constrain the output size.
2. **`load-sound`** is not implemented on GTK+ and Motif.

See also

`free-sound`
`play-sound`
`read-sound-file`
18.2 Sounds

locate-interface *Generic Function*

Summary

Finds an interface of a given class that matches supplied initargs.

Package

`capi`

Signature

`locate-interface` *class-spec* **`&rest`** *initargs* **`&key`** *screen no-busy-interface* **`&allow-other-keys`** **`=>`** *interface*

Arguments

class-spec⇓	A specifier for a subclass of **interface**.
initargs⇓	Initialization arguments for *class-spec*.
screen⇓	A **screen** or **nil**.
no-busy-interface⇓	A boolean, defaulting to **nil**.

Values

interface	An interface of class *class-spec*, or **nil**.

Description

The generic function **locate-interface** finds an interface of the class specified by *class-spec* that matches *initargs* and *screen*.

First, **locate-interface** finds all interfaces of the class specified by *class-spec* by calling **collect-interfaces** with *class-spec* and *screen*. The first of these which match *initargs* (by **interface-match-p**) is returned.

If there is no match, then **locate-interface** finds the first of these which can be reused for *initargs*, by **interface-reuse-p**. This reusable interface is reinitialized by **reinitialize-interface** and returned.

no-busy-interface controls the use of the busy cursor during reinitializing of a reusable interface. If *no-busy-interface* is **nil**, then this interface has the busy cursor during reinitialization. If *no-busy-interface* is true, then there is no busy cursor.

If no matching or reusable interface is found, or if global interface re-use is disabled by **(setf reuse-interfaces-p)**, then **locate-interface** returns **nil**.

See also

collect-interfaces
interface-match-p
interface-reuse-p
reuse-interfaces-p

lower-interface *Function*

Summary

Pushes a window to the back of the screen.

Package

capi

Signature

lower-interface *pane*

Arguments

pane⇓	A **simple-pane**.

Description

The function **lower-interface** pushes the window containing *pane* to the back of the screen.

To raise the window use **raise-interface**, and to iconify it use **hide-interface**.

See also

hide-interface
interface
raise-interface
quit-interface
7.7 Manipulating top-level windows

make-container *Function*

Summary

Creates a container for a specified element.

Package

capi

Signature

make-container *element* **&rest** *interface-args*

Arguments

element⇓	A Lisp object.
interface-args⇓	Initialization arguments of **interface**.

Description

The function **make-container** creates a container for *element* such that calling **display** on it will produce a window containing *element* on the screen. It will produce a container for any of the following classes of object:

- simple-pane
- layout
- interface
- pinboard-object
- menu
- menu-item
- menu-component
- list

In the case of a **list**, the CAPI tries to see what sort of objects they are and makes an appropriate container. For instance, if they were all simple panes it would put them into a column layout.

The arguments *interface-args* will be passed through to the **make-instance** of the top-level interface, assuming that pane is not a top-level interface itself.

The complementary function **contain** uses **make-container** to create a container for an element which it then displays.

Examples

```
(capi:display (capi:make-container
                (make-instance
                 'capi:text-input-pane)))
```

See also

contain
display
interface
element
10.5 Creating your own dialogs

make-docking-layout-controller *Function*

Summary

Makes a docking layout controller object.

Package

capi

Signature

make-docking-layout-controller => *controller*

Values

controller	A docking layout controller.

Description

The function **make-docking-layout-controller** returns a docking layout controller object for use as the **:controller** initarg in **docking-layout**.

Layouts which share a docking layout controller are known as a Docking Group. See **docking-layout** for information about Docking Groups.

See also

docking-layout

make-foreign-owned-interface *Function*

Summary

Creates a dummy interface which allows another application's window to be the owner of a CAPI dialog.

Package

`capi`

Signature

`make-foreign-owned-interface &key` *handle name* **`=>`** *interface*

Arguments

handle⇓	A Microsoft Windows hwnd.
name⇓	A string naming *interface*.

Values

interface⇓	An instance of **`foreign-owned-interface`**.

Description

The function **`make-foreign-owned-interface`** creates an instance of **`foreign-owned-interface`**. *interface* can be used as the *owner* argument when displaying a dialog. For information about dialog owners, see **10 Dialogs: Prompting for Input**.

handle must be supplied and is the window handle (Windows hwnd) of a window in some application. For a CAPI window this window handle can be obtained by **`simple-pane-handle`**. For non-CAPI applications, the method of finding the window handle will depend on the language and the way windows are represented, so you should consult the appropriate documentation.

name becomes the name of *interface*, and has no other meaning.

`make-foreign-owned-interface` is implemented only on Microsoft Windows.

Examples

This example shows how a CAPI window can be the owner of a dialog in another LispWorks image.

Start LispWorks for Windows.

1. In the Listener, do **Tools > Interface > Listen**. This puts the Listener interface in the value of **`*`**.
2. In the Listener enter **`(capi:simple-pane-handle *)`**. The returned value is the window handle, it should be an integer. Denote this value by *hwnd*.

Start another LispWorks for Windows image (do not quit the first image). In the Listener of this second LispWorks image:

1. Enter **`(setq foi (capi:make-foreign-owned-interface :handle`** *hwnd***`)).`**
2. Enter **`(capi:prompt-for-color "Color?" :owner foi)`**.

Now note that the Color dialog is owned by the Listener of the first LispWorks image.

make-general-image-set *Function*

Summary

Creates an **image-set** object.

Package

capi

Signature

make-general-image-set &key *image-count width height id* **=>** *image-set*

Arguments

image-count⇓	An integer.
width⇓	An integer or **nil**.
height⇓	An integer or **nil**.
id⇓	A pathname, string or symbol.

Values

image-set	An **image-set** object.

Description

The function **make-general-image-set** creates an **image-set** object that refers to an image or a file containing an image.

id is a pathname or string identifying an image file, or a symbol previously registered with **register-image-translation**.

width and *height* are the dimensions of a single sub-image within the main image, and *image-count* specifies the number of sub-images in the image.

Examples

```
(example-edit-file "capi/choice/tree-view")
```

```
(example-edit-file "capi/choice/extended-selection-tree-view")
```

```
(example-edit-file "capi/elements/toolbar")
```

See also

image-set
make-resource-image-set

5.10.4 image-list, image-set and image-locator

make-icon-resource-image-set *Function*

Summary

Constructs an image set object identifying a icon resource in a Windows DLL.

Package

`capi`

Signature

`make-icon-resource-image-set &key` *image-count width height library id* **`=>`** *image-set*

Arguments

image-count⇓	An integer.
width⇓	An integer.
height⇓	An integer.
library⇓	A string.
id⇓	A string or an integer.

Values

image-set	An **`image-set`** object.

Description

The function **`make-icon-resource-image-set`** constructs an image set object that identifies an image stored as a icon resource in a DLL on Microsoft Windows.

width and *height* are the dimensions of a single sub-image within the main image, and *image-count* specifies the number of sub-images in the image.

library should be a string specifying the name of the DLL.

id should be either an integer which is the resource identifier of the icon, or a string naming the icon resource.

Notes

`make-icon-resource-image-set` is only available in LispWorks for Windows.

See also

`image-set`
`make-general-image-set`
5.10.4 image-list, image-set and image-locator

make-image-locator *Function*

Summary

Creates an **image-locator** object to use with toolbars, list views and tree views.

Package

capi

Signature

make-image-locator &key *image-set index* **=>** *image-locator*

Arguments

image-set⇓	An **image-set**.
index⇓	A non-negative integer.

Values

image-locator	An **image-locator**.

Description

The function **make-image-locator** creates an **image-locator** object for use with **toolbar**, **list-view** and **tree-view**. It is used to specify a single sub-image with index *index* from a larger image in *image-set* that contains many images side by side. It is also useful for accessing some images that can only be specified by means of an **image-set**.

See also

image-set
5.10.4 image-list, image-set and image-locator

make-menu-for-pane *Function*

Summary

Makes a **menu** or a **menu-component** for a pane.

Package

capi

Signature

make-menu-for-pane *pane items* **&key** *title menu-name component-p* **=>** *menu*

Arguments

pane⇓ A pane.

items⇓ A list of **menu-object**s.

title⇓ A string or **nil**.

menu-name⇓ A string or **nil**.

component-p⇓ A boolean.

Values

menu⇓ A **menu** or a **menu-component**.

Description

The function **make-menu-for-pane** makes a **menu** or a **menu-component** for the pane *pane* with the items specified by *items*.

items should be a list in which each element is a **menu-item**, **menu-component** or **menu**.

title and *menu-name* provide a title and name for *menu*. *title* and *menu-name* both default to **nil**.

If *component-p* is true, then **make-menu-for-pane** creates a **menu-component** rather than a **menu**. The default value of *component-p* is **nil**.

menu is set up so that by default each callback inside it is done on the pane *pane* itself. This is the useful feature of **make-menu-for-pane** because it avoids the need to set up items to do their callbacks on *pane* explicitly.

Note that this is merely the default behavior. You can specify different callback behavior on a per-item basis, using *setup-callback-argument* and *callback-data-function* (see **menu-object**), *callback-type* (see **callbacks**) and *data* for **menu-item** (see **item**).

See also

make-pane-popup-menu
pane-popup-menu-items
8.12 Popup menus for panes

make-pane-popup-menu *Generic Function*

Summary

Generates a popup **menu** or **menu-component**.

Package

capi

Signature

make-pane-popup-menu *pane interface* **&key** *title menu-name component-p* **=>** *menu*

Arguments

pane⇓ A pane in an interface.

interface⇓ An interface or **nil**.

title⇓ A string or **nil**.

menu-name⇓ A string or **nil**.

component-p⇓ A boolean.

Values

menu⇓ A **menu** or a **menu-component**.

Description

The generic function **make-pane-popup-menu** generates a popup menu for *pane*.

interface can be **nil** if *pane* has already been created, in which case the *interface* of *pane* is used (obtained by the **element** accessor **element-interface**).

title and *menu-name* provide a title and name for *menu*. *title* and *menu-name* both default to **nil**.

If *component-p* is true, then **make-pane-popup-menu** creates a **menu-component** rather than a **menu**. The default value of *component-p* is **nil**.

Examples

This code makes an interface with two **graph-pane**s. The **initialize-instance** method uses **make-pane-popup-menu** to add a menu to the menu bar from which the user can perform operations on the graphs.

Note that, because **make-pane-popup-menu** calls **make-menu-for-pane** to make each menu, the callbacks in the menus are automatically done on the appropriate graph.

```
(capi:define-interface gg ()
  ()
  (:panes
   (g1 capi:graph-pane)
   (g2 capi:graph-pane))
  (:layouts
   (main-layout capi:column-layout '(g1 g2)))
  (:menu-bar)
  (:default-initargs
   :visible-min-width 200
   :visible-min-height 300))

(defmethod initialize-instance :after ((self gg)
                                       &key)
  (with-slots (g1 g2) self
    (setf
     (capi:interface-menu-bar-items self)
     (append
      (capi:interface-menu-bar-items self)
      (list
       (make-instance
        'capi:menu
        :title "Graphs"
        :items
        (list
         (capi:make-pane-popup-menu
```

```
              g1 self :title "graph1")

             (capi:make-pane-popup-menu
              g2 self :title "graph2"))))))))

    (capi:display (make-instance 'gg))
```

See also

make-menu-for-pane
8.12 Popup menus for panes

make-resource-image-set *Function*

Summary

Constructs an image set object identifying a bitmap resource in a Windows DLL.

Package

capi

Signature

make-resource-image-set &key *image-count width height library id* **=>** *image-set*

Arguments

image-count⇓	An integer.
width⇓	An integer.
height⇓	An integer.
library⇓	A string.
id⇓	A string or an integer.

Values

image-set	An **image-set** object.

Description

The function **make-resource-image-set** constructs an image set object that identifies an image stored as a bitmap resource in a DLL on Microsoft Windows.

width and *height* are the dimensions of a single sub-image within the main image, and *image-count* specifies the number of sub-images in the image.

library should be a string specifying the name of the DLL.

id should be either an integer which is the resource identifier of the bitmap, or a string naming the bitmap resource.

Notes

make-resource-image-set is only available in LispWorks for Windows.

See also

image-set
make-icon-resource-image-set
make-general-image-set
5.10.4 image-list, image-set and image-locator

make-scaled-general-image-set *Function*

Summary

Constructs an image set object which scales images in another image set on Microsoft Windows.

Package

capi

Signature

make-scaled-general-image-set &key *width height id image-count* **=>** *image-set*

Arguments

width⇓	An integer.
height⇓	An integer.
id⇓	A pathname, string or symbol.
image-count⇓	An integer.

Values

image-set	An **image-set** object.

Description

The function **make-scaled-general-image-set** constructs an image set that provides scaled images based on an **image-set** object constructed from *id* as if by **make-general-image-set**.

width and *height* are the dimensions of a single sub-image within the main image, and *image-count* specifies the number of sub-images in both images. That is, the sub-images are scaled to this size.

The default value of *image-count* is 1.

Notes

make-scaled-general-image-set is only available in LispWorks for Windows.

See also

image-set
make-general-image-set
5.10.4 image-list, image-set and image-locator

make-scaled-image-set *Function*

Summary

Creates an image set by scaling the images of another image set on Microsoft Windows.

Package

capi

Signature

make-scaled-image-set &key *image-count width height base-image-set* **=>** *image-set*

Arguments

image-count⇓	An integer.
width⇓	An integer.
height⇓	An integer.
base-image-set⇓	An **image-set** object.

Values

image-set⇓	An **image-set** object.

Description

The function **make-scaled-image-set** constructs an image set that provides scaled images based on an existing image set object *base-image-set*.

width and *height* are the dimensions of a single sub-image within the main image. That is, the sub-images in *base-image-set* are scaled to this size to produce the sub-images of *image-set*.

image-count specifies the number of sub-images in the image. It is unspecified what happens if *image-count* is different from the image count in *base-image-set*.

Notes

make-scaled-image-set is only available in LispWorks for Windows.

See also

image-set
make-general-image-set
5.10.4 image-list, image-set and image-locator

make-sorting-description *Function*

Summary

Makes a sorting description suitable for use in a **sorted-object**.

Package

capi

Signature

make-sorting-description &key *type key sort reverse-sort sort-function object-sort-caller* **=>** *sorting-description*

Arguments

type⇓	A Lisp object naming the type of sorting.
key⇓	A function of 1 argument.
sort⇓	A function of 2 arguments.
reverse-sort⇓	A function of 2 arguments.
sort-function⇓	A sorting function.
object-sort-caller⇓	A function of 5 arguments.

Values

sorting-description	A sorting description object.

Description

The function **make-sorting-description** makes a sorting description object that can be used as one of the *sort-descriptions* in a **sorted-object** such as a **list-panel**.

type is a name that should be unique (compared by **cl:equalp**) amongst the *sort-descriptions* of a **sorted-object**.

key is a function that is passed to *sort-function* as its **:key** argument. The default value of *key* is **cl:identity**.

sort is a predicate function that is passed to *sort-function* to compare pairs of items.

reverse-sort is a predicate function that is passed to *sort-function* for reverse sorting.

Unless *object-sort-caller* is supplied, *sort-function* is the function that is called to actually do the sorting. Its signature is:

> **sort-function** *items predicate* **&key** *key*

The default value of *sort-function* is **cl:sort**.

When *object-sort-caller* is supplied, then it is called instead of calling *sort-function*, and is responsible for the sorting. The signature of the caller is:

> **object-sort-caller** *sorted-object items sort-function sort-predicate key* **=>** *sorted-items*

where *sorted-object* is the **sorted-object** itself, *items* is the list of items to sort, and *sort-function*, *sort-predicate* and *key* are taken from the description. *sort-predicate* is either *sort* or *reverse-sort* as appropriate. The caller needs to return a sorted list of the items.

The caller can do the default behavior by:

funcall *sort-function item sort* **:key** *key*

Notes

1. The purpose of using *object-sort-caller* is to allow access to the **sorted-object** to decide how to do the sorting. When using *object-sort-caller*, *sort-function*, *sort*, *reverse-sort* and *key* are used solely as arguments to it, hence in this case you can supply arbitrary values which the caller interprets.

2. The sorting can be destructive.

Examples

```
(setq lp
      (capi:contain
       (make-instance
        'capi:list-panel
        :items '("Apple"
                 "Orange"
                 "Mangosteen"
                 "Pineapple")
        :visible-min-height '(:character 5)
        :sort-descriptions
        (list (capi:make-sorting-description
               :type :length
               :sort
               #'(lambda (x y)
                   (> (length x) (length y)))
               :reverse-sort
               #'(lambda (x y)
                   (< (length x) (length y))))
              (capi:make-sorting-description
               :type :alphabetic
               :sort 'string-greaterp
               :reverse-sort 'string-lessp)))))

(capi:sorted-object-sort-by lp :length)

(capi:sorted-object-sort-by lp :alphabetic)
```

See also

sort-object-items-by
sorted-object
sorted-object-sort-by

manipulate-pinboard *Generic Function*

Summary

Adds or removes one or more **pinboard-object**s on a pinboard.

Package

capi

Signature

manipulate-pinboard *pinboard-layout pinboard-object action* **&key** *position*

Arguments

pinboard-layout⇓	A **pinboard-layout**.
pinboard-object⇓	A **pinboard-object** to be added, or (with *action* **:add-many**) a list of **pinboard-object**s to be added, or (with *action* **:delete-if**) a function of one argument, for multiple deletion.
action⇓	One of **:add**, **:add-top**, **:add-bottom**, **:add-many** or **:delete**. Can also be **:delete-if**, for multiple deletion.
position⇓	One of **:top** or **:bottom**, or a non-negative integer.

Description

The generic function **manipulate-pinboard** adds *pinboard-object* to *pinboard-layout*, or removes one or more **pinboard-object**s from *pinboard-layout*. These operations can also be effected using **(setf layout-description)**, but **manipulate-pinboard** is much more efficient and produces a better display.

If *action* is **:add**, then the **pinboard-object** *pinboard-object* is added according to the value of *position*:

:top	On top of the other pinboard objects.
:bottom	Below the other pinboard objects.
An integer	At index *position* in the sequence of pinboard objects, where 0 is the index of the topmost pinboard object. Values of *position* greater than the number of pinboard objects are interpreted as **:bottom**.

action **:add-top** is the same as passing *action* **:add** and *position* **:top**.

action **:add-bottom** is the same as passing *action* **:add** and *position* **:bottom**.

action **:add-many** is like calling the function with *action* **:add** several times, but is more efficient. The value of *pinboard-object* must be a list of **pinboard-object**s, each of which is added at the specified *position*, as for **:add**.

action **:delete** deletes the **pinboard-object** *pinboard-object* from *pinboard-layout*.

When *action* is **:delete-if**, *pinboard-object* should be a function which takes one argument, a **pinboard-object**. This function is applied to each **pinboard-object** in *pinboard-layout* and each object for which it returns true is deleted from *pinboard-layout*.

Notes

You can control automatic resizing of *pinboard-object* using **set-object-automatic-resize**.

Examples

```
(setq pl
      (capi:contain
       (make-instance 'capi:pinboard-layout
                      :visible-min-height 500
                      :visible-min-width 200)))
```

Add some **pinboard-object**s:

```
(capi:apply-in-pane-process
 pl #'(lambda (pp)
        (dotimes (y 10)
          (let ((yy (* y 40)))
            (capi:manipulate-pinboard
             pp
             (make-instance 'capi:line-pinboard-object
                            :start-x 4  :start-y yy
                            :end-x 54   :end-y (+ 6 yy))
             :add-top)
            (capi:manipulate-pinboard
             pp
             (make-instance 'capi:pinboard-object
                            :x 4  :y (+ 20 yy)
                            :width 50 :height 6
                            :graphics-args
                            '(:background :red))
             :add-top))))
 pl)
```

Remove some **pinboard-object**s:

```
(capi:apply-in-pane-process
 pl
 #'(lambda (pp)
     (dotimes (y 15)
       (let ((po (capi:pinboard-object-at-position pp
                                                    10
                                                    (* y 30))))
         (when po (capi:manipulate-pinboard pp
                                             po
                                             :delete)))))
 pl)
```

Remove all **line-pinboard-object**s:

```
(capi:apply-in-pane-process
 pl 'capi:manipulate-pinboard pl
 #'(lambda (x)
     (typep x 'capi:line-pinboard-object))
 :delete-if)
```

See also

pinboard-layout
set-object-automatic-resize

map-collection-items *Generic Function*

Summary

The generic function **`map-collection-items`** calls a specified function on all the items in a collection.

Package

`capi`

Signature

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

Arguments

collection⇓	A **`collection`**.
function⇓	A function designator for a function of one argument.
collect-results-p⇓	A generalized boolean.

Description

Calls *function* on each item in *collection* by calling *collection*'s *items-map-function*. If *collect-results-p* is true, the results of these calls are returned in a list.

Examples

```
(setq collection (make-instance 'capi:collection
                                :items '(1 2 3 4 5)))

(capi:map-collection-items collection
                           'princ-to-string t)
```

See also

`collection`
`choice`

map-pane-children *Generic Function*

Summary

Calls a function on each of a pane's children.

Package

`capi`

Signature

`map-pane-children` *pane function* **`&key`** *visible test reverse*

Arguments

pane⇓ A CAPI pane.

function⇓ A function of one argument.

visible⇓ A boolean. The default value is **`nil`**.

test⇓ A function of one argument, or **`nil`**. The default is **`nil`**.

reverse⇓ A boolean. The default value is **`nil`**.

Description

The generic function **`map-pane-children`** applies *function* to *pane*'s immediate children.

If *visible* is true, then *function* is applied only to the visible children.

If *test* is non-nil, it is a function which is applied first to each child, and only those for which *test* returns a true value are then passed to *function*.

If *reverse* is non-nil, the order in which the children are processed is reversed.

Examples

This example constructs a pinboard containing random ellipses. A repainting function is mapped over them, restricted to those with width greater than height.

```
(defun random-color ()
  (aref #(:red :blue :green :yellow :cyan
          :magenta :pink :purple :black :white)
        (random 10)))

(defun random-origin ()
  (list (random 350) (random 250)))

(defun random-size ()
  (list (+ 10 (random 40))
        (+ 10 (random 40))))

(setf ellipses
      (capi:contain
       (make-instance
        'capi:pinboard-layout
        :children
        (loop for i below 40
              for origin = (random-origin)
              for size = (random-size)
              collect
              (make-instance 'capi:ellipse
                             :x (first origin)
                             :y (second origin)
                             :width (first size)
                             :height (second size)
                             :graphics-args
                             (list :foreground
                                   (random-color))
                             :filled t)))))
```

```
(defun repaint (ellipse)
  (setf (capi:pinboard-object-graphics-args ellipse)
       (list :foreground (random-color)))
  (capi:redraw-pinboard-object ellipse t))

(defun widep (ellipse)
  (capi:with-geometry ellipse
      (> capi:%width% capi:%height%)))

(capi:map-pane-children ellipses 'repaint :test 'widep)
```

See also

`map-pane-descendant-children`
3.7 Hierarchy of panes

map-pane-descendant-children *Generic Function*

Summary

Calls a function on each of the descendant panes of a pane.

Package

`capi`

Signature

`map-pane-descendant-children` *pane function* **`&key`** *visible test reverse leaf-only*

Arguments

pane⇓	A CAPI pane.
function⇓	A function of one argument.
visible⇓	A boolean. The default value is **`nil`**.
test⇓	A function of one argument, or **`nil`**. The default is **`nil`**.
reverse⇓	A boolean. The default value is **`nil`**.
leaf-only⇓	A generalized boolean. The default value is **`nil`**.

Description

The generic function **`map-pane-descendant-children`** applies *function* to *pane*'s descendant panes (that is, the children and each of their children recursively), depth first.

If *visible* is true, then *function* is applied only to the visible descendant panes.

If *test* is non-nil, it is a function which is applied first to each descendant pane, and only those for which *test* returns a true value are then passed to *function*.

If *reverse* is non-nil, the order in which the children are processed is reversed.

If *leaf-only* is true, then *function* is applied only to those panes which do not have children.

See also

map-pane-children
pane-descendant-child-with-focus
3.7 Hierarchy of panes

map-typeout *Function*

Summary

Makes a **collector-pane** visible.

Package

capi

Signature

map-typeout *pane* **&rest** *args*

Arguments

pane⇓	A pane.
args⇓	Initialization arguments for **collector-pane**.

Description

The function **map-typeout** makes a **collector-pane** the visible child of a **switchable-layout**, and returns it as well. The switchable layout is found by looking up the parent hierarchy starting from *pane*.

The **switchable-layout** should have one or more children. If it has one child, a new **collector-pane** is made using *args* as the initargs with **:buffer-name** defaulting to **"Background Output"**. If it has more than one child, it searches through the children to find the first **collector-pane**.

See also

unmap-typeout
with-random-typeout
collector-pane

maximum-moving-objects-to-track-edges *Variable*

Summary

Limits the tracking of edges in a graph.

Package

capi

Initial Value

15

Description

The variable ***maximum-moving-objects-to-track-edges*** limits the tracking of edges in a graph.

If there are more than ***maximum-moving-objects-to-track-edges*** objects being moved in a graph, then edges are not tracked.

The value should be an integer.

See also

graph-pane

menu *Class*

Summary

The class **menu** creates a menu for an interface when specified as part of the menu bar (or as a submenu of a menu on the menu bar). It can also be displayed as a context menu.

Package

capi

Superclasses

element
titled-menu-object

Initargs

:items	The items to appear in the menu.
:items-function	A function to dynamically compute the items.
:mnemonic	A character, integer or symbol specifying a mnemonic for the menu.
:mnemonic-escape	A character specifying the mnemonic escape. The default value is **#\&**.
:mnemonic-title	A string specifying the title and a mnemonic.
:image-function	A function providing images for the menu items, or **nil**.

Accessors

menu-items
menu-image-function

Description

A menu has a title, and has items appearing in it, where an item can be either a **menu-item**, a **menu-component** or another **menu**.

The simplest way of providing items to a menu is to pass them as the argument *items*, but if you need to compute the items dynamically you should provide the setup callback *items-function*. This function should return a list of menu items for the new menu. By default *items-function* is called on the menu's interface, but a different argument can be specified using the **menu-object** initarg *setup-callback-argument*.

If an item is not of type **menu-object**, then it gets converted to a **menu-object** with the item as its data. This function is called before the *popup-callback* and the *enabled-function* which means that they can affect the new items.

To specify a mnemonic in the menu title, you can use the initarg **:mnemonic**. The value *mnemonic* can be:

An integer — The index of the mnemonic in the title.

A character — The mnemonic in the title.

nil — A character is chosen from a list of common mnemonics, or the **:default** behavior is followed. This is the default.

:default — A mnemonic is chosen using some rules.

:none — The title has no mnemonic.

An alternative way to specify a mnemonic is to pass *mnemonic-title* (rather than *title*) This is a string which provides the text for the menu title and also specifies the mnemonic character. The mnemonic character is preceded in *mnemonic-title* by *mnemonic-escape,* and *mnemonic-escape* is removed from *mnemonic-title* before the text is displayed. For example:

```
:mnemonic-title "&Open File..."
```

At most one character can be specified as the mnemonic in *mnemonic-title*. To make *mnemonic-escape* itself appear in the button, precede it in *mnemonic-title* with *mnemonic-escape*. For example:

```
:mnemonic-title "&Compile && Load File..."
```

If *image-function* is non-nil, it should be a function of one argument. *image-function* is called with the data of each menu item and should return one of:

nil — No image is shown.

An **image** — The menu displays this image.

An image id or **external-image**

The system converts the value to a temporary **image** for the menu item and frees it when it is no longer needed.

If *image-function* is **nil**, no items in the menu have images. This is the default value.

Notes

1. *items-function* is called before the menu is raised (in order to initialize accelerators) and in particular it may be called before the interface is created. Therefore *items-function,* if you supply it, should work at this early stage.

2. On Microsoft Windows, Cocoa and GTK+, menu items can contain both images and strings, so the *print-function* should return the appropriate string or **""** if no string is required. On Motif, if there is an image then the string is ignored. You can test programmatically whether menus with images are supported with **pane-supports-menus-with-images**.

3. On Microsoft Windows and GTK+, menu items that can have check marks (those inside **menu-component** with *interaction* **:multiple-selection** or **:single-selection**) cannot have images (the image is ignored for such items).

4. When debugging a menu, it may be useful to pop up a window containing a menu with the minimum of fuss. The function **`contain`** will do just that for you.
5. To display a menu as a context (right button) menu, use **`display-popup-menu`**, and to display a menu via a labelled button use **`popup-menu-button`**.
6. You must not use a menu object in multiple different places in menu bar(s) at the same time. Supply distinct instances instead. The one exception is popup menus, which can be used repeatedly and in different places.
7. Microsoft Windows can hide mnemonics when the user is not using the keyboard. See **3.1.4.2 Mnemonics on Microsoft Windows**.

Examples

```
(capi:contain (make-instance 'capi:menu
                             :title "Test"
                             :items '(:red :green :blue)))
```

```
(capi:contain (make-instance
               'capi:menu
               :title "Test"
               :items '(:red :green :blue)
               :print-function 'string-capitalize))
```

```
(capi:contain (make-instance
               'capi:menu
               :title "Test"
               :items '(:red :green :blue)
               :print-function 'string-capitalize
               :callback #'(lambda (data interface)
                             (capi:display-message
                              "Pressed ~S" data))))
```

Here is an example showing how to add submenus to a menu:

```
(setq submenu (make-instance 'capi:menu
                             :title "Submenu..."
                             :items '(1 2 3)))
```

```
(capi:contain (make-instance
               'capi:menu
               :title "Test"
               :items (list submenu)))
```

Here is an example showing how to use the *items-function*:

```
(capi:contain (make-instance
               'capi:menu
               :title "Test"
               :items-function #'(lambda (interface)
                                   (loop for i below 8
                                    collect (random 10)
                                   ))))
```

Finally, some examples showing how to specify a mnemonic in a menu title:

```
(capi:contain (make-instance
               'capi:menu
```

```
                  :title "Mnemonic Title"
                  :mnemonic 1
                  :items '(1 2 3)))


(capi:contain (make-instance
                  'capi:menu
                  :mnemonic-title "M&nemonic Title"
                  :items '(1 2 3)))


(capi:contain (make-instance
                  'capi:menu
                  :mnemonic-title "M&e && You"
                  :items '("Me" "You")))
```

This example shows how to make a menu with images:

```
(example-edit-file "capi/elements/menu-with-images")
```

There are further examples here:

```
(example-edit-file "capi/applications/")
```

See also

display-popup-menu
menu-component
menu-item
menu-object
ole-control-add-verbs
pane-supports-menus-with-images
popup-menu-button
1.2.1 CAPI elements
8 Creating Menus
13.10 Working with images

menu-component *Class*

Summary

The class **menu-component** is a choice that is used to group menu items and submenus both visually and functionally. The items contained by the **menu-component** appear separated from other items, menus, or menu components, by separators.

Package

capi

Superclasses

choice
titled-menu-object

Initargs

:items The items to appear in the menu.

:items-function A setup callback function to dynamically compute the items.

:selection-function
A setup callback function to dynamically compute the selection.

:selected-item-function
A setup callback function to dynamically compute the selected item.

:selected-items-function
A setup callback function to dynamically compute the selected items.

Description

Because **menu-component** is a choice, the component can have *interaction* **:no-selection**, **:single-selection** or **:multiple-selection** (extended selection does not apply here). This is represented visually in the menu as appropriate to the window system that the CAPI is running on (by ticks in Microsoft Windows, and by radio buttons and check buttons in Motif).

Note that it is not appropriate to have menu components or submenus inside **:single-selection** and **:multiple-selection** components, but it is OK in **:no-selection** components.

items and *items-function* behave as in **menu**.

No more than one of *selection-function*, *selected-item-function* and *selected-items-function* should be non-nil. Each defaults to **nil**. If one of these setup callbacks is supplied, it should be a function which is called before the **menu-component** is displayed and which determines which items are selected. By default the setup callback is called on the interface of the **menu-component**, but this argument can be changed by passing the **menu-object** initarg *setup-callback-argument*.

selection-function, if non-nil, should return a value which is suitable for passing to the **choice** accessor **(setf choice-selection)**. This will be **nil**, or a single index (for *interaction* **:single-selection**), or a list of item indices (for *interaction* **:multiple-selection** and **:extended-selection**).

selected-item-function, if non-nil, should return an object which is an item in the **menu-component**, or is equal to such an item when compared by the **menu-component**'s *test-function*.

selected-items-function, if non-nil, should return a list of such objects.

Examples

```
(capi:contain (make-instance
              'capi:menu-component
              :items '(:red :green :blue)
              :print-function 'string-capitalize
              :interaction :single-selection))


(capi:contain (make-instance
               'capi:menu-component
               :items '(:red :green :blue)
               :print-function 'string-capitalize
               :interaction :multiple-selection))


(capi:contain (make-instance
               'capi:menu
               :items (list
                       "An Item"
                       (make-instance
                        'capi:menu-component
                        :items '(:red :green :blue)
                        :print-function
```

```
 'string-capitalize
:interaction :no-selection)
"Another Item")))
```

See also

menu
menu-item
1.2.1 CAPI elements
5 Choices - panes with items
8 Creating Menus

menu-item *Class*

Summary

An individual item in a menu or menu component.

Package

capi

Superclasses

item
titled-menu-object

Initargs

:accelerator	A character, string or plist, or the keyword **:default**.
:alternative	A generalized boolean.
:help-key	An object used for lookup of help. Default value **t**.
:mnemonic	A character, integer or symbol specifying a mnemonic for the menu item.
:mnemonic-escape	A character specifying the mnemonic escape. The default value is **#\&**.
:mnemonic-title	A string specifying the text and a mnemonic.
:selected-function	A setup callback determining whether the item is selected.
:enabled-function-for-dialog	**nil**, **t**, **:same-as-normal** or a function designator. Determines enabled state when a dialog is on screen.

Readers

help-key

Description

The class **menu-item** is an individual item in a menu or menu component. Instances of **menu-item** are often made automatically by **define-interface**, but you can make them explicitly as well.

The text displayed in the menu item is the contents of the *text* slot, or the contents of the *title* slot, otherwise it is the result of applying the *print-function* to the *data*.

If *selected-function* is non-nil it should a function which is called before the **menu-item** is displayed and which determines whether or not the **menu-item** is selected. By default *selected-function* is called on the interface of the **menu-item**, but this argument can be changed by passing the **menu-object** initarg *setup-callback-argument*. The default value of *selected-function* is **nil**.

Callbacks are made in response to a user gesture on a **menu-item**. The *callback-type* (see **callbacks**), *callback* and *callback-data-function* (see **menu-object**) are found by looking for a non-nil value, first in 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-type* slot non-nil if needed.

To specify a mnemonic in the menu item, you can use the initarg **:mnemonic**, or the initargs **:mnemonic-title** and **:mnemonic-escape**. These initargs are all interpreted just as in **menu**.

A menu item should not be used more in more than one place at a time.

help-key is interpreted as described for **element**.

accelerator can be a character or string specifying a key gesture which will be the accelerator for the menu item.

Note that **both-case-p** characters are not allowed with the single modifier **Shift** in the accelerator argument. So instead of:

```
:accelerator "shift-x"
```

use:

```
:accelerator "X"
```

Note that the **Shift** modifier still appears in the menu.

A **both-case-p** character is allowed with **Shift** if there are other modifiers, for example:

```
:accelerator "alt-shift-x"
```

If *accelerator* is a **character** then the system adds the normal modifier for the platform. That is, **Command** on Cocoa and **Control** on Microsoft Windows. The shortcut is validated for the platform.

If *accelerator* is a **string** with modifier keys then the system uses it only if it follows the normal conventions for the platform. The shortcut is validated for the platform.

The special virtual modifier name "accelerator" is allowed in string values of *accelerator*. It is interpreted as the normal modifier key for the platform. For example:

```
:accelerator "accelerator-x"
```

means **Control+X** on Microsoft Windows and Motif, and **Command+X** on Cocoa.

If *accelerator* is a plist then its keys are keywords naming some or all of the supported libraries (as returned by **default-library**). The plist's values are characters or strings which the system interprets as above, except that no check is made that the keyboard shortcut is valid for the platform.

accelerator has a special default value **:default**, which means that, depending on **interface-keys-style** for the interface, a standard accelerator is added if the item title matches a standard menu command. For the full set of standard accelerators see **8.7.1 Standard default accelerators**.

Note: *accelerator* is not supported when the **menu-item** is in the *pane-menu* of a **simple-pane**.

alternative, when true, makes the **`menu-item`** an "alternative item". Alternative items are invoked if modifiers are held while selecting the "main item". These modifiers are defined by the item's *accelerator*. The main item is the one before the first alternative item, and each alternative item must be within the same menu and menu component. For an example see:

```
(example-edit-file "capi/elements/accelerators")
```

and for more information see **8.8 Alternative menu items**.

enabled-function-for-dialog determines whether the item is enabled when a dialog is on the screen. Items in the menu bar menus and sub-menus are disabled by default while a dialog is on the screen on top of the active window. You can override this by specifying *enabled-function-for-dialog*. The value can be one of:

`t` The item is enabled whenever there is a dialog.

`nil` The item is disabled whenever there is a dialog.

`:same-as-normal` Do the same as when there is no dialog. This depends on the *enabled-function* (see **`menu-object`**).

A function A function that is called instead of the *enabled-function* to decide if the item should be enabled. It is called with one argument, by the default the menu interface, which can be overridden by the initarg **`:setup-callback-argument`** (see **`menu-object`** for details).

The default value of *enabled-function-for-dialog* is **`nil`**.

Notes

Some accelerators do not work on some platforms because they have other standard meanings, for example on Microsoft Windows **`F1`** always invokes the *help-callback*.

On X11/Motif the accelerators of alternative items do not work.

Examples

```
(capi:contain (make-instance 'capi:menu-item
                             :text "Press Me"))


(capi:contain (make-instance 'capi:menu-item
                             :data :red
                             :print-function
                              'string-capitalize))


(capi:contain (make-instance
               'capi:menu-item
               :data :red
               :print-function 'string-capitalize
               :callback #'(lambda (data interface)
                             (capi:display-message
                              "Pressed ~S"
                              data))))
```

In this example note how the **File** menu gets accelerators automatically for its standard items:

```
(defun do-menu-item (item)
  (capi:display-message
   (format nil "~A" (capi:item-data item))))

(capi:define-interface mmm () ()
```

```
      (:menu-bar f-menu a-menu)
      (:menus
       (f-menu
         "File"
        (("Open..." :data "Open...")
         ("New"     :data "New"))
        :callback 'do-menu-item
        :callback-type :item)
       (a-menu
        "Another Menu"
        (("Open..." :data "Another Open")
         ("New" :data "Another New")
         ("Blancmange" :data "Blancmange"
                       :accelerator "accelerator-b"))
        :callback 'do-menu-item
        :callback-type :item))
      (:default-initargs
       :width 300
       :height 200))

    ;; This causes automatic accelerators on all platforms.

    ;; That is the default behavior on Microsoft Windows.
    (defmethod capi:interface-keys-style ((self mmm))
      :pc)

    (capi:contain (make-instance 'mmm))
```

These are further examples:

```
(example-edit-file "capi/applications/hangman")
```

```
(example-edit-file "capi/printing/fit-to-page")
```

See also

choice
interface-keys-style
menu
menu-component
1.2.1 CAPI elements
3.12 Tooltips
8 Creating Menus
9.4.1 Sharing toolbar callbacks with menu items

menu-object *Class*

Summary

The class **menu-object** is the superclass of all menu objects, and provides functionality for handling generic aspects of menus, menu components and menu items.

Package

capi

Superclasses

callbacks

Subclasses

titled-menu-object

Initargs

:popup-callback Callback before the menu appears.

:enabled-function Returns true if the menu is enabled.

:enabled-slot The object is enabled if the slot is non-nil.

:callback The selection callback for the object.

:callback-data-function
A function to return data for the callback.

:setup-callback-argument
If non-nil, specifies the argument to the setup callbacks (listed below) that are used to set up the **menu-object**.

:title The title for the object.

:title-function A setup callback which returns the title for the object, and optionally a mnemonic for the title.

Accessors

menu-popup-callback
menu-title
menu-title-function

Readers

menu-object-enabled

Description

The simplest way to give a title to a **menu-object** is to just supply a *title* string, and this will then appear as the title of the object.

Alternatively, a *title-function* can be provided which will be called when the menu is about to appear and which should return the title to use. By default *title-function* is called on the interface of the **menu-object**, but this argument can be changed by passing the initarg *setup-callback-argument*.

To specify a mnemonic in the title returned by *title-function*, make *title-function* return the mnemonic as a second value. This value is interpreted in the same way as the *mnemonic* argument for **menu**.

When the menu object is about to appear on the screen, the CAPI does the following:

1. The setup callback *items-function* (if there is one) is called and the result is used to set the items, for **menu** and **menu-component**. The argument passed to *items-function* is the same as for the other setup callbacks (see below).
2. The *popup-callback* (if there is one) is called and can make arbitrary changes to that object. The *popup-callback* is always called with the menu object, regardless of the value of *setup-callback-argument*.

3. The other setup callbacks are called to set up the selection, enabled state and title. These setup callbacks include *enabled-function* for all **`menu-object`**s and *title-function* for all **`titled-menu-object`**s. The additional setup callbacks for **`menu-component`** are *selection-function*, *selected-item-function*, and *selected-items-function*. **`menu-item`** has the additional setup callback *selected-function*.

   By default *setup-callback-argument* is **`nil`**, which means that each of the setup callbacks is called on the interface of the **`menu-object`**. If *setup-callback-argument* is non-nil, then it is passed (instead of the interface) as the argument to each of the setup callbacks.

4. The menu containing the object appears with all of the changes made.

Note that *enabled-slot* is a short-hand means of creating an *enabled-function* which checks the value of a slot in the menu object's interface.

The enabled state of a **`menu-object`** is computed each time the menu is displayed, using *enabled-function* or *enabled-slot*. Therefore the accessor **`menu-object-enabled`** is only useful as a reader.

The *callback* argument is placed in the *selection-callback*, *extend-callback* and *retract-callback* slots unless these are given explicitly, and so will get called when the menu object is selected or deselected.

The *callback-data-function* is a function that is called with no arguments and the value it returns is used as the data to the callbacks.

Notes

1. The function *enabled-function* should not display a dialog or do anything that may cause the system to hang. In general this means interacting with anything outside the Lisp image, including files, databases and so on.

2. The subclass **`titled-menu-object`** is retained only for backward compatibility.

Examples

```
(capi:contain (make-instance
               'capi:menu-item
               :text "Press Me"
               :enabled-function #'(lambda (item)
                                     (eq (random 2)
                                     1))))
```

The next example illustrates the use of *setup-callback-argument*. The **`initialize-instance`** method adds to the "Some Numbers" menu a sub-menu that lists the selected items in the **`list-panel`**. By using *setup-callback-argument* in this menu, the setup callbacks (in this case *enabled-function* and *items-function*) are called directly on the **`list-panel`**.

Note that, while this example uses a CAPI object as the *setup-callback-argument*, any object of any type can be used.

```
(capi:define-interface my-interface ()
  ()
  (:panes
   (list-panel
    capi:list-panel
    :items '(1 2 3 4 5 6 7 8 9 0)
    :interaction :extended-selection
    :visible-min-height '(character 10)))
  (:menus
   (a-menu
    "Some Numbers"
    ("One" "Two")
    ))
  (:menu-bar a-menu))
```

```
(defmethod initialize-instance :after
  ((self my-interface) &key)
  (with-slots (a-menu list-panel) self
    (setf (capi:menu-items a-menu)
          (append
           (capi:menu-items a-menu)
           (list
            (make-instance 'capi:menu
                           :items-function
                           'capi:choice-selected-items
                           :setup-callback-argument
                           list-panel
                           :enabled-function
                           'capi:choice-selection
                           :title
                           "Selected Items"))))))

(capi:display (make-instance 'my-interface))
```

See also

menu
menu-item
menu-component

merge-menu-bars *Generic Function*

Summary

Computes the menu bar for a **document-frame** on Microsoft Windows.

Package

capi

Signature

merge-menu-bars *frame document* **=>** *menus*

Arguments

frame⇓	A **document-frame**.
document⇓	An **interface** or **nil**.

Values

menus	A list of **menu** objects.

Description

The generic function **merge-menu-bars** is called by the system to compute the menu bar for a **document-frame** interface *frame*.

The set of visible menus in such an interface is typically made up from those of *frame* and those of the active document *document* within it.

There is a built-in unspecialized method that appends the menu bars of the two interfaces and is equivalent to this:

```
(defmethod capi:merge-menu-bars ((frame t)
                                  (document t))
  (append
   (capi:interface-menu-bar-items frame)
   (and document
        (capi:interface-menu-bar-items document))))
```

You can customize the menu bar by adding methods which specialize on particular frame and document interface classes.

Notes

merge-menu-bars is implemented only in LispWorks for Windows.

See also

document-frame
interface
menu

message-pane *Class*

Summary

The class displaying the message when a pane is created with the **:message** initarg.

Package

capi

Superclasses

title-pane

Description

The class **message-pane** is used to implement the message decoration on subclasses of **titled-object**.

A **message-pane** with *text* "Message" is created automatically when a **titled-object** is created with *message* "Message".

Notes

message-pane does not add functionality to **title-pane**, and it is used only to allow different resources in GTK+ and Motif.

See also

titled-object

metafile-port *Class*

Summary

A graphics port created by **with-external-metafile** and **with-internal-metafile**.

Package

capi

Superclasses

graphics-port-mixin

Description

The class **metafile-port** is the graphics port that **with-external-metafile** and **with-internal-metafile** create when their *pane* argument is not supplied.

See also

with-external-metafile
with-internal-metafile

modify-editor-pane-buffer *Function*

Summary

Modifies the contents and fill mode of a specified buffer.

Package

capi

Signature

modify-editor-pane-buffer *pane* **&key** *contents flag fill fixed-fill force*

Arguments

pane⇓	An **editor-pane**.
contents⇓	A string or **nil**.
flag⇓	A keyword.
fill⇓	A boolean, with special meaning for a fixnum and **:default**.
fixed-fill⇓	An integer or **nil**.
force⇓	A generalized boolean.

Description

The function **`modify-editor-pane-buffer`** modifies the **`editor-pane`** *pane* according to the keyword arguments.

contents (if non-nil) supplies a new string to place in the buffer.

If *fill* is non-nil the editor fills each paragraph in the buffer. If *fill* is a fixnum then the buffer is filled at that width. If *fill* is **`:default`** (the default value) and *fixed-fill* is supplied then the value *fixed-fill* is used. Otherwise the buffer is filled to the window width.

fixed-fill defaults to **`nil`**.

If *force* is true (the default), then an editor buffer is created for *pane* if it does not have one yet. If *force* is false then **`modify-editor-pane-buffer`** will signal an error if *pane* does not have an editor buffer.

Notes

The argument *flag* is deprecated. You can supply the initarg **`:flag`** when creating an **`editor-pane`**.

See also

`editor-pane`

modify-multi-column-list-panel-columns *Function*

Summary

Modify the columns of a **`multi-column-list-panel`**.

Package

`capi`

Signature

`modify-multi-column-list-panel-columns` *self* **`&key`** *columns x-adjust reorderable-columns sort-descriptions column-function item-print-functions*

Arguments

self⇓	A **`multi-column-list-panel`**.
columns⇓	A list of column specifications.
x-adjust⇓	A list.
reorderable-columns⇓	A list.
sort-descriptions⇓	A list.
column-function⇓	A function of one argument or a list of functions of one argument. The default is **`identity`**.
item-print-functions⇓	A function of one argument, or a list of such functions.

Description

The function **`modify-multi-column-list-panel-columns`** modifies the columns of *self*.

All the keyword arguments have the same meaning as the corresponding initargs in **`multi-column-list-panel`**. See the entry for **`multi-column-list-panel`** for details.

For all the keyword arguments except *x-adjust* and *reorderable-columns*, if they are not supplied then the value does not change. For all keyword arguments except *sort-descriptions*, *x-adjust* and *reorderable-columns*, if they are passed as **`nil`** then the corresponding value does not change. If *sort-descriptions* is passed as **`nil`**, then the *sort-descriptions* are changed to **`nil`**.

Notes

1. *columns* and *column-function* need to match, so normally you modify them both. Supplying *column-function* as a list of functions makes it easier to match, by just making *column-function* a list parallel to *columns*.
2. An alternative solution is to use a *column-function* that decides dynamically what values to return based on some value that you set when you call **`modify-multi-column-list-panel-columns`**. For example you can make *column-function* a function that closes over the containing interface, and check a slot in it to decide which columns to return, and then update this slot whenever you call **`modify-multi-column-list-panel-columns`**.
3. If *item-print-functions* is a list, it will also have to be updated when *columns* are updated.
4. If *columns* is supplied then *x-adjust* and *reorderable-columns* are also used to modify the columns, so you might need to supply them as well. *x-adjust* and *reorderable-columns* are ignored if *columns* is not supplied.
5. Since *sort-descriptions* are searched, they do not need to be updated when *columns* is updated, provided that they already contain all the sort kinds that any column may use.

See also

`multi-column-list-panel`

modify-stacked-tree *Function*

Summary

Modify several properties of a **`stacked-tree`** at the same time.

Package

`capi`

Signature

`modify-stacked-tree` *stacked-tree* **`&key`** *root value max-level item-function*

Arguments

stacked-tree⇓ A **`stacked-tree`**.

root⇓, *value*⇓, *max-level*⇓, *item-function*⇓
See the initargs of **`stacked-tree`**.

Description

The function **`modify-stacked-tree`** can be used to modify several properties in *stacked-tree* at the same time. Most importantly, it allows you to set the properties that you are likely to want to change at the time you set the root.

Setting *max-level* and *item-function* has no effect until the next time the root is set. If you want to set one or both of them for the existing root, just supply the **`:root`** keyword with the current root using **`stacked-tree-root`**.

Supplying *root* or *value* has an immediate effect, and *stacked-tree* is redrawn with the new setting. When supplying *root*, this means recomputing the whole tree, which may take enough time to cause a noticeable delay.

For keywords that are not supplied, the corresponding properties do not change.

`modify-stacked-tree` can be called before *stacked-tree* is displayed, but will not have any affect until then.

See also

`stacked-tree`

mono-screen *Class*

Summary

A class for monochrome screen.

Package

`capi`

Superclasses

`screen`

Description

Instances of the class **`mono-screen`** are created for monochrome screens. It is available primarily as a means of discriminating on whether or not to use colors in an interface.

See also

`color-screen`

move-line *Generic Function*

Summary

Moves a **`line-pinboard-object`**.

Package

`capi`

Signature

`move-line` *line-pinboard-object start-x start-y end-x end-y* **`&key`** *redisplay*

Arguments

line-pinboard-object⇓	An instance of **`line-pinboard-object`** or a subclass.
start-x⇓	The x coordinate of the start of the line.
start-y⇓	The y coordinate of the start of the line.
end-x⇓	The x coordinate of the end of the line.
end-y⇓	The y coordinate of the end of the line.
redisplay⇓	A boolean.

Description

The generic function **`move-line`** moves *line-pinboard-object* to a new location with start and end points specified by *start-x*, *start-y*, *end-x* and *end-y*.

This automatically adjusts the geometry of the object, taking into account other constraints. Examples of such constraints are the label in a **`labelled-line-pinboard-object`** and the arrowhead in a **`arrow-pinboard-object`**.

The default value of *redisplay* is **`t`**, which means that the changed line is redrawn immediately. If you are moving many objects at the same time, it is useful to pass **`:redisplay nil`**.

See also

`line-pinboard-object`
`line-pinboard-object-coordinates`

multi-column-list-panel *Class*

Summary

A list panel with multiple columns of text.

Package

`capi`

Superclasses

`list-panel`

Initargs

`:column-function`	A function of one argument or a list of functions of one argument. The default is **`identity`**.
`:item-print-functions`	A function of one argument, or a list of such functions.
`:columns`	A list of column specifications.

`:header-args` A plist of keywords and values.

`:auto-reset-column-widths`
A boolean. The default is **`t`**.

`:reorderable-columns`
A boolean. The default is **`nil`**.

`:x-adjust` A list. The default is **`nil`**.

Description

The class **`multi-column-list-panel`** is a list panel which displays multiple columns of text. The columns can each have a title.

Note that this is a subclass of **`list-panel`**, and hence of **`choice`**, and inherits the behavior of those classes.

Each item in a **`multi-column-list-panel`** is displayed in a line of multiple objects. The corresponding objects of each line are aligned in a column.

The *column-function* generates the objects for each item. It should take an item as its single argument and return a list of objects to be displayed. The default *column-function* is **`identity`**, which works if each item is a list.

column-function can also be a list of function designators. In this case the length has to match the length of the *columns*. Each function is called with the item to generate the object for the corresponding column.

The *item-print-functions* argument determines how to calculate the text to display for each element. If *item-print-functions* is a single function, it is called on each object, and must return a string. Otherwise *item-print-functions* should be a sequence of length no less than than the number of columns. The text to display for each object is the result (again, a string) of calling the corresponding element of *item-print-functions* on that object.

The *columns* argument specifies the number of columns, and whether the columns have titles and callbacks on these titles.

Each element of *columns* is a specification for a column. Each column specification is a plist of keyword and values, where the allowed keywords are as follows:

`:title` Specifies the title to use for the column. If any of the columns has a title, a header object is created which displays the titles. The values of the **`:title`** keywords are passed as the *items* of the header, unless *header-args* specifies **`:items`**.

`:adjust` Specifies how to adjust the column. The value can be one of **`:right`**, **`:left`**, or **`:center`**.

`:width` Specifies a fixed width of the column.

`:default-width` Specifies the default initial width of the column. The user can resize it. If **`:width`** is supplied it overrides **`:default-width`**.

`:visible-min-width` Minimum width of the column.

`:gap` Specifies an additional gap alongside the text in the column. **`:gap`** is not supported consistently across platforms (see Notes below).

`:reorderable` Has an effect only on GTK+. When non-nil, **`:reorderable`** specifies that the column is reorderable, that is the user can drag the header of the column to move the column to another position. Note that the initarg *reorderable-columns* forces all columns to be reorderable, overriding any **`:reorderable`** value in the column specification.

:separator A boolean specifying whether the column has a separator from its previous column. The first column never has a separator. For columns that do not have this keyword, whether they have a separator is determined by the initarg **:separators**, which is inherited from **list-panel** and defaults to **nil**.

The values of **:width**, **:visible-min-width** and **:gap** are interpreted as standard geometric hints. See **element** for information about these hints.

columns should indicate how many columns to display. At a minimum, the value needs to be **(() ())** for two columns without any titles.

header-args is a plist of initargs passed to the header which displays the titles of the columns. The header object is a **collection**. The following **collection** initargs are useful to pass in *header-args*:

:selection-callback

A callback function for clicking on the header, or the keyword **:sort** which specifies sorting as described below.

:callback-type Defines the arguments of the *selection-callback*.

:items The items of the header object, that is the titles. Note that **:items** overrides **:title** if that is supplied in *columns*.

:print-function Controls how each of *items* is printed, providing the title of each column.

header-args may also contain the keyword **:alignments**. The value should be a list of alignment keywords, each of which is interpreted like an **:adjust** value in *columns*. The alignment is applied to the title only.

When the callback is **:sort**, clicking on a header causes a call to **sorted-object-sorted-by** on the pane, with *sort-type* the title of the column, as given either by **:items** or **:title** in the columns. To make it work, you also need to define the *sort-definitions*, by making the pane with *sort-descriptions* with types that match the *titles* (see **sorted-object** and **make-sorting-description**).

If *auto-reset-column-widths* is true, then the widths of the columns are recomputed when the items of the **multi-column-list-panel** are set.

reorderable-columns has effect only on GTK+. When *reorderable-columns* is non-nil, it makes all the columns in the **multi-column-list-panel** reorderable, so the user can change their order by dragging the header of a column. Note that you can also make only some columns reorderable by not using *reorderable-columns*, and instead using **:reorderable** in the column specification.

If *x-adjust* is non-nil, then it specifies the *adjust* values for each columns, that is it has the same effect as having **:adjust** in each column specification with the value being the matching item in *x-adjust*. *x-adjust* (when non-nil) must be a list of the same length as *columns*, where each item is one of the keywords that **:adjust** in the column specification can accept. The value in *x-adjust* overrides any **:adjust** given in the column specification.

Notes

1. Similar and enhanced functionality is provided by **list-view**.
2. On Microsoft Windows, **:width** in a column specification does not actually make the column width be fixed, though it does supply the initial width.
3. On Microsoft Windows, **:gap** in a column specification adds the gap on both sides of the text. On Motif it adds the gap only on the right side of the text. On GTK+ and Cocoa **:gap** is ignored.

4. The number of columns in a **`multi-column-list-panel`**, their titles and what they show can be changed after the pane is displayed using **`modify-multi-column-list-panel-columns`**.

Examples

This example uses the *columns* initarg:

```
(capi:contain
 (make-instance
  'capi:multi-column-list-panel
  :visible-min-width 300
  :visible-min-height :text-height
  :columns '((:title "Fruits"
              :adjust :right
              :width (character 15))
             (:title "Vegetables"
              :adjust :left
              :visible-min-width (character 30)))
  :items '(("Apple" "Artichoke")
           ("Pomegranate" "Pumpkin"))))
```

This example uses *header-args* to add callbacks and independent alignment on the titles:

```
(defun mclp-header-callback (interface item)
  (declare (ignorable interface))
  (capi:display-message "Clicked on ~a" item))

(capi:contain
 (make-instance
  'capi:multi-column-list-panel
  :visible-min-width 300
  :visible-min-height :text-height
  :columns '((:adjust :right
              :width (character 15))
             (:adjust :left
              :visible-min-width (character 30)))
  :header-args '(:items ( "Fruits" "Vegetables")
                 :selection-callback
                 mclp-header-callback
                 :alignments (:left :right))
  :items '(("Apple" "Artichoke")
           ("Pomegranate" "Pumpkin"))))
```

This example file illustrates the use of the header's *selection-callback* **`:sort`** to implement sorting of the columns:

```
(example-edit-file "capi/choice/multi-column-list-panels")
```

This example uses *column-function* to implement a primitive process browser:

```
(defun get-process-elements (process)
  (list (mp:process-name process)
        (mp:process-whostate process)
        (mp:process-priority process)))

(capi:contain
 (make-instance
  'capi:multi-column-list-panel
  :visible-min-width '(character 70)
  :visible-min-height '(character 15)
  :items (mp:list-all-processes)
  :columns '((:title "Name" :adjust :left
              :visible-min-width (character 30))
```

```
                (:title "State" :adjust :center
                 :visible-min-width (character 20))
                (:title "Priority" :adjust :center
                 :visible-min-width (character 12)))
      :column-function 'get-process-elements))
```

There are further examples in **20 Self-contained examples**.

See also

`collection`
`list-panel`
`list-view`
`make-sorting-description`
`modify-multi-column-list-panel-columns`
`sorted-object-sorted-by`
5.3.7 Multi-column list panels

multi-line-text-input-pane *Class*

Summary

A pane allowing several lines of text to be entered.

Package

`capi`

Superclasses

`text-input-pane`

Description

The class **`multi-line-text-input-pane`** behaves like a **`text-input-pane`**, except that the text entered by the user is allowed to span several lines — that is, it is allowed to contain Newline characters.

See also

`text-input-pane`
3.5 Displaying and entering text

non-focus-list-add-filter
non-focus-list-remove-filter
non-focus-list-toggle-filter *Functions*

Summary

Add or remove the filter in a non-focus list.

Package

capi

Signatures

non-focus-list-add-filter *non-focus-list-interface*

non-focus-list-remove-filter *non-focus-list-interface*

non-focus-list-toggle-filter *non-focus-list-interface*

Arguments

non-focus-list-interface⇓

A **non-focus-list-interface**.

Description

These functions add or remove the filter in a non-focus list *non-focus-list-interface*.

non-focus-list-toggle-filter calls **non-focus-list-add-filter** if the filter is off, otherwise it calls **non-focus-list-remove-filter** (it is used as the callback for the *filtering-gesture*).

non-focus-list-add-filter adds a filter is it is not already on, resets the text in it to empty string, and enables it.

non-focus-list-remove-filter removes the filter if it is on.

See also

prompt-with-list-non-focus

non-focus-list-interface *Class*

Summary

Created (and destroyed) only by **prompt-with-list-non-focus** and **text-input-pane-in-place-complete**.

Package

capi

Superclasses

interface

Description

The class **non-focus-list-interface** is the class of interface created and destroyed only by **prompt-with-list-non-focus** and **text-input-pane-in-place-complete**. Do not instantiate this class directly.

See also

prompt-with-list-non-focus

text-input-pane-in-place-complete

non-focus-list-toggle-enable-filter *Function*

Summary

Toggles the enabled state of the filter.

Package

capi

Signature

non-focus-list-toggle-enable-filter *non-focus-list-interface*

Arguments

non-focus-list-interface⇓
A **non-focus-list-interface**.

Description

The function **non-focus-list-toggle-enable-filter** toggles the enabled state of the filter in a non-focus list *non-focus-list-interface* created by **prompt-with-list-non-focus** or **text-input-pane-in-place-complete**. It has no effect if the filter is off.

It is used as the callback of the *filtering-toggle*.

See also

prompt-with-list-non-focus

non-focus-maybe-capture-gesture *Function*

Summary

Maybe capture a gesture by a **non-focus-list-interface**.

Package

capi

Signature

non-focus-maybe-capture-gesture *non-focus-list-interface* *gesture* **=>** *result*

Arguments

non-focus-list-interface⇓
A **non-focus-list-interface**.

gesture⇓ A gesture specifier.

Values

result A generalized boolean.

Description

The function **non-focus-maybe-capture-gesture** is used to pass input gestures to a **non-focus-list-interface** that was created by **prompt-with-list-non-focus**.

A **non-focus-list-interface**, by its nature, does not receive keyboard input, but in most of cases it is still useful if it responds to some gestures. For that to happen, **non-focus-maybe-capture-gesture** must be called.

gesture must be a gesture specifier, which is an object that can be coerced to a **gesture-spec** by **sys:coerce-to-gesture-spec**.

Currently **non-focus-maybe-capture-gesture** does the following:

1. If *gesture* is **Escape**, it calls **non-focus-terminate** on *non-focus-list-interface*.
2. It checks whether the gesture matches any of the gestures in the gesture callbacks of *non-focus-list-interface*. The gesture callbacks are either explicitly defined using the keyword arguments **:gesture-callbacks** or **:add-gesture-callbacks** in **prompt-with-list-non-focus**, or implicitly. By default, all the gestures that are used in in-place completion (see **10.6 In-place completion**) are defined implicitly. These include **Up**, **Down**, **PageUp**, **PageDown** (change selection in the list panel), **Return** (invoke the **:action-callback**), **Control+Return** and **Control+Shift+Return** (control of the filter in the list panel). The implicitly defined gestures are affected by the keywords **:gesture-callbacks**, **:filtering-gesture** and **:filtering-toggle** in **prompt-with-list-non-focus**.

   If a match is found, it is invoked as described for *gesture-callbacks* in **prompt-with-list-non-focus**.
3. If filtering is enabled, it checks if the gesture is captured by the filter. A gesture is captured by the filter if it is:

   A plain graphic character.

   It is inserted to the filter.

   Backspace The last character in the filter is deleted.

   One of the gestures that update the state of the filter (by default **Control+Shift+R**, **Control+Shift+E**, **Control+Shift+C**)

   The state of the filter is updated.

In any case, where a gesture is captured by the filter the list panel is updated.

If the gesture is captured by one of the possibilities above, **non-focus-maybe-capture-gesture** returns **t**, otherwise it returns **nil**.

See also

non-focus-terminate
prompt-with-list-non-focus

non-focus-terminate *Generic Function*

Summary

Terminates the non-focus interface.

Package

capi

Signature

non-focus-terminate *non-focus-interface*

Method signatures

non-focus-terminate (*non-focus-interface* **non-focus-list-interface)**

Arguments

non-focus-interface⇓ A **non-focus-list-interface**.

Description

The generic function **non-focus-terminate** closes the non-focus interface *non-focus-interface*.

It has no return value.

The method terminates a **non-focus-list-interface**. It destroys the interface in the correct process.

See also

prompt-with-list-non-focus

non-focus-update *Generic Function*

Summary

Updates the non-focus-interface.

Package

capi

Signature

non-focus-update *non-focus-interface*

Method signatures

non-focus-update (*non-focus-interface* **non-focus-list-interface)**

Arguments

non-focus-interface⇓ A **non-focus-list-interface**.

Description

The generic function **non-focus-update** updates the non-focus-interface *non-focus-interface*.

It has no return value.

The method on **non-focus-list-interface** needs to be invoked in the process in which the *list-updater* that was passed to **prompt-with-list-non-focus** is expecting to run.

It invokes the *list-updater* without arguments, and then updates the non-focus-interface with result. See the description of *list-updater* in **prompt-with-list-non-focus**.

Note that if *list-updater* returns **:destroy**, this invokes **non-focus-terminate** on the interface.

See also

prompt-with-list-non-focus
non-focus-terminate

ole-control-add-verbs *Function*

Summary

Adds to the menu entries for the "verbs" that a component in an **ole-control-pane** supports.

Package

capi

Signature

ole-control-add-verbs *pane menu item-identifier*

Arguments

pane⇓ An **ole-control-pane**.

menu⇓ A **menu**.

item-identifier⇓ A string or symbol.

Description

The function **ole-control-add-verbs** adds to the menu entries for the "verbs" that the component supports. The **ole-control-pane** *pane* must have an object already, and the **menu** *menu* must have already been created, so **ole-control-add-verbs** is typically called in the *popup-callback* of *menu*.

item-identifier identifies an item in the menu or a component in the menu (but not in a sub-menu), either by being **cl:eq** to the name of the item or **cl:equalp** to the title of the item. If the item is found, it is replaced either by a sub-menu with the verbs that the object supports, or, if the object supports only one verb, by an entry for this.

When the user selects an added menu item, the verb is passed to the object (by a call to **IOleObject::DoVerb**).

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **`(require "embed")`**.

See also

`menu`
`ole-control-pane`

ole-control-close-object *Function*

Summary

Closes the object in an **`ole-control-pane`**.

Package

`capi`

Signature

`ole-control-close-object` *pane*

Arguments

pane⇓	An **`ole-control-pane`**.

Description

The function **`ole-control-close-object`** closes the object that is currently in the **`ole-control-pane`** *pane*.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **`(require "embed")`**.

See also

`ole-control-pane`

ole-control-component *Class*

Summary

An implementation of the interfaces in the OLE Control protocol.

Package

`capi`

Superclasses

`com:standard-i-unknown`

Initargs

`:pane-function`	A function that is called when OLE embeds the Control in a container.
`:create-callback`	A function called just after the pane is created.
`:destroy-callback`	A function called just before the pane is destroyed.

Readers

`ole-control-component-pane`

Description

The class **`ole-control-component`** provides an implementation of the interfaces in the OLE Control protocol, to allow a CAPI pane to be embedded in an OLE Control container implemented outside LispWorks. It is typically used with the macro **`define-ole-control-component`** to define a subclass of **`ole-control-component`** that implements a particular coclass from a type library. Instances of this class are usually created by the COM run time system, not by explicit calls to **`make-instance`**.

A function designator *pane-function* must be supplied. *pane-function* that is called when OLE embeds the Control in a container. It receives the component as its argument and should return a CAPI pane that will implement the visual aspects of the control.

Note: The pane returned by *pane-function* must be a **`output-pane`**, **`layout`** or **`interface`** in the current implementation. The pane is stored in the component and can be accessed using the reader **`ole-control-component-pane`**.

create-callback, if non-nil, is a function called when the pane returned by *pane-function* has been created in the window system. The argument is the pane itself. *create-callback* can perform initialization such as loading images.

destroy-callback, if non-nil, is a function called when the pane returned by *pane-function* is going to be destroyed. The argument is the pane itself. *destroy-callback* can perform cleanups.

Notes

When using an **`ole-control-component`**, the normal hierarchy of CAPI objects such as a layout and an interface do not exist above it. The layout and control of the top level window is the responsibility of the application that embeds the control. It can communicate with the control by using COM/Automation.

`ole-control-component` is implemented only in LispWorks for Windows. Load the functionality by **`(require "embed")`**.

See also

`define-ole-control-component`

ole-control-doc *Class*

Summary

A class that implements the document around the object inside an **ole-control-pane**.

Package

capi

Superclasses

pinboard-layout

Subclasses

ole-control-frame

Description

The class **ole-control-doc** is a CAPI pane that can be used to implement the document around the object inside an **ole-control-pane**. That is, it supports the **IOleInPlaceUIWindow** interface. Note that this is optional, and is rarely useful.

To use it the **ole-control-doc** pane needs to be the parent, not necessarily directly, of an **ole-control-pane**. When the object calls **IOleInPlaceSite::GetWindowContext**, it will get (in the **ppdoc [out]** argument) an **IOleInPlaceUIWindow** interface associated with the **ole-control-doc**.

A **ole-control-doc** must have exactly one sub-pane (that is, the length of its *description* must be 1), but underneath this pane there can be many panes.

Normally the program does not need to do anything else with the **ole-control-doc**. It acts in response to resizing of the window and method calls from the object on the **IOleInPlaceUIWindow** interface.

Notes

ole-control-doc is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

Even though it is a subclass of **pinboard-layout**, normally you should not use the **pinboard-layout** functionality when using **ole-control-doc**.

See also

ole-control-pane

ole-control-frame *Class*

Summary

Implements the frame of components in an **ole-control-pane**.

Package

capi

Superclasses

ole-control-doc

Description

The class **ole-control-frame** is a CAPI pane that implements the frame of components, that is it supports the **IOleInPlaceFrame** interface. When an **ole-control-pane** pane is created, it looks upwards in the hierarchy of panes, and if finds an **ole-control-frame** pane it uses this as the frame. It uses the first such pane found. When the object in the **ole-control-pane** calls **IOleInPlaceSite::GetWindowContext**, it gets back in the **ppframe arg** an interface associated with this frame.

Like **ole-control-doc**, a **ole-control-frame** can have only one sub-pane, which itself may contain many panes.

Normally the program does not need to do anything else with the **ole-control-frame**. It acts in response to resizing of the window and method calls from the object on the **IOleInPlaceFrame** interface.

Note that having a frame is optional, and ActiveX does not need it. It is required when embedding an application by **ole-control-insert-object**.

Notes

ole-control-frame is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

Even though it is a subclass of **pinboard-layout**, normally you should not use the **pinboard-layout** functionality when using **ole-control-frame**.

See also

ole-control-insert-object
ole-control-pane

ole-control-i-dispatch *Function*

Summary

Returns the **com:i-dispatch** of the component of an **ole-control-pane**.

Package

capi

Signature

ole-control-i-dispatch *pane* **=>** *result*

Arguments

pane⇓	An **ole-control-pane**.

Values

result A **com:i-dispatch** or **nil**.

Description

The function **ole-control-i-dispatch** returns the **com:i-dispatch** (that is, the **IDispatch** interface) of the component within *pane*, or **nil** if none exists. The **com:i-dispatch** is the one that would be returned by **com:query-interface** on the **com:i-ole-object**.

Notes

Calling **ole-control-i-dispatch** does not affect the reference count of the interface.

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

ole-control-pane

ole-control-insert-object *Function*

Summary

Embeds a user-specified document in an **ole-control-pane**.

Package

capi

Signature

ole-control-insert-object *pane*

Arguments

pane⇓ An **ole-control-pane**.

Description

The function **ole-control-insert-object** prompts the user for a document using the Microsoft Windows function **OleUIInsertObject**.

When the user specifies a document in the dialog presented, **ole-control-insert-object** embeds this document in the **ole-control-pane** *pane*.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

ole-control-pane

ole-control-ole-object *Function*

Summary

Returns the **com:i-ole-object** of the component of an **ole-control-pane**.

Package

capi

Signature

ole-control-ole-object *pane* **=>** *result*

Arguments

pane⇓ An **ole-control-pane**.

Values

result A **com:i-ole-object** or **nil**.

Description

The function **ole-control-ole-object** returns the **com:i-ole-object** (that is, the **IOleObject** interface) of the component of the **ole-control-pane** *pane*, or **nil** if none exists.

Notes

Calling **ole-control-ole-object** does not affect the reference count of the interface.

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

ole-control-pane

ole-control-pane *Class*

Summary

A class that implements embedding of external components on Microsoft Windows.

Package

capi

Superclasses

pinboard-layout

Initargs

`:component-name`	A string or **`nil`**.
`:user-component`	A COM interface pointer or **`nil`**.
`:save-name`	A string.
`:insert-callback`	A function.
`:close-callback`	A function.
`:sinks`	A list of sink specifications.

Description

The class **`ole-control-pane`** is used to implement embedding of external components.

Note: **`ole-control-pane`** is implemented only in LispWorks for Windows. Load the functionality by **`(require "embed")`**.

Note: even though it is a subclass of **`pinboard-layout`**, normally you should not use the **`pinboard-layout`** functionality when using **`ole-control-pane`**.

component-name (if non-nil) specifies the *component-name* of the pane, as used by **`component-name`**.

user-component (if non-nil) is a COM interface pointer of an object that supports the **`com:i-ole-object`** interface, and is ready to display as described in **`ole-control-user-component`**.

save-name is used when creating the **`IStorage`** object for this component.

insert-callback (if non-nil) is a function that takes a single argument, the pane. It is called immediately after a component was inserted into the pane. This can be used for any additional initialization that is required, for example setting the properties of the control.

close-callback (if non-nil) is a function that takes a single argument, the pane. It is called just before the component is going to be closed, and can be used to do any cleanups that may be required.

sinks is a list of sink specifications for attaching event handlers to the source interfaces of the control. Each element of *sinks* should be a list of the form:

(*interface-name* **&key** *invoke-callback sink-class sink***)**

The *interface-name* is used to specify the name of the source interface in the control, which is either a string naming the interface or **`:default`** for the default source interface. If *invoke-callback* is given, then it should be a function which will be called with the pane, method-name, method-kind and arguments vector for each source event. The *sink-class* can be given to set the class of the internal object used for the sink interface. This is similar to calling **`attach-simple-sink`**. Alternatively, instead of calling *invoke-callback*, the *sink* can be specified directly. This is similar to calling **`attach-sink`**.

When the **`ole-control-pane`** is destroyed, the sinks are automatically detached.

There are currently three ways to insert an external component into an **`ole-control-pane`**. These are:

1. Call **`ole-control-user-component`**, which asks the user for something to insert.
2. Set the *component-name* of the pane. This can be done either via the initarg **`:component-name`** or by calling **`(setf component-name)`**.
3. Set the *user-component* of the pane, either via the initarg **`:user-component`** or by calling **`(setf ole-control-user-component)`**.

Examples

```
(capi:contain
 (list
  (make-instance 'capi:ole-control-pane
                 :component-name "OWC.Spreadsheet.9")))
```

This is a full example:

```
(example-edit-file "com/ole/html-viewer")
```

See also

attach-simple-sink
attach-sink
component-name
detach-sink
interface-menu-groups
ole-control-add-verbs
ole-control-close-object
ole-control-i-dispatch
ole-control-insert-object
ole-control-ole-object
ole-control-pane-frame
ole-control-user-component
report-active-component-failure

ole-control-pane-frame *Function*

Summary

Returns the **ole-control-frame** of an **ole-control-pane**.

Package

capi

Signature

ole-control-pane-frame *pane* **=>** *result*

Arguments

pane⇓	An **ole-control-pane**.

Values

result	An **ole-control-frame** or **nil**.

Description

The function **ole-control-pane-frame** returns the **ole-control-frame** of the **ole-control-pane** *pane*, if there is one.

Note: this function is implemented only in LispWorks for Windows. Load the functionality by **`(require "embed")`**.

See also

`ole-control-frame`
`ole-control-pane`

ole-control-pane-simple-sink *Class*

Summary

A class that implements a sink interface for an embedded component on Microsoft Windows.

Package

`capi`

Superclasses

`com:simple-i-dispatch`

Initargs

`:ole-control-pane` A class instance.

Description

The class **`ole-control-pane-simple-sink`** is used by the function **`attach-simple-sink`** to implement a sink interface for an embedded component on Microsoft Windows.

ole-control-pane is the object of type **`ole-control-pane`** to whose source interface the sink is being attached.

This class can be subclassed to provide additional functionality in callbacks. See **`com:simple-i-dispatch`** in the *COM/Automation User Guide and Reference Manual* for more details.

Note: **`ole-control-pane-simple-sink`** is implemented only in LispWorks for Windows. Load the functionality by **`(require "embed")`**.

See also

`attach-simple-sink`
`ole-control-pane`

ole-control-user-component *Accessor*

Summary

Gets and sets the *user-component* of an **`ole-control-pane`**.

Package

`capi`

Signature

ole-control-user-component *pane* **=>** *user-component*

(setf ole-control-user-component) *user-component* *pane* **=>** *user-component*

Arguments

pane⇓ An **ole-control-pane**.

user-component⇓ A COM **com:i-ole-object** interface pointer or **nil**.

Values

user-component⇓ A COM **com:i-ole-object** interface pointer or **nil**.

Description

The accessor **ole-control-user-component** gets and sets the *user-component* of the **ole-control-pane** *pane*.

user-component (if non-nil) is a COM interface pointer of an object that supports the **com:i-ole-object** interface, and has been opened and initialized and is ready to be displayed. This is typically created by calling **OleCreate**, **OleCreateFromFile**, **OleCreateFromData** or **OleLoad** with *pCLientSite* null.

user-component will be closed and released by the **ole-control-pane** *pane*, so after you have called **(setf ole-control-user-component)** you should not try to use it again or release it. Setting *user-component* also sets the pane's *component-name* to **nil**.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

ole-control-pane

option-pane *Class*

Summary

A pane which offers a choice of items, but which displays only the currently selected item.

Package

capi

Superclasses

choice
titled-object
simple-pane

Initargs

:enabled	Non-nil if the option pane is enabled.
:visible-items-count	An integer or the symbol **:default**.
:popup-callback	A function called just before the popup menu appears, or **nil**.
:popdown-callback	
:image-function	A function providing images for items, or **nil**.
:image-lists	A plist of keywords and **image-list** objects.
:separator-item	An item that acts as a separator between other items, or **nil**.
:enabled-positions	A list of fixnums, or the keyword **:all**.
:window-styles	A list of keywords.

Accessors

option-pane-enabled
option-pane-image-function
option-pane-visible-items-count
option-pane-popup-callback
option-pane-separator-item
option-pane-enabled-positions

Description

The class **option-pane** provides a pane which offers a choice between a number of items via a popup menu. Only the currently selected item is displayed.

The class **option-pane** inherits from **choice**, and so has all of the standard choice behavior such as selection and callbacks. It also has an extra *enabled* slot along with an accessor which is used to enable and disable the option pane.

visible-items-count is implemented only on Microsoft Windows. If *visible-items-count* is an integer then the popup menu is no longer than this, and is scrollable if there are more items. If *visible-items-count* is **:default**, then the popup menu is no longer than 10. This is the default value.

When *popup-callback* is non-nil, it should be a function of one argument that will be called just before the popup menu appears when the user clicks on it. The single argument to the function is the option pane and the return value is ignored. If required, the function can change the items or selection of the pane. The default value of *popup-callback* is **nil**.

If *image-function* is non-nil, it should be a function of one argument which is called with each item. The return value depends on *image-lists*. If *image-lists* contains an **image-list** for the **:normal** key, then the result of *image-function* should be one of the following:

A pathname or string	This specifies the filename of a file suitable for loading with **load-image**. Currently this must be a bitmap file.
A symbol	The symbol must have been previously registered by means of a call to **register-image-translation**.
An **image** object	For example, as returned by **load-image**.
An image locator object	This allowing a single bitmap to be created which contains several button images side by side. See **make-image-locator** for more information. On Microsoft Windows, it also allows access to bitmaps stored as resources in a DLL.

An integer — This is a zero-based index into the option-pane's **image-list**. This is generally only useful if the image list is created explicitly. See **image-list** for more details.

Otherwise if there is no **image-list** then it should return one of:

nil — No image is shown.

An **image** object — The pane displays this image.

An image id or an **external-image** object

The system converts the value to a temporary **image** for the item and frees it when it is no longer needed.

If *image-function* is **nil**, no items have images. This is the default value..

If *image-lists* is specified, it should be a plist containing the keyword **:normal** as a key. The corresponding value should be an **image-list** object. No other keys are supported at the present time. The **image-list** associated with the **:normal** key is used with the *image-function* (see above) to specify an image to display in each tab.

separator-item should be an item (compared using *test-function*) that acts as a separator between other items. A separator item is not selectable. The default value **nil** means that there are no separators (regardless of *test-function*).

If *enabled-positions* is **:all** then all the items can be selected. Otherwise the value is a list of fixnums indicating the positions in the item list which can be selected. The default value is **:all**.

On Microsoft Windows, if *window-styles* contains the keyword **:simple-text-only**, then the **option-pane** is displayed using the UI theme and the *enabled-positions*, *separator-item*, *image-function* and *visible-items-count* initargs are not supported. Otherwise it is displayed without the UI theme and those options work as documented. This is a limitation in Microsoft Windows.

Notes

1. The user cannot edit the items in an **option-pane**. For an element with similar functionality which allows editing, see **text-input-choice**.
2. **:image-function** and **:image-lists** are currently only implemented for Microsoft Windows, GTK+ and Cocoa.
3. On Motif, the separator is represented simply as a blank item between the other items.
4. On Motif and GTK+ versions older than 2.12, there is no visible representation of the disabled items.

Examples

This example sets the selection and changes the enabled state of an **option-pane**:

```
(setq option-pane (capi:contain
                   (make-instance 'capi:option-pane
                                  :items '(1 2 3 4 5)
                                  :selected-item 3)))

(capi:apply-in-pane-process
 option-pane #'(setf capi:choice-selected-item)
 5 option-pane)

(capi:apply-in-pane-process
 option-pane #'(setf capi:option-pane-enabled)
 nil option-pane)

(capi:apply-in-pane-process
```

```
option-pane #'(setf capi:option-pane-enabled)
t option-pane)
```

This example illustrates the use of *visible-items-count* (Windows only):

```
(capi:contain
 (make-instance 'capi:option-pane
                :items
                (loop for i below 20 collect i)
                :visible-items-count 6))
```

These are further examples:

```
(example-edit-file "capi/choice/option-pane")
```

```
(example-edit-file "capi/choice/option-pane-with-images")
```

There are further examples in **20 Self-contained examples**.

See also

`text-input-choice`
3.1.4.1 Controlling Mnemonics
5 Choices - panes with items
9.7.1 Toolbar items other than buttons with images

output-pane *Class*

Summary

An output pane is a pane whose display and input behavior can be controlled by the programmer.

Package

`capi`

Superclasses

`titled-object`
`simple-pane`
`graphics-port-mixin`

Subclasses

`pinboard-layout`
`editor-pane`

Initargs

`:display-callback`	A function called to redisplay the pane.
`:drawing-mode`	A keyword controlling quality of drawing, especially anti-aliasing of text.
`:graphics-options`	A platform-specific plist of options controlling how graphics are drawn.
`:draw-with-buffer`	A boolean controlling whether output is buffered, on Microsoft Windows and Motif.

:input-model A list of input specifications, otherwise known as a command table.

:scroll-callback A function called when the pane is scrolled, or **nil**. The default is **nil**.

:coordinate-origin Either **:scrolled**, **:fixed** or **:fixed-graphics**.

:focus-callback A function called when the pane gets or loses the input focus, or **nil**. The default is **nil**.

:resize-callback A function called when the pane is resized, or **nil**. The default is **nil**.

:create-callback A function called just after the pane is created.

:destroy-callback A function called just before the pane is destroyed.

:use-native-input-method

Controls whether to use native input method to interpret keyboard input. Currently this has an effect only on GTK+.

:composition-callback

This is called for various events related to composition, which here means composing input characters into other characters by an input method.

Accessors

output-pane-display-callback
output-pane-focus-callback
output-pane-resize-callback
output-pane-scroll-callback
output-pane-create-callback
output-pane-destroy-callback
output-pane-composition-callback
output-pane-input-model

Readers

output-pane-graphics-options
output-pane-coordinate-origin

Description

The class **output-pane** is a subclass of **gp:graphics-port-mixin** which means that it supports the graphics ports drawing operations such as **draw-image**, **draw-string** and **draw-path**.

When the CAPI needs to redisplay a region of the output pane, the *display-callback* gets called with the **output-pane** and the *x*, *y*, *width* and *height* of the region that needs redrawing. The *display-callback* should then use Graphics Ports functions to redisplay that area. To force an area to be re-displayed, use the function **invalidate-rectangle**.

Note: if you need to temporarily prevent the *display-callback* from running, for example because it is slow, then use the Cached Display interface so that the pane still redraws. See **output-pane-cache-display** for the details.

drawing-mode should be either **:compatible** which causes drawing to be the same as in LispWorks 6.0, or **:quality** which causes all the drawing to be transformed properly, and allows control over anti-aliasing on Microsoft Windows and GTK+. The default value of *drawing-mode* is **:quality**.

For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

graphics-options is currently only used by the macOS Cocoa implementation. The single option defined is **:text-rendering**, with allowed values:

:glyph Draw glyphs directly using Core Graphics. This only draws characters with glyphs in the chosen font.

:atsui Draw using ATSUI APIs where possible.This is slower but can handle more characters.

When *draw-with-buffer* is true, display of the **output-pane** (that is drawing the background and calling the *display-callback*) is done by first drawing to a pixmap buffer, and then drawing from that buffer. This is useful to avoid flickering if the display is complex. The default value of *draw-with-buffer* is **nil**.

The *input-model* provides a means to get callbacks on m ouse and keyboard gestures. An *input-model* is a list of mappings from gesture to callback, where each mapping is a list:

(*gesture callback* **.** *extra-callback-args***)**

gesture specifies the type of gesture, which can be Gesture Spec (representing keyboard input), character, mouse button (including multiple clicks made in quick succession), modifier change, key, command or cursor motion. On Microsoft Windows and Cocoa *gesture* can also specify multi-touch gestures that come from trackpad or touchscreen devices, including zoom, rotate, pan and more.

gesture can match specific input such as uppercase **A** with the **Control** key pressed, or a general class of input such as any character.

input-model can be set before the pane is displayed, but changes after that are ignored. **cl:initialize-instance** is the natural place for subclasses to modify the existing *input-model*, using the **output-pane** accessor **output-pane-input-model**. Note that since the mappings are processed in order, prepending to an existing *input-model* overrides it when there are clashes, while appending affects only gestures for which the original *input-model* did not have a match.

For all the details of *input-model* syntax and the precedence and interpretation of the various gesture types, see **12.2.1 Detailed description of the input model**.

When *coordinate-origin* is **:scrolled**, which is the default, then the CAPI is responsible for scrolling over the scroll range, and the origin for all the coordinates in callbacks and drawing is scrolling when the user scrolls the pane. This is known as ordinary scrolling, and is what you normally use.

When *coordinate-origin* is **:fixed**, then the user code is responsible for handling scrolling inside the *scroll-callback* of the **output-pane**, and the origin for all coordinates is fixed relative to the top left of the visible area.

When *coordinate-origin* is **:fixed-graphics**, the behavior is like **:fixed**, except that the origin for all CAPI callbacks and function is scrolled (like the ordinary case). Note that in this case, the CAPI coordinates do not match the coordinates used when drawing.

Programming with *coordinate-origin* **:fixed** or **:fixed-graphics** is more complex, but is also much more flexible. See **12.4 output-pane scrolling** for full details.

When the output pane is scrolled, the CAPI calls the *scroll-callback* if this is non-nil. The arguments of the scroll callback are the **output-pane**, the direction (**:vertical**, **:horizontal** or **:pan**), the scroll operation (**:move**, **:drag**, **:step** or **:page**), the amount of scrolling (an integer), and a keyword argument **:interactive**. This has value **t** if the scroll was invoked interactively, and value **nil** if the scroll was programmatic, such as via the function **scroll**. In the macOS Cocoa implementation the direction is always **:pan**. See the following CAPI example files:

```
(example-edit-file "capi/output-panes/scrolling-without-bar.lisp")
(example-edit-file "capi/graphics/scrolling-test.lisp")
```

focus-callback, if non-nil, is a function of two arguments. The first argument is the **output-pane** itself, and the second is a boolean. When the **output-pane** gets the focus, *focus-callback* is called with second argument **t**, and when the **output-pane** loses the focus, *focus-callback* is called with second argument **nil**.

resize-callback, if non-nil, is a function of five arguments called when the **output-pane** is resized. The first argument is the **output-pane** itself, and the rest are its new geometry: *x*, *y*, *width* and *height*.

create-callback, if non-nil, is a function of one argument which is called just after the pane is created (but before it becomes visible). The argument is the pane itself. This function can perform initialization such as loading images.

destroy-callback, if non-nil, is a function of one argument which is called just before the pane is destroyed, for example when the window is closed or the pane is removed from its layout. The argument is the pane itself. This function can perform cleanup operations (though note that images associated with the pane are automatically freed).

use-native-input-method should be **`nil`**, **`t`** or **`:default`**. If *use-native-input-method* is not supplied, or is **`:default`**, the default is used, which is controlled by **`set-default-use-native-input-method`**. The default setting is always to use native input methods.

composition-callback is a function with signature:

`composition-callback` *pane what*

where *pane* is the output pane and *what* can be one of:

`:start` The composition operation is starting.

`:end` The composition ends.

A list A plist describing the "preedit" string, which is a string containing the partial input that should be displayed while the composition is ongoing. These calls with a plist occur only when the underlying system does not display the partial input itself. Currently on Microsoft Windows the system always displays the preedit string itself, so these calls occur only on GTK+ and Cocoa.

During composition there will be repeated calls with a list, in general each time that the preedit string changes. Each call is a complete description of what needs to be displayed. The data from previous calls should be ignored.

The keys that can appear in the plist are currently:

`:string-face-lists` The value is a list where each element is itself a list, where the first element is a string and the second a plist describing a face (a face plist). The strings are the strings that need to be displayed, and the face plist describing the face that the underlying GUI thinks that each string needs to be displayed. The face plist may contain any of the following keywords: **`:foreground`**, **`:background`**, **`:font`**, **`:bold-p`**, **`:italic-p`**, **`:underline-p`**. The argument *string-face-lists* may be **`nil`**, which means display nothing.

`:cursor` The argument is an integer describing where the "cursor" should be displayed. The index is into the string that is concatenation of the strings in *string-face-lists*.

`:selected-range` If present, the value specifies the selected range as a cons of start and length in characters. The start is an index into the string that is a concatenation of the strings in the *string-face-lists*.

`:selection-needs-face`

A boolean specifying whether the *selected-range* should have a different face to the unselected range.

The editor uses the **`:start`** call to position the composition window at the cursor by using **`set-composition-placement`** and the calls with a list to display the partial composition string.

Notes

1. A composition session is initiated and managed by the underlying windowing system (not CAPI) when it is set to use input method which needs to compose characters from several keyboard gestures (mostly input methods for east Asian languages). Keyboard gestures that are used by the composition session are not visible to the application, but some keyboard gestures, typically gestures with modifiers, may be passed through.

2. When the user commits the composition session, the user callbacks from the *input-model* are called on each character in the resulting string (as if the user typed each of these characters). The call to *composition-callback* with **`:start`** should typically use **`set-composition-placement`** to tell the system where the interaction should happen. The calls to *composition-callback* with a list do not always happen, the underlying system may do it all itself.

3. You can stop an ongoing composition session by calling **`output-pane-stop-composition`**. That is useful for gestures like mouse clicks that may change the interaction such that it does not make sense to continue the composition.

4. *draw-with-buffer* is typically useful for a **`pinboard-layout`** with large number of pinboard objects, or any other feature that may cause it to flicker.

5. The GTK+ and Cocoa libraries always buffer, so *draw-with-buffer* is ignored on these platforms.

6. In GTK+ versions before 2.12 the **`:start`** and **`:end`** calls are not reliable.

Compatibility note

In LispWorks 7.0 and earlier versions, the initarg **`:pane-can-scroll`** was used instead of **`:coordinate-origin`**. **`:pane-can-scroll`** can still be used, but it is deprecated. **`:pane-can-scroll nil`** is the same as **`:coordinate-origin :scrolled`**. **`:pane-can-scroll t`** is the same as **`:coordinate-origin :fixed-graphics`**. There was no documented equivalent to **`:coordinate-origin :fixed`**.

Examples

Firstly, here is an example that draws a circle in an output pane.

```
(defun display-circle (self x y width height)
  (declare (ignore x y width height))
  (gp:draw-circle self 200 200 200 :filled t))

(capi:contain (make-instance
               'capi:output-pane
               :display-callback 'display-circle)
              :best-width 200 :best-height 200)
```

Here is an example that shows how to use a button gesture.

```
(defun test-callback (self x y)
  (capi:display-message
   "Pressed button 1 at (~S,~S) in ~S" x y self))

(capi:contain
 (make-instance
  'capi:output-pane
  :title "Press button 1:"
  :input-model `(((:button-1 :press)
                  test-callback)))
 :best-width 200 :best-height 200)
```

This example illustrates Gesture Spec mappings.

```
(defun draw-input (self x y gspec)
  (let ((data (sys:gesture-spec-data gspec))
        (mods (sys:gesture-spec-modifiers gspec)))
    (gp:draw-string
     self
     (with-output-to-string (ss)
       (sys:print-pretty-gesture-spec
        gspec ss :force-shift-for-upcase nil))
     x y)))
```

```
(capi:contain
 (make-instance
  'capi:output-pane
  :title "Press keys in the pane..."
  :input-model '((:gesture-spec
                  draw-input)))
 :best-width 200 :best-height 200)

(capi:contain
 (make-instance
  'capi:output-pane
  :title "Press Control-a in the pane..."
  :input-model '(((:gesture-spec "Control-a")
                  draw-input)))
 :best-width 200 :best-height 200)
```

Here is a simple example that draws the character typed at the cursor point.

```
(defun draw-character (self x y character)
  (gp:draw-character self character x y))

(capi:contain
 (make-instance
  'capi:output-pane
  :title "Press keys in the pane..."
  :input-model '((:character draw-character)))
 :best-width 200 :best-height 200)
```

This example shows how to use the motion gesture.

```
(defun draw-red-blob (self x y)
  (gp:draw-circle self x y 3
                  :filled t
                  :foreground :red))

(capi:contain
 (make-instance
  'capi:output-pane
  :title "Drag button-1 across this pane."
  :input-model '(((:button-1 :motion)
                  gp:draw-point)
                 ((:button-1 :motion :control)
                  draw-red-blob)))
 :best-width 200 :best-height 200)
```

This example illustrates the use of *focus-callback*:

```
(capi:contain
 (make-instance
  'capi:output-pane
  :focus-callback
  #'(lambda (x y)
      (format t
              "Pane ~a ~:[lost~;got~] the focus~%"
              x y))))
```

This example illustrates the use of *graphics-options* to specify ATSUI drawing on Cocoa:

```
(defvar *string*
  (coerce (loop for i from 0 below 60
                collect (code-char (* 5 i)))
```

```
              'text-string))

    (capi:contain
     (make-instance 'capi:output-pane
                    :visible-min-width 400
                    :visible-max-height 50
                    :display-callback
                    #'(lambda (pane x y w h)
                        (gp:draw-string pane
                                        *string*
                                        10 10))
                    :graphics-options
                    '(:text-rendering :atsui)))
```

This example illustrates some effects of *drawing-mode*:

```
(example-edit-file "capi/graphics/catherine-wheel")
```

This example shows how to draw a rectangle indicating selection of objects in response to mouse movement:

```
(example-edit-file "capi/graphics/highlight-rectangle")
```

This example illustrate drawing the results of dynamic computation:

```
(example-edit-file "capi/graphics/plot-offline")
```

There are further examples here:

```
(example-edit-file "capi/output-panes/")
```

See also **20 Self-contained examples**.

See also

define-command
pane-modifiers-state
output-pane-resize
output-pane-stop-composition
pinboard-object
scroll
set-default-use-native-input-method
set-composition-placement
system:gesture-spec
3.12 Tooltips
7 Programming with CAPI Windows
8.12 Popup menus for panes
12 Creating Panes with Your Own Drawing and Input
13 Drawing - Graphics Ports
12.4 output-pane scrolling
16 Printing from the CAPI—the Hardcopy API
17 Drag and Drop

output-pane-cached-display-user-info *Accessor*

Summary

Gets and sets the *user-info* in the current cached display of an output pane.

Package

`capi`

Signature

`output-pane-cached-display-user-info` *pane* **`=>`** *user-info*

`(setf output-pane-cached-display-user-info)` *user-info pane* **`=>`** *user-info*

Arguments

pane⇓	An **`output-pane`**.
user-info⇓	A Lisp object.

Values

user-info⇓	A Lisp object.

Description

The accessor **`output-pane-cached-display-user-info`** gets and sets the *user-info* in the current cached display of the output pane *pane*.

If *pane* does not have a cached display, the getter returns **`nil`** and the setter has no effect (but returns the new *user-info* as per normal Common Lisp conventions).

A value that is set by the setter will be returned by the getter until the cached display is freed by a call to **`output-pane-free-cached-display`**, either explicitly or implicitly. Note that this means that calls to **`start-drawing-with-cached-display`** and **`output-pane-cache-display`** also reset the *user-info*.

See also

`output-pane-free-cached-display`
`start-drawing-with-cached-display`
12.5 Transient display on output-pane and subclasses

output-pane-cache-display *Function*

Summary

Caches the display of an output pane, ready for later drawing.

Package

capi

Signature

output-pane-cache-display *output-pane* **&optional** *from-display-p*

Arguments

output-pane⇓	An **output-pane**.
from-display-p⇓	A generalized boolean.

Description

The function **output-pane-cache-display** caches the display of the output-pane *output-pane*, that is what it currently shows. The result can be used later by **output-pane-draw-from-cached-display**.

When *from-display-p* is false the cached display is created by a "dummy" call to the *display-callback* of *output-pane*. If *from-display-p* is true the cached display is created by copying whatever is currently showing on the screen. Note that any obscured part of the pane will not be copied in this case. The default value of *from-display-p* is false.

Before caching the display, **output-pane-cache-display** performs an implicit call to **output-pane-free-cached-display**, which undoes the effect of all previous Cached Display interface calls.

Notes

1. Caching the display is useful when you want to avoid calls to the *display-callback* during some period, which may be because it is slow or perhaps some other reason.
2. The Cached Display interface functions do not affect the *display-callback* and it is your responsibility to prevent the *display-callback* being called. See **output-pane-draw-from-cached-display** for more information.

See also

output-pane
output-pane-draw-from-cached-display
output-pane-free-cached-display
start-drawing-with-cached-display
12.5 Transient display on output-pane and subclasses

output-pane-draw-from-cached-display *Function*

Summary

Draws from the cached display of an output pane.

Package

capi

Signature

`output-pane-draw-from-cached-display` *pane x y width height*

Arguments

pane⇓ An **`output-pane`**.

x⇓, *y*⇓, *width*⇓, *height*⇓

Real numbers.

Description

The function **`output-pane-draw-from-cached-display`** copies into the output pane *pane* from the last cached display in the region specified by *x*, *y*, *width* and *height*.

Notes

The Cached Display interface functions do not affect the *display-callback* of *pane*. It is your responsibility to prevent the *display-callback* being called, and instead use **`output-pane-draw-from-cached-display`**. One way of achieving this is to have a *display-callback* that does:

```
(if (drawing-from-cached-display-p pane)
    (progn
      (output-pane-draw-from-cached-display
       pane x y width height)
      (draw-some-temporary-stuff pane))
  (real-display-callback pane x y width height))
```

Another way is to replace the *display-callback* for a while.

See also **`start-drawing-with-cached-display`**, which replaces the *display-callback* too.

See also

`output-pane-cache-display`
`output-pane-free-cached-display`
`start-drawing-with-cached-display`
12.5 Transient display on output-pane and subclasses

output-pane-free-cached-display *Function*

Summary

Frees the cached display in an output pane.

Package

`capi`

Signature

`output-pane-free-cached-display` *pane* **`=>`** *user-info*

Arguments

pane⇓ An **output-pane**.

Values

user-info⇓ A Lisp object.

Description

The function **output-pane-free-cached-display** frees the last cached display in *pane*. This is useful because the cached display can be large in memory.

output-pane-free-cached-display returns the *user-info* that is associated with the cached display. Such *user-info* can be set either by **(setf output-pane-cached-display-user-info)** or by passing *user-info* to **start-drawing-with-cached-display**.

Notes

1. **output-pane-free-cached-display** also undoes any effect of **start-drawing-with-cached-display**.
2. The Cached Display interface functions do not affect the *display-callback* and it is your responsibility to prevent the *display-callback* being called. See **output-pane-draw-from-cached-display** for more information.

Examples

This file illustrates the use of **output-pane-free-cached-display** in a drag operation:

```
(example-edit-file "capi/output-panes/cached-display")
```

See also

output-pane-cache-display
start-drawing-with-cached-display
12.5 Transient display on output-pane and subclasses

output-pane-resize *Generic Function*

Summary

Called when an **output-pane** is resized.

Package

capi

Signature

output-pane-resize *output-pane x y width height*

Method signatures

output-pane-resize (*output-pane* **output-pane) (***x* **t) (***y* **t) (***width* **t) (***height* **t)**

Arguments

output-pane⇓ An **output-pane**.

x⇓, *y*⇓, *width*⇓, *height*⇓

Non-negative integers.

Description

The generic function **output-pane-resize** is called when the **output-pane** *output-pane* is resized. *width* and *height* specify the new width and height. *x* and *y* specify the position, but are not reliable and should not be used.

output-pane-resize should not called by the user.

The primary method specialized on **output-pane** sets up internal slots and calls the *resize-callback*.

Notes

1. Normally you respond to resizing by specifying the *resize-callback* with the **:resize-callback** initarg. It is useful to define your own **output-pane-resize** method only when you define your own subclass of **output-pane** which needs to do something when resizing, and you want to allow different *resize-callback*s for individual instances of this class.
2. **output-pane-resize** should not draw anything. Newly-exposed areas are automatically displayed by a later call to the *display-callback*. If areas that are already exposed need redrawing, **output-pane-resize** should call **invalidate-rectangle** to mark these areas for the *display-callback*.

See also

output-pane
invalidate-rectangle

output-pane-stop-composition *Function*

Summary

Stops the ongoing composition.

Package

capi

Signature

output-pane-stop-composition *output-pane* **&key** *process-p* *x* *y* **=>** *result*

Arguments

output-pane⇓ An **output-pane**.

process-p⇓ A generalized boolean.

x⇓, *y*⇓ An integer or **nil**.

Values

result A string or **nil**.

Description

The function **output-pane-stop-composition** stops the ongoing composition session if there is any, returning the currently composed string.

If *process-p* is true and there is a composition, the current composition string is processed as if the user committed it. That is, for each character, the user callbacks from the input model are invoked as if it was typed by the user. The default value of *process-p* is **nil**.

x and *y* provide coordinates for the callbacks. If either of them is **nil**, the current pointer position is used. When *process-p* is **nil**, *x* and *y* are ignored.

output-pane-stop-composition returns the current composition string, if any, or **nil**.

Notes

1. A composition session is initiated and managed by the underlying windowing system (not CAPI) when it is set to use an input method which needs compositioning (mostly input methods for east Asian languages). You can tell when it happens by using **:composition-callback** in *output-pane*.
2. Calling **output-pane-stop-composition** when there is no composition session has no effect.
3. You will typically need to use **output-pane-stop-composition** when a gesture that is not processed by the input method (for example a mouse click) changes the interaction such that it does not make sense to continue the composition.

See also

output-pane

over-pinboard-object-p *Generic Function*

Summary

Tests whether a point lies within the boundary of a pinboard object.

Package

capi

Signature

over-pinboard-object-p *pinboard-object x y*

Arguments

pinboard-object⇓ A **pinboard-object**.

x⇓, *y*⇓ Reals.

Description

The generic function **over-pinboard-object-p** returns non-nil if the coordinates specified by *x* and *y* are within the boundary of *pinboard-object*. To find the actual object at this position, use **pinboard-object-at-position**.

The default method returns **t** if *x* and *y* are within the bounding area of the pinboard object. A method is supplied for **line-pinboard-object** and you may add methods for your own **pinboard-object** subclasses.

See also

pinboard-object-at-position
pinboard-object-overlap-p
pinboard-object
pinboard-layout

page-setup-dialog *Function*

Summary

Displays the page setup dialog for a given printer.

Package

capi

Signature

page-setup-dialog &key *screen owner printer continuation*

Arguments

screen⇓	A **screen** or **nil**.
owner⇓	A pane or **nil**.
printer⇓	A printer or **nil**.
continuation⇓	A function or **nil**.

Description

The function **page-setup-dialog** displays the page setup dialog for *printer*. If *printer* is not specified, the dialog for the current printer is displayed.

The CAPI screen on which to display the dialog is given by *screen*, which is the current screen by default.

owner specifies an owner window for the dialog. See **10 Dialogs: Prompting for Input** for details.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts one argument. *continuation* is called with the values that would normally be returned by **page-setup-dialog**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **display-dialog** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

Examples

```
(example-edit-file "capi/printing/simple-print-port")
```

See also

current-printer
16 Printing from the CAPI—the Hardcopy API

pane-adjusted-offset *Generic Function*

Summary

Calculates the offset required to place a pane correctly in a layout.

Package

capi

Signature

pane-adjusted-offset *pane adjust available-size actual-size* **&key &allow-other-keys =>** *offset*

Arguments

pane⇓	A pane.
adjust⇓	A keyword or a list of keyword and an integer.
available-size⇓	An integer.
actual-size⇓	An integer.

Values

offset	An integer.

Description

The generic function **pane-adjusted-offset** calculates the offset required by *adjust* so that the pane *pane* of size *actual-size* pixels is placed correctly within *available-size* pixels in its parent layout. It is called by all of the layouts that inherit from **x-y-adjustable-layout** to interpret the values of *x-adjust* and *y-adjust*.

Typically, *adjust* will be a keyword or a list of the form **(***keyword n***)** where *n* is an integer. These values of *adjust* are interpreted as by **pane-adjusted-position**.

However, new methods can accept alternative values for *adjust* where required and can also add extra keywords. For example, **grid-layout** allows *adjust* to be a list of adjust values, and then passes the offset into this list as an additional keyword.

Notes

1. **pane-adjusted-offset** is deprecated.
2. Only a keyword value for *adjust* should be supplied when *pane* is a **column-layout** or **row-layout**.

Examples

```
(setq button-panel (make-instance 'capi:button-panel
```

```
                                          :items '(1 2 3)))

(capi:pane-adjusted-offset button-panel
                           :center 200 100)

(capi:pane-adjusted-offset button-panel
                           :left 200 100)

(capi:pane-adjusted-offset button-panel
                           :right 200 100)
```

See also

layout
x-y-adjustable-layout

pane-adjusted-position *Generic Function*

Summary

Calculates how to place a pane correctly within a layout, given a minimum and maximum position.

Package

capi

Signature

pane-adjusted-position *pane adjust min-position max-position* **&key &allow-other-keys**

Arguments

pane⇓	A pane.
adjust⇓	A keyword or a list of keyword and an integer.
min-position⇓	An integer.
max-position⇓	An integer.

Description

The generic function **pane-adjusted-position** calculates the position required by *adjust* so that the pane *pane* is placed correctly within the available space in its parent layout, between *min-position* and *max-position*. It is a complementary function to **pane-adjusted-offset**, and the default method actually calls **pane-adjusted-offset** with the gap between the two positions, and then adds on the minimum position to get the new position.

The default method accepts the following values for *adjust*.

:top	Place *pane* at the top of the region.
:bottom	Place *pane* at the bottom of the region.
:left	Place *pane* at the left of the region.

:right	Place *pane* at the right of the region.
:center	Place *pane* in the center of the region.
(:top *n***)**	Place the top of *pane n* pixels below the top of the region.
(:bottom *n***)**	Place the bottom of *pane n* pixels above the bottom of the region.
(:left *n***)**	Place the left of *pane n* pixels after the left of the region.
(:right *n***)**	Place the right of *pane n* pixels before the right of the region.
(:center *n***)**	Place the center of *pane n* pixels below the center of the region.

However, new methods can accept alternative values for *adjust* where required and can also add extra keywords. For example, **grid-layout** allows *adjust* to be a list of adjust values, and then passes the offset into this list as an additional keyword. It is preferable to add new methods to **pane-adjusted-offset** as these changes will be seen by the default method of **pane-adjusted-position**.

Notes

pane-adjusted-position is deprecated.

Examples

```
(setq button-panel (make-instance 'capi:button-panel
                                   :items '(1 2 3)))


(capi:pane-adjusted-position button-panel
                             :center 100 200)


(capi:pane-adjusted-position button-panel
                             :right 100 200)


(capi:pane-adjusted-position button-panel
                             :left 100 200)
```

See also

layout
graph-pane
x-y-adjustable-layout

pane-can-restore-display-p *Function*

Summary

The predicate for whether a pane's disabled display can be restored.

Package

capi

Signature

`pane-can-restore-display-p` *pane* **=>** *result*

Arguments

pane⇓ A CAPI pane.

Values

result⇓ A boolean.

Description

The function **`pane-can-restore-display-p`** is the predicate for whether a pane that has its display disabled can be restored by **`pane-restore-display`**.

result is **`t`** if *pane* has its display disabled and this can be restored by **`pane-restore-display`**. Otherwise *result* is **`nil`**.

See also

`pane-restore-display`
18.4 Restoring display while debugging

pane-close-display *Function*

Summary

Closes the X display of a pane.

Package

`capi`

Signature

`pane-close-display` *pane* **=>** *closedp*

Arguments

pane⇓ A CAPI element.

Values

closedp⇓ A boolean.

Description

The function **`pane-close-display`** closes the X display connection on which *pane* is currently displayed. This destroys all the other panes on the same connection.

closedp is true if the connection was closed.

Notes

pane-close-display is deprecated. It has no effect on Microsoft Windows and Cocoa, and may not do anything useful on GTK+ either.

pane-descendant-child-with-focus *Function*

Summary

Finds the child with the input focus.

Package

capi

Signature

pane-descendant-child-with-focus *pane* **=>** *result*

Arguments

pane⇓ A pane or layout.

Values

result A pane or **nil**.

Description

The function **pane-descendant-child-with-focus** attempts to find the pane inside *pane* that currently has the input focus, and returns this pane if successful.

pane-descendant-child-with-focus may return **nil** if it does not find a pane with the focus.

See also

pane-has-focus-p
3.1.5 Focus

pane-got-focus *Generic Function*

Summary

A function called when the focus is set programmatically.

Package

capi

Signature

pane-got-focus *interface pane*

Arguments

interface⇓ The interface of *pane*.

pane⇓ A CAPI element.

Description

The generic function **pane-got-focus** is called just before the focus is set in *pane* by **set-pane-focus**. *interface* is the interface of *pane*.

The supplied primary method does nothing. You may add methods on your own interface classes, which can be useful for example when the focus is set programmatically to a pane which is hidden inside a **tab-layout** or **switchable-layout**. Your method can check for this case and modify the layout as required.

See also

set-pane-focus
3.1.5 Focus

pane-has-focus-p *Generic Function*

Summary

Determines whether a pane has the focus.

Package

capi

Signature

pane-has-focus-p *pane* **=>** *focusp*

Arguments

pane⇓ A CAPI element.

Values

focusp A boolean.

Description

The generic function **pane-has-focus-p** is the predicate for whether *pane* currently has the input focus.

Notes

On Motif, **pane-has-focus-p** cannot be used in menu functions such as the *enabled-function* or *popup-callback* of a menu item. It will always return **`nil`**, because the focus is on the menu button when the user clicks on it.

See also

`accepts-focus-p`
`pane-descendant-child-with-focus`
`set-pane-focus`
3.1.5 Focus

pane-initial-focus *Accessor Generic Function*

Summary

Gets or sets the initial focus pane.

Package

`capi`

Signature

`pane-initial-focus` *pane-with-children* **`=>`** *pane*

`(setf pane-initial-focus)` *pane pane-with-children* **`=>`** *pane*

Arguments

pane-with-children⇓	A pane with children.
pane	A child of *pane-with-children*.

Values

pane	A child of *pane-with-children*.

Description

The accessor generic function **`pane-initial-focus`** get or sets the child of *pane-with-children* that has the input focus when *pane-with-children* is first displayed.

`(setf pane-initial-focus)` may be used to set the initial focus pane, but only before *pane-with-children* has been created. If the setter is called after *pane-with-children* has been created, an error is signalled.

pane-with-children should be a pane with child panes such as a **`layout`**, an **`interface`**, a **`button-panel`** or a **`toolbar`**.

See also

`pane-has-focus-p`
3.1.5 Focus

pane-interface-copy-object
pane-interface-copy-p
pane-interface-cut-object
pane-interface-cut-p
pane-interface-deselect-all
pane-interface-deselect-all-p
pane-interface-paste-object
pane-interface-paste-p
pane-interface-select-all
pane-interface-select-all-p
pane-interface-undo
pane-interface-undo-p

Generic Functions

Summary

Implements "edit/select operations" and the associated predicates for the active pane.

Package

capi

Signatures

pane-interface-copy-object *pane interface* **=>** *object*, *string*, *plist*

pane-interface-copy-p *pane interface* **=>** *boolean*

pane-interface-cut-object *pane interface*

pane-interface-cut-p *pane interface* **=>** *boolean*

pane-interface-deselect-all *pane interface*

pane-interface-deselect-all-p *pane interface* **=>** *boolean*

pane-interface-paste-object *pane interface*

pane-interface-paste-p *pane interface* **=>** *boolean*

pane-interface-select-all *pane interface*

pane-interface-select-all-p *pane interface* **=>** *boolean*

pane-interface-undo *pane interface*

pane-interface-undo-p *pane interface* **=>** *boolean*

Arguments

pane⇓ A pane.

interface⇓ The interface of *pane*.

Values

object A Lisp object.

string A string.

plist A plist.

boolean A generalized boolean.

Description

The active pane "edit/select operations" call these generic functions when the active pane does not specify how to perform the operation. Do not call these directly.

interface is the top level interface of *pane*. The predicate functions (those with names ending with **`-p`**) should return true if the operation can be performed. The other functions should perform the operations.

You can implement your own methods specializing on pane and interface classes.

Notes

1. These generic functions should not display a dialog or do anything that may cause the system to hang. In general this means interacting with anything outside the Lisp image, including files, databases and so on.
2. The three return values of **`pane-interface-copy-object`** are passed to **`set-clipboard`**.

See also

`active-pane-copy`
`item-pane-interface-copy-object`
`set-clipboard`
7.6 Edit actions on the active element

pane-modifiers-state *Function*

Summary

Returns an integer describing which modifiers are currently active.

Package

`capi`

Signature

`pane-modifiers-state` *pane* **`=>`** *gesture-spec-bits*

Arguments

pane⇓ A CAPI pane.

Values

gesture-spec-bits An integer or **nil**.

Description

The function **pane-modifiers-state** returns an integer describing which modifiers are currently pressed. The modifiers are **Control**, **Shift**, **Meta** and **Hyper** (representing **Command** on macOS). It also describes whether Caps Lock is currently on.

pane should be a pane that is displayed on the screen. If it is not displayed, **pane-modifiers-state** returns **nil**.

The result is a **cl:logior** of the following bits:

- **sys:gesture-spec-shift-bit**
- **sys:gesture-spec-control-bit**
- **sys:gesture-spec-meta-bit**
- **sys:gesture-spec-hyper-bit**
- **sys:gesture-spec-caps-lock-bit**

The Caps Lock bit behaves in a special way: it is on when Caps is locked, rather than when the **Caps Lock** key is pressed.

For example, to check if the **Control** modifier is currently pressed call:

```
(logtest (pane-modifiers-state pane)
         sys:gesture-spec-control-bit)
```

Notes

On Cocoa **sys:gesture-spec-hyper-bit** is for **Command**.

output-pane supports responding to modifier changes - see **:modifier-change** in the *input-model*.

sys:gesture-spec-shift-bit and so on are documented in the *LispWorks® User Guide and Reference Manual*.

See also

output-pane
sys:gesture-spec-shift-bit
sys:gesture-spec-control-bit
sys:gesture-spec-meta-bit
sys:gesture-spec-hyper-bit
sys:gesture-spec-caps-lock-bit
18.3 Modifier keys state

pane-popup-menu-items *Generic Function*

Summary

Generates the items for the menu associated with a pane.

Package

capi

Signature

pane-popup-menu-items *pane* *interface* **=>** *items*

Arguments

pane⇓ A pane in interface *interface*.

interface⇓ An **interface**.

Values

items⇓ A list in which each element is a **menu-item**, **menu-component** or **menu**.

Description

The generic function **pane-popup-menu-items** generates the items for the menu associated with the pane *pane* within the interface *interface*. The default method of **make-pane-popup-menu** calls **pane-popup-menu-items** to find the items for the menu. If **pane-popup-menu-items** returns **nil**, then **make-pane-popup-menu** returns **nil**.

To specify items for menus associated with panes in your interfaces, define **pane-popup-menu-items** methods specialized on your interface class.

For most supplied CAPI pane classes, the system method returns **nil**. The exceptions are **editor-pane** and **graph-pane**. To inherit the items from the system method (or other more general method), call **call-next-method**.

Notes

1. **pane-popup-menu-items** is not supported for text panes on Cocoa such as **rich-text-pane**.
2. **pane-popup-menu-items** is intended to allow multiple calls on the same pane, to generate menus in different places (as in the example in **make-pane-popup-menu**). Therefore the **menu-object**s that it returns, and their descendant **menu-object**s, must be constructed each time that **pane-popup-menu-items** is called, so that no two menus share any menu item.
3. The returned *items* may specify the arguments for their callbacks, but it is not required. If they do not specify the arguments, then **make-pane-popup-menu** (by calling **make-menu-for-pane**) sets up the callbacks such that they are called on the pane *pane*.

Examples

The methods below specialized on interface class **edgraph**:

1. Append the items that were returned by the system method in the bottom of the menu for the **editor-pane**, and:
2. Add them as a sub-menu for the menu of the **graph-pane**.

```
(capi:define-interface edgraph ()
  ()
  (:panes
   (e1 capi:editor-pane)
   (g1 capi:graph-pane))
  (:layouts
   (main-layout capi:column-layout '(e1 g1)))
```

```
  (:menu-bar  )
  (:default-initargs
   :visible-min-width 200
   :visible-min-height 300))

(defun my-callback (pane)
  (capi:display-message "Callback on pane ~S." pane))

(defmethod capi:pane-popup-menu-items
           ((self capi:editor-pane) (interface edgraph))
  (list*
   (make-instance 'capi:menu-item
                  :title "Item for My Editor Menu."
                  :selection-callback 'my-callback)
   (call-next-method)))

(defmethod capi:pane-popup-menu-items
           ((self capi:graph-pane) (interface edgraph))
  (list
   (make-instance 'capi:menu-item
                  :title "Item for My Graph Menu."
                  :selection-callback 'my-callback)
   (capi:make-menu-for-pane self (call-next-method)
                            :title "Default Graph Menu")))

(capi:display (make-instance 'edgraph))
```

This is a further example:

```
(example-edit-file "capi/elements/pane-popup-menu-items")
```

See also

make-pane-popup-menu
8.12 Popup menus for panes

pane-restore-display *Function*

Summary

Restores the disabled display of a pane if possible.

Package

capi

Signature

pane-restore-display *pane* **=>** *result*

Arguments

pane⇓ A CAPI pane.

Values

result A boolean.

Description

The function **pane-restore-display** restores the disabled display of the pane *pane* if possible.

If the display of *pane* is disabled and can be restored, the function **pane-restore-display** restores it and returns **t**. Otherwise it returns **nil**.

The display of a pane may be disabled to a "restorable" state by some feature, typically a restart around the display callback. For example, if there is an error inside the *display-callback* of an **output-pane**, a restart is added that removes the display callback. If this restart is used, the **output-pane** is not displayed (its *display-callback* is not called) until it is restored (or the *display-callback* gets set explicitly).

Examples

The Window Browser tool in the LispWorks IDE uses **pane-restore-display** in the **Enable Display** item in its menu.

See also

pane-can-restore-display-p
18.4 Restoring display while debugging

pane-screen-internal-geometry *Function*

Summary

Returns the internal geometry of the monitor in which a pane's interface is displayed.

Package

capi

Signature

pane-screen-internal-geometry *pane* **=>** *x*, *y*, *width*, *height*

Arguments

pane⇓ A CAPI pane.

Values

x⇓ An integer.

y⇓ An integer.

width⇓ A positive integer.

height⇓ A positive integer.

Description

The function **pane-screen-internal-geometry** returns the internal geometry of the "monitor" in which the interface that contains *pane* is displayed. A "monitor" is typically a physical monitor, but can be anything that the underlying GUI system considers a monitor.

pane must be inside an interface that is already displayed. **pane-screen-internal-geometry** returns the internal geometry of the monitor on which this interface is displayed. If the interface spreads across multiple monitors, it returns the geometry for the monitor on which the largest area of the interface is displayed.

The internal geometry of a monitor is a rectangle which excludes "system areas" like taskbars and global menu bars and so on. Examples of these include the Windows taskbar, the macOS menu bar, and the macOS Dock. See **screen-internal-geometry** for information about displaying CAPI windows in system areas.

x, *y*, *width* and *height* specify a screen rectangle. *x* and *y* are offsets from the top-left of the primary monitor.

Notes

On GTK+ the internal geometry is of the workspace in which the interface is displayed. When there are multiple monitors these values may be incorrect. You can check the number of monitors by **screen-monitor-geometries**.

See also

screen-internal-geometry
screen-internal-geometries
virtual-screen-geometry
3.13 Screens
4.3 Support for multiple monitors
11.6 Querying and modifying interface geometry

pane-string *Generic Function*

Summary

Returns the text displayed in an **editor-pane**.

Package

capi

Signature

pane-string *pane* **=>** *text*

Arguments

pane⇓	An **editor-pane**.

Values

text	A string.

Description

The generic function **pane-string** returns as a string the text of the buffer that is currently displayed in the **editor-pane** *pane*.

Notes

`pane-string` is deprecated. Use the accessor **`editor-pane-text`** instead.

See also

`editor-pane`

pane-supports-menus-with-images *Function*

Summary

Tests whether a pane supports menus with images.

Package

`capi`

Signature

`pane-supports-menus-with-images` *pane* **`=>`** *result*

Arguments

pane⇓	A displayed CAPI pane.

Values

result	A boolean.

Description

The function **`pane-supports-menus-with-images`** returns **`t`** if *pane* supports menus with images. This means that the menus display both the images and the text correctly.

See the *image-function* of **`menu`** for details of creating a menu with images.

When **`pane-supports-menus-with-images`** returns **`nil`**, menus can display images, but not together with text at the same item They may also display images with transparency incorrectly.

Whether the pane supports menus with images depends on the library in which it is displayed. Support is currently limited to GTK+ and Cocoa.

See also

`menu`
8 Creating Menus

parse-layout-descriptor *Generic Function*

Summary

Returns the object that layout uses for displaying a child.

Package

capi

Signature

parse-layout-descriptor *child-descriptor interface layout* **=>** *result*

Arguments

child-descriptor⇓	An **element**, a symbol, a geometry object or a string.
interface⇓	An **interface**.
layout⇓	A **layout**.

Values

result	An **element** or a geometry object.

Description

The generic function **parse-layout-descriptor** takes a description of a layout's child, and returns the object that the layout is actually going to use. The returned object is an element (**simple-pane** or **pinboard-object**) or a geometry object (the result of call to the default method of **parse-layout-descriptor**).

layout is the **layout** for which *child-descriptor* is being parsed. *interface* is the interface of *layout*.

parse-layout-descriptor is called by **interpret-description** to parse individual children in a layout.

The default method accepts a *child-descriptor* argument which can be one of:

- An **element**.
- A symbol naming a slot in the interface which contains an **element**.
- A geometry object.
- A string (used to construct a **title-pane** or **item-pinboard-object** with the string as its *text*).

Note that when **parse-layout-descriptor** is passed an element, it does not necessarily return that element. For example, it may wrap it with some layout that adds functionality. It may also return a completely separate element.

You can define your own methods, which may specialize on the interface, the layout if you define your own layout class(es), or the description by using a description of your own defined type.

The element that **parse-layout-descriptor** returns, whether explicitly or indirectly, must not be returned more than once for any layouts that are displayed at the same time.

See also

interpret-description
define-layout
layout
6 Laying Out CAPI Panes

password-pane *Class*

Summary

A pane designed for entering passwords, such that when the password is entered it is not visible on the screen.

Package

capi

Superclasses

text-input-pane

Initargs

:overwrite-character

A **base-char**.

Readers

password-pane-overwrite-character

Description

The class **password-pane** is a pane designed for entering passwords, such that when the password is entered it is not visible on the screen. **password-pane** inherits most of its functionality from **text-input-pane**. It starts with the initial text and caret position specified by the arguments *text* and *caret-position* respectively, and limits the number of characters entered with the *max-characters* argument (which defaults to **nil**, meaning there is no maximum).

The password pane can be enabled and disabled with the **text-input-pane** accessor **text-input-pane-enabled**.

overwrite-character is a **base-char** which is the character to display instead of the real characters. The default value of *overwrite-character* is #*.

Examples

```
(setq password-pane (capi:contain
                     (make-instance
                      'capi:password-pane
                      :callback
                      #'(lambda (password interface)
                                (capi:display-message
                                "Password: ~A"
                                password)))))
```

```
(capi:text-input-pane-text password-pane)
```

```
(setq password-pane
      (capi:contain
       (make-instance 'capi:password-pane
                      :max-characters 5
                      :text "abc"
                      :overwrite-character #\$)))
```

```
(capi:password-pane-overwrite-character password-pane2)
```

See also

editor-pane
text-input-pane

pinboard-layout *Class*

Summary

The class **pinboard-layout** provides two very useful pieces of functionality for displaying CAPI windows. Firstly it is a subclass of **static-layout** and so it allows its children to be positioned anywhere within itself (like a pinboard). Secondly it supports **pinboard-object**s which are rectangular areas within the layout which have size and drawing functionality.

Package

capi

Superclasses

output-pane
static-layout

Subclasses

simple-pinboard-layout

Initargs

:highlight-style A keyword.

Description

When a **pinboard-layout** lays out its children, it positions them at the *x* and *y* specified as hints (using **:x** and **:y**), and sizes them to their minimum size (which can be specified using **:visible-min-width** and **:visible-max-width**). Objects can be moved and resized inside the **pinboard-layout** using **(setf pinboard-pane-position)** and **(setf pinboard-pane-size)**. You can find which object is the top object at a point by using **pinboard-object-at-position**.

By default, the **pinboard-layout** is made sufficiently large to accommodate all of its children, as specified by *fit-size-to-children* in the superclass **static-layout**. Note that this results in the pinboard resizing itself automatically when objects are added, removed, moved or resized. If the layout has scrollbars these are also affected. If you need the sizing capabilities, then use the class **simple-pinboard-layout** which surrounds a single child, and adopts the size constraints of that child.

The pinboard layout handles the display of pinboard objects itself by calculating which objects are visible in the region that needs redrawing, and then by calling the generic function **draw-pinboard-object** on these objects in the order that they are specified in the layout description. This means that if two pinboard objects overlap, the later one in the layout description will be on top of the other one. In other words, the description defines the Z-order for objects of type **pinboard-object**. (See the note below regarding the Z-order for objects of type **simple-pane**.)

The children of the **pinboard-layout** are defined by its *description* (inherited from **layout**). When the contents of the layout need to be manipulated while it is on the screen, it is possible to do this by using **(setf layout-description)**. However, when the change involves only **pinboard-object**s, it is much more efficient to use **manipulate-pinboard** instead. This will also cause less flickering.

Highlighting of the layout's children by **highlight-pinboard-object** is controlled by the value of *highlight-style*, as follows:

:invert	Swaps the foreground and background colors.
:standard	Uses system colors.
:default	Calls **draw-pinboard-object-highlighted**.

The default value of *highlight-style* is **:default**.

record-dependent-object can be used to record objects that need to be cleaned-up when the pinboard layout is destroyed.

Notes

1. The **output-pane** initarg **:drawing-mode** controls quality of drawing in a **pinboard-layout**, including anti-aliasing of any text displayed on Microsoft Windows and GTK+.
2. If redrawing flickers on Microsoft Windows or Motif, perhaps because there are many pinboard objects, you can pass the **output-pane** initarg **:draw-with-buffer t**, which uses a pixmap to buffer the output before drawing it to the screen. See **output-pane** for more information.
3. **pinboard-layout** defines its own default *display-callback* (see **output-pane**), **pinboard-layout-display**. If you want to do additional drawing, see **pinboard-layout-display**.
4. Objects of type **simple-pane** are drawn directly by the windowing system and cannot be clipped relative to **pinboard-object**s, which are drawn by CAPI. Therefore **simple-pane**s always appear on top in a pinboard, and their position in the *description* does not affect the Z-order.

Examples

Here are some examples of the use of pinboard objects with pinboard layouts.

```
(capi:contain
 (make-instance
  'capi:pinboard-layout
  :description
  (list
   (make-instance
    'capi:image-pinboard-object
    :image
    (example-file "capi/graphics/Setup.bmp")
    :x 20 :y 20)))
 :best-width 540 :best-height 415)
```

```
(capi:contain
 (make-instance
  'capi:pinboard-layout
  :description (list
               (make-instance
```

```
                     'capi:item-pinboard-object
                     :text "Hello"
                     :x 40 :y 10)
                    (make-instance
                     'capi:line-pinboard-object
                     :x 10 :y 30
                     :visible-min-width 100)))
                    :best-width 200 :best-height 200)
```

There are further examples here:

```
(example-edit-file "capi/applications/")
```

and here:

```
(example-edit-file "capi/graphics/")
```

This example illustrates use of *draw-with-buffer* **t**:

```
(example-edit-file "capi/graphics/compositing-mode")
```

This example shows how to draw a rectangle as the user moves the mouse to select pinboard objects:

```
(example-edit-file "capi/graphics/highlight-rectangle-pinboard")
```

There are further examples in **20 Self-contained examples**.

See also

12.3 Creating graphical objects
manipulate-pinboard
output-pane
pinboard-object
pinboard-object-at-position
pinboard-pane-position
pinboard-pane-size
record-dependent-object
redraw-pinboard-object
static-layout
1.2.1 CAPI elements
3.12.1 Tooltips for output panes

pinboard-layout-display *Generic Function*

Summary

Draws the children of a **pinboard-layout**, by default.

Package

capi

Signature

pinboard-layout-display *pane x y width height*

Arguments

pane⇓	A **pinboard-layout**.
x⇓, *y*⇓	Real numbers.
width⇓, *height*⇓	Positive real numbers.

Description

The generic function **pinboard-layout-display** is the default *display-callback* of **pinboard-layout** (see **output-pane** for documentation of *display-callback* and a description of *pane*, *x*, *y*, *width* and *height*). It is responsible for the drawing of all the children of the pinboard layout.

If you want to have drawing on a **pinboard-layout** which is not done via the children, you can either supply your own *display-callback* to do the other drawing and call **pinboard-layout-display** (or **draw-pinboard-layout-objects**) to draw the children, or subclass **pinboard-layout** and add methods to **pinboard-layout-display** specialized on your class.

In either case, if any of your drawing is "behind" the children, that is children may overlap it and need to obscure it, you need to do your drawing first and then tell the pane about it by calling **redraw-pinboard-layout** with the region that was redrawn and the optional argument *redisplay* = **nil**.

Compatibility note

In LispWorks 6.1 and earlier versions the default *display-callback* was called **pinboard-pane-display** and was not exported, but apparently some programmers defined methods on it anyway. If you did this, you must change your method to **pinboard-layout-display** for LispWorks 7.0 and later versions.

See also

pinboard-layout
output-pane
redraw-pinboard-layout
draw-pinboard-layout-objects
12 Creating Panes with Your Own Drawing and Input

pinboard-object *Class*

Summary

Provides a rectangular area in a **pinboard-layout** with drawing capabilities.

Package

capi

Superclasses

capi-object

Subclasses

ellipse
item-pinboard-object
image-pinboard-object
line-pinboard-object
drawn-pinboard-object
rectangle

Initargs

:pinboard	The output pane on which the pinboard object is drawn.
:activep	If **t**, the pinboard object is made active.
:graphics-args	A plist of Graphics Ports drawing options.
:automatic-resize	A plist.
:no-highlight	A boolean.
:x	A geometry hint specifying the initial *x* position of the pinboard object in the pinboard.
:y	A geometry hint specifying the initial *y* position of the pinboard object in the pinboard.
:external-min-width	A geometry hint specifying the initial minimum width of the pinboard object in the pinboard.
:external-min-height	A geometry hint specifying the initial minimum height of the pinboard object in the pinboard.
:external-max-width	A geometry hint specifying the initial maximum width of the pinboard object in the pinboard.
:external-max-height	A geometry hint specifying the initial maximum height of the pinboard object in the pinboard.
:visible-min-width	A geometry hint specifying the initial minimum visible width of the pinboard object.
:visible-min-height	A geometry hint specifying the initial minimum visible height of the pinboard object.
:visible-max-width	A geometry hint specifying the initial maximum visible width of the pinboard object.
:visible-max-height	A geometry hint specifying the initial maximum height of the pinboard object.
:internal-min-width	A geometry hint specifying the initial minimum width of the display region.
:internal-min-height	A geometry hint specifying the initial minimum height of the display region.
:internal-max-width	A geometry hint specifying the initial maximum width of the display region.
:internal-max-height	A geometry hint specifying the initial maximum height of the display region.

Accessors

pinboard-object-pinboard
pinboard-object-activep
pinboard-object-graphics-args

Description

The class **pinboard-object** provides a rectangular area in a **pinboard-layout** with drawing and highlighting capabilities. A pinboard object behaves just like a simple pane within layouts, meaning that they can be placed into rows, columns and other layouts, and that they size themselves in the same way. The main distinction is that a pinboard object is a much smaller object than a simple pane as it does not need to create a native window for itself.

Each pinboard object is placed into a pinboard layout (or into a layout itself inside a pinboard layout), and then when the pinboard layout wishes to redisplay a region of itself, it calls the function **draw-pinboard-object** on each of the pinboard objects that are contained in that region (in the order that they are specified as children to the layout).

The *graphics-args* slot allows drawing options to be set. These include the *font*, the *background* and *foreground* colors, and others (see **graphics-state**). The *graphics-args* are used by the built-in **pinboard-object** (all subclasses of **pinboard-object** except **drawn-pinboard-object**) as extra arguments in calls to drawing functions. For example, to create a filled red rectangle object, you can use:

```
(make-instance
 'capi:rectangle
 :filled t :x 100 :y 100
 :visible-min-width 100
 :visible-min-height 100
 :graphics-args '(:foreground :red))
```

The graphics args can be accessed after creation using **pinboard-object-graphics-args**, and it is also possible to modify a single value using **pinboard-object-graphics-arg**.

When *no-highlight* is **t**, CAPI does not call **draw-pinboard-object-highlighted** even when the object is highlighted. Typically, the drawing function you supply (either the method **draw-pinboard-object** or the *display-callback* for **drawn-pinboard-object**) will do the highlight in this case, using **pinboard-object-highlighted-p** to check if they need to.

The geometry hints are interpreted as described for **element**. After creation, you can query the geometry of a **pinboard-object** using the functions **static-layout-child-position** and **static-layout-child-size** and **static-layout-child-geometry**. You can also set the geometry using **cl:setf** with these functions.

By default a **pinboard-object** does not accept the input focus.

There are a number of predefined pinboard objects provided by the CAPI. They are as follows:

ellipse Draws an ellipse.

rectangle Draws a rectangle.

item-pinboard-object Draws a title.

line-pinboard-object Draws a line.

right-angle-line-pinboard-object

Draws a right-angled line.

image-pinboard-object Draws an image.

drawn-pinboard-object Uses a user-defined display function.

The main user of pinboard objects in the CAPI is the graph pane, which uses **item-pinboard-object** and **line-pinboard-object** to display its nodes and edges respectively.

To force a pinboard object to redraw itself call **redraw-pinboard-object**. The redrawing may be cached and displayed at a later date.

Call the generic functions **highlight-pinboard-object** and **unhighlight-pinboard-object** to highlight a pinboard and remove its highlighting. If you want non-standard highlighting, you can implement methods for your subclass of **pinboard-object**.

You can test whether a whether a point or region coincides with a pinboard object by the generic functions **over-pinboard-object-p** and **pinboard-object-overlap-p**. The default methods assume a rectangle based on the geometry, which must always be the enclosing rectangle of the whole pinboard object. Therefore you only need to implement methods if your subclass of **pinboard-object** has a non-rectangular shape.

automatic-resize makes the pinboard object resize automatically. This has an effect only if it is placed inside a **static-layout** (including subclasses like **pinboard-layout**). The effect is that when the **static-layout** is resized then the pinboard object also changes its geometry.

The value of *automatic-resize* defines how the pinboard object's geometry changes. It must be a plist of keywords and values which match the keywords of the function **set-object-automatic-resize** and are interpreted in the same way.

Notes

You can also control automatic resizing of a pinboard object using **set-object-automatic-resize**.

Examples

```
(example-edit-file "capi/graphics/pinboard-test")
```

```
(example-edit-file "capi/graphics/highlight-rectangle-pinboard")
```

```
(example-edit-file "capi/graphics/circled-graph-nodes")
```

There are further examples in **20 Self-contained examples**.

See also

pinboard-layout
draw-pinboard-object
graph-pane
highlight-pinboard-object
over-pinboard-object-p
redraw-pinboard-object
redraw-pinboard-layout
pinboard-object-overlap-p
pinboard-object-graphics-arg
set-object-automatic-resize
static-layout
unhighlight-pinboard-object
6 Laying Out CAPI Panes
12.3 Creating graphical objects

pinboard-object-at-position *Generic Function*

Summary

Returns the uppermost pinboard object containing a specified point.

Package

capi

Signature

pinboard-object-at-position *pinboard x y*

Arguments

pinboard⇓	A **pinboard-layout**.
x⇓	A real.
y⇓	A real.

Description

The generic function **pinboard-object-at-position** returns the uppermost pinboard object in *pinboard* that contains the point specified by *x* and *y*. It determines this by mapping over every pinboard object within the pinboard until it finds one for which the generic function **over-pinboard-object-p** returns **t**.

Examples

```
(setq pinboard
      (capi:contain
       (make-instance
        'capi:pinboard-layout)
       :best-width 300
       :best-height 300))

(capi:apply-in-pane-process
 pinboard
 #'(lambda ()
     (make-instance 'capi:item-pinboard-object
                    :text "Hello world"
                    :x 100 :y 100
                    :parent pinboard)))

(capi:pinboard-object-at-position pinboard 0 0)

(capi:pinboard-object-at-position pinboard 110 110)
```

See also

over-pinboard-object-p
pinboard-object-overlap-p
pinboard-object
pinboard-layout

pinboard-object-graphics-arg *Accessor Generic Function*

Summary

Gets or sets the value of a particular drawing parameter in a **pinboard-object**.

Package

capi

Signature

pinboard-object-graphics-arg *self keyword* **=>** *value*

(setf pinboard-object-graphics-arg) *value self keyword* **=>** *value*

Arguments

self⇓	A **pinboard-object**.
keyword⇓	A keyword denoting a graphics state parameter.
value	The value of the drawing option *keyword* in *self*.

Values

value	The value of the drawing option *keyword* in *self*.

Description

The accessor generic function **pinboard-object-graphics-arg** returns or sets the value of the graphics state parameter *keyword* in *self*.

pinboard-object-graphics-arg accesses the value in the *graphics-args* plist of the **pinboard-object** *self*, and **(setf pinboard-object-graphics-arg)** sets the value in this plist. A call to **(setf pinboard-object-graphics-args)** will overwrite anything set by previous calls to **(setf pinboard-object-graphics-arg)**.

The *graphics-args* are used by built-in subclasses of **pinboard-object**.

See **graphics-state** for details of the drawing parameters.

See also

graphics-state
pinboard-object

pinboard-object-highlighted-p *Function*

Summary

The predicate for whether a **pinboard-object** is in the highlighted state.

Package

capi

Signature

pinboard-object-highlighted-p *pinboard-object* **=>** *result*

Arguments

pinboard-object⇓ A **pinboard-object**.

Values

result A boolean.

Description

The function **pinboard-object-highlighted-p** tests whether *pinboard-object* is in the highlighted state. The state is switched by calls to **highlight-pinboard-object** or **unhighlight-pinboard-object**. In **graph-pane** and **tracking-pinboard-layout**, the state switches automatically, but in other panes it happens only by your calls to **highlight-pinboard-object** or **unhighlight-pinboard-object**.

pinboard-object-highlighted-p is useful when the **draw-pinboard-object** method also does the highlighting, so needs to decide if the object is highlighted or not.

pinboard-object-overlap-p *Generic Function*

Summary

Tests whether a specified region overlaps with the region of a pinboard object.

Package

capi

Signature

pinboard-object-overlap-p *pinboard-object top-left-x top-left-y bottom-right-x bottom-right-y* **=>** *result*

Arguments

pinboard-object⇓ A **pinboard-object**.

top-left-x⇓ A real.

top-left-y⇓ A real.

bottom-right-x⇓ A real.

bottom-right-y⇓ A real.

Values

result A boolean.

Description

The generic function **`pinboard-object-overlap-p`** returns true if the region of the pinboard object *pinboard-object* overlaps with the region specified by *top-left-x*, *top-left-y*, *bottom-right-x* and *bottom-right-y*.

See also

`pinboard-object-at-position`
`over-pinboard-object-p`
`pinboard-object`
`pinboard-layout`

pinboard-pane-position *Accessor*

Summary

Gets and sets the location of an object inside its parent **`pinboard-layout`**. This function is deprecated.

Package

`capi`

Signature

`pinboard-pane-position` *self* **`=>`** *x*, *y*

`setf (pinboard-pane-position` *self***`) (values`** *x y***`) =>`** *x*, *y*

Arguments

self⇓ A **`pinboard-object`** or **`simple-pane`**.

x⇓, *y*⇓ The horizontal and vertical coordinates in the **`pinboard-layout`** parent of *self*.

Values

x⇓, *y*⇓ The horizontal and vertical coordinates in the **`pinboard-layout`** parent of *self*.

Description

The accessor **`pinboard-pane-position`** gets and sets the coordinates (*x* and *y*) of *self* inside its parent **`pinboard-layout`** as multiple values.

Examples

```
(let* ((po (make-instance 'capi:item-pinboard-object
                          :text "5x5" :x 5 :y 5
                          :graphics-args
                          '(:background :red)))
       (pl (capi:contain
            (make-instance 'capi:pinboard-layout
                           :description (list po)
                           :visible-min-width 200
                           :visible-min-height 200))))
  (capi:execute-with-interface
   (capi:element-interface pl)
   #'(lambda (po)
       (dotimes (x 20)
         (mp:wait-processing-events 1)
         (let ((new-x (* (1+ x) 10))
               (new-y (* 5 (+ 2 x))))
           (setf (capi:item-text po)
                 (format nil "~ax~a" new-x new-y))
           (setf (capi:pinboard-pane-position po)
                 (values new-x new-y)))))
   po))
```

Notes

pinboard-pane-position is deprecated, but is retained in this version for backwards compatibility. Please use **static-layout-child-position** instead. This does just the same.

See also

static-layout-child-position

pinboard-pane-size *Accessor*

Summary

Gets and sets the size of an object inside its parent **pinboard-layout**. This function is deprecated.

Package

capi

Signature

pinboard-pane-size *self* **=>** *width***,** *height*

setf (pinboard-pane-size *self***) (values** *width height***) =>** *width***,** *height*

Arguments

self⇓	A **pinboard-object** or a **simple-pane**.
width⇓, *height*⇓	Positive integers.

Values

width⇓, *height*⇓ Positive integers.

Description

The accessor **pinboard-pane-size** gets and sets the dimensions (*width* and *height*) of *self* as multiple values.

Examples

```
(let* ((po (make-instance 'capi:pinboard-object
                          :x 5 :y 5
                          :width 5 :height 5
                          :graphics-args
                          '(:background :red)))
       (pl (capi:contain
            (make-instance 'capi:pinboard-layout
                           :description (list po)
                           :visible-min-width 200
                           :visible-min-height 200))))
  (capi:execute-with-interface
   (capi:element-interface pl)
   #'(lambda(po)
       (dotimes (x 20)
         (mp:wait-processing-events 1)
         (let ((new-x (* (1+ x) 10))
               (new-y (* 5 (+ 2 x))))
           (setf (capi:pinboard-pane-size po)
                 (values new-x new-y)))))
   po))
```

Notes

pinboard-pane-size is deprecated, but is retained in this version for backwards compatibility. Please use **static-layout-child-size** instead. This does just the same.

See also

static-layout-child-size

play-sound *Function*

Summary

Plays a loaded sound on Microsoft Windows and Cocoa.

Package

capi

Signature

play-sound *sound* **&key** *wait*

Arguments

sound⇓ A sound object returned by **load-sound**.

wait⇓ A generalized boolean.

Description

The function **play-sound** plays the loaded sound *sound*.

If *wait* is true then **play-sound** will not return until *sound* has finished playing. That is, it plays the sound synchronously. The default value of *wait* is **nil**.

Notes

1. **:wait t** is only implemented on Microsoft Windows.
2. **play-sound** is not implemented on GTK+ and Motif.

See also

load-sound
stop-sound
18.2 Sounds

popup-confirmer *Function*

Summary

Creates a dialog with predefined implementations of **OK** and **Cancel** buttons and a programmer-specified pane in a layout with the buttons.

Package

capi

Signature

popup-confirmer *pane message* **&rest** *interface-args* **&key** *title title-font value-function exit-function apply-function apply-check apply-button ok-function ok-check ok-button no-button no-function all-button all-function cancel-button help-button help-function buttons print-function callbacks callback-type button-position buttons-uniform-size-p foreground background font modal screen focus owner timeout x y position-relative-to button-container button-font continuation callback-error-handler* **=>** *result***,** *successp*

Arguments

pane⇓ A CAPI pane or interface.

message⇓ A string or **nil**.

interface-args⇓ Initialization arguments for **interface**.

title⇓ A string specifying the title of the dialog window.

title-font⇓ The font used in the title.

value-function⇓ Controls the value returned, and whether a value can be returned.

exit-function⇓ Called on exiting the dialog.

apply-function⇓, *apply-check*⇓, *apply-button*⇓

Define the callback, check function and title an **Apply** button.

ok-function⇓, *ok-check*⇓, *ok-button*⇓

Define the callback, check function and title of an **OK** button.

no-button⇓, *no-function*⇓

Define the title and callback of a **No** button.

all-button⇓, *all-function*⇓

Define the title and callback of an **All** button.

cancel-button⇓ Defines the title of a **Cancel** button.

help-button⇓, *help-function*⇓

Define the title and callback of a **Help** button.

buttons⇓ Defines extra buttons.

print-function⇓ Displays *ok-button*, *no-button*, *cancel-button*, *apply-button* and *all-button* as button titles.

callbacks⇓ Defines callbacks for *buttons*.

callback-type⇓ Specifies the callback-type of *buttons*.

button-position⇓ One of **:bottom**, **:top**, **:left**, **:right**.

buttons-uniform-size-p⇓

Controls relative button sizes.

foreground⇓, *background*⇓

Specify colors.

font⇓ A font or a font description.

modal⇓, *screen*⇓, *focus*⇓, *owner*⇓, *timeout*⇓, *x*⇓, *y*⇓, *position-relative-to*⇓

These are passed to **display-dialog**.

button-container⇓ A layout controlling where the buttons of the dialog appear.

button-font⇓ A font or a font description.

continuation⇓ A function or **nil**.

callback-error-handler⇓

A function designator or **nil**.

Values

result⇓ The result of *value-function*, or *pane*, or **nil**.

successp **nil** if the dialog was cancelled, **t** otherwise.

Description

The function **popup-confirmer** is the quickest way to create new dialogs. It creates a dialog with predefined implementations of buttons such as **OK** and **Cancel** and a programmer-specified pane in a layout with the buttons.

Generally the **Return** key selects the dialog's **OK** button and the **Escape** key selects the **Cancel** button, if there is one.

The argument *value-function* should provide a callback which is passed *pane* and should return the value to return from **popup-confirmer**. If *value-function* is not supplied, then *pane* itself will be returned as *result*. If *value-function* wants to indicate that the dialog cannot return a value currently, then it should return a second value that is non-nil.

ok-check is passed the result returned by *value-function* and should return true if it is acceptable for that value to be returned. These two functions are used by **popup-confirmer** to decide when the **OK** button should be enabled, thus stopping the dialog from returning with invalid data. The **OK** button's state can be updated by a call to **redisplay-interface** on the top-level, so the dialog should call it when the button may enable or disable.

ok-button, *no-button* and *cancel-button* are the text strings for the **OK**, **No** and **Cancel** buttons respectively, or **nil** meaning do not include that button. The **OK** button returns successfully from the dialog (with the result of *value-function*), the **No** button means continue but return **nil**, and the **Cancel** button aborts the dialog. Note that there are clear expectations on the part of users as to the functions of these buttons — check the style guidelines of the platform you are developing for.

apply-button, if passed, specifies the title of an extra button which appears near to the **OK** button. *apply-check* and *apply-function* define its functionality.

all-button, if passed, specifies the title of an extra button which is always enabled and which appears near to the button added by *apply-button* (if that exists) or the **OK** button. *all-function* defines its functionality.

help-button, if passed, specifies the title of a help button which appears to the right of the **Cancel** button. *help-function* defines its functionality.

print-function is called on the various *button* arguments to generate a string to display for each button title.

button-position specifies where to put the buttons. The default is **:bottom**.

buttons-uniform-size-p specifies whether the buttons are all the same size, regardless of the text on them. The default is **t**, but **nil** can be passed to make each button only as wide as its text.

foreground and *background* specify colors to use for the parts of the dialog other than *pane*, including the buttons.

font specifies the font to use for *message*.

button-font specifies the font to use in the buttons.

button-container indicates where the buttons of the dialog appear. It must be a layout which is a descendant of *pane*. The description of this layout is automatically set to the button-panel containing the buttons.

exit-function, *ok-function* and *no-function* are the callbacks that are called when exiting, pressing **OK** and pressing **No** respectively. *exit-function* defaults to **exit-confirmer**, *ok-function* defaults to *exit-function* and *no-function* defaults to a function exiting the dialog with **nil**.

buttons, *callbacks* and *callback-type* are provided as a means of extending the available buttons. The buttons provided by *buttons* will be placed after the buttons generated by **popup-confirmer**, with the functions in *callbacks* being associated with them. Finally *callback-type* will be provided as the callback type for the buttons.

If any of *callbacks* need to access *pane*, you could use **confirmer-pane** together with a *callback-type* that passes the interface.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **popup-confirmer**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **popup-confirmer** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

callback-error-handler, if non-nil, should be a function designator for a function of one argument which is a condition, like the *handler-function* in **cl:handler-bind**. The handler is established (by **cl:handler-bind** with type **cl:error**) around each callback call inside the scope of **popup-confirmer** or **display-dialog**. In recursive calls, only the handler of the innermost call to **popup-confirmer** or **display-dialog** is established.

callback-error-handler can use **current-popup** to find the popup (first argument to the innermost call of **display-dialog** or **popup-confirmer**).

If *callback-error-handler* wants to do a non-local exit, it should either call **abort-callback** to abort the callback but leave the dialog, or **exit-dialog** (or **abort-dialog**) to exit (or abort) the dialog.

title, *title-font*, *foreground*,*background*, *font* and the initargs specified by *interface-args* will be passed to the call to **make-instance** for the interface that will be displayed using **display-dialog**. Thus geometry information, colors, and so on can be passed in here as well. *foreground*, *background* and *font* default to the corresponding values in *pane*.

modal, *screen*, *focus*, *owner*, *timeout*, *x*, *y* and *position-relative-to* will be passed to the call to **display-dialog**.

Notes

1. On Microsoft Windows and Motif, the effect of *callback-error-handler* can be achieved by using **cl:handler-bind** around the call to **display-dialog** or **popup-confirmer** (the handler will also handle errors during raising the dialog, but these are not expected to happen). On Cocoa, using such an error handler does not necessarily work, because the callback may happen in another process. *callback-error-handler* ensures that the callback is in the scope of the handler on all platforms. From the same reason the handler should not rely on the dynamic environment (including catchers and restarts), and needs to use **current-popup** to find its "context" and use **abort-callback**, **exit-dialog** or **abort-dialog** for non-local exit.

2. If the callback itself calls **popup-confirmer** or **display-dialog**, the error handler *callback-error-handler* will stay until the callback returns. Unless the recursive call handles the error, the handler of the outer call may be called to handle it, and needs to be written to deal with this possibility correctly. If the handler inside a recursive call needs to access the popup that was used in the same call that the handler was used, it should close over it, because **current-popup** returns the innermost one.

3. A handler that is established by the callback (by **cl:handler-bind** or **cl:handler-case**) is inside the scope of *callback-error-handler*, and therefore will be called first.

Examples

Here are two simple examples which implement the basic functionality of two CAPI prompters: the first implements a simple **prompt-for-string**, while the second implements **prompt-for-confirmation**.

```
(capi:popup-confirmer
  (make-instance 'capi:text-input-pane
                 :callback
                 'capi:exit-confirmer)
  "Enter some text:"
  :value-function 'capi:text-input-pane-text)


(capi:popup-confirmer nil
  "Yes or no?"
  :callback-type :none
  :ok-button "Yes"
  :no-button "No"
  :cancel-button nil
  :value-function #'(lambda (dummy) t))
```

This example demonstrates the use of **:redisplay-interface** to make the **OK** button enable and disable on each keystroke.

```
(defun pane-integer (pane)
  (ignore-errors (values
```

```
                    (read-from-string
                     (capi:text-input-pane-text
                      pane)))))

(capi:popup-confirmer
  (make-instance 'capi:text-input-pane
                 :callback 'capi:exit-confirmer
                 :change-callback :redisplay-interface)
  "Enter an integer"
  :value-function 'pane-integer
  :ok-check 'integerp)
```

An example illustrating the use of **`:button-container`**:

```
(let* ((bt (make-instance 'capi:simple-layout
                          :title "Button Container"
                          :title-position :left))
       (tip1 (make-instance 'capi:text-input-pane
                            :title "Top"))
       (tip2 (make-instance 'capi:text-input-pane
                            :title "Bottom"))
       (layout (make-instance 'capi:column-layout
                              :description
                              (list tip1
                                    bt
                                    tip2))))
  (capi:popup-confirmer layout nil
                        :title
                        "Dialog using button-container"
                        :button-container bt))
```

An example with all the defined buttons in use:

```
(defun all-buttons-dialog (&optional (num 20))
  (let ((pane
         (make-instance 'capi:list-panel
                        :items
                        (loop for ii from 1
                              to num
                              collect
                              (format nil "~r" ii))
                        :visible-min-width
                        '(character 20))))
    (capi:popup-confirmer
     pane
     "All Buttons"
     :callback-type :none
     :button-position :right
     :cancel-button "Cancel Button"
     :ok-button "OK Button"
     :ok-function #'(lambda (x)
                      (declare (ignorable x))
                      (capi:exit-dialog
                       (capi:choice-selected-item pane)))
     :no-button "No Button"
     :no-function
     #'(lambda ()
         (capi:exit-dialog
          (cons :no
                (capi:choice-selected-item pane))))
     :apply-button "Apply Button"
     :apply-function
     #'(lambda ()
         (capi:display-message
```

```
             "Applying to ~a"
             (capi:choice-selected-item pane)))
        :help-button "Help Button"
        :help-function
        #'(lambda ()
            (capi:display-message
             "~a is ~:[an odd~;an even~] number"
             (capi:choice-selected-item pane)
             (oddp (capi:choice-selection pane))))
        :all-button "All Button"
        :all-function
        #'(lambda()
            (capi:exit-dialog
             (capi:collection-items pane))))))

(all-buttons-dialog)
```

A dialog with arbitrary buttons:

```
(capi:popup-confirmer
 (make-instance 'capi:text-input-pane)
 "Dialog with arbitrary buttons"
 :buttons '(:abc :xyz)
 :callbacks
 (list #'(lambda (data)
           (capi:display-message
            "Button ~A was pressed" data))
       #'(lambda (data)
           (capi:display-message
            "Button with ~A was pressed, exiting with ~S" data data)
           (capi:exit-dialog data)))
 :callback-type :data)
```

This example illustrates the use of *callback-error-handler*:

```
(defun my-error-handler (condition)
  (let ((pane (capi:current-popup)))
    (capi:display-message
     "Error inside dialog: ~a : ~a"
     (capi:capi-object-name pane)
     condition)
    (capi:abort-callback)))

(let*
    ((foo-callback
      (lambda ()
        (let ((md (make-instance
                   'capi:push-button
                   :text "Error inside Callback-Error-Handler"
                   :name "Chicken"
                   :callback-type :data
                   :data "Twisted ankle."
                   :callback 'error)))
          (capi:popup-confirmer
           md nil
           :callback-error-handler 'my-error-handler))))
     (foo (make-instance
           'capi:push-button
           :text
           "Popup confirmer with Callback-Error-Handler"
           :callback-type :none
           :callback foo-callback))
     (bar (make-instance
           'capi:push-button
```

```
              :text "Error without a handler"
              :callback-type :data
              :data "Broken leg."
              :callback 'error)))
     (capi:contain (list foo bar)))
```

See also

abort-dialog
abort-exit-confirmer
confirmer-pane
display-dialog
exit-confirmer
exit-dialog
10 Dialogs: Prompting for Input

popup-menu-button *Class*

Summary

A button with a popup menu.

Package

capi

Superclasses

simple-pane
item

Initargs

:menu	A **menu** or **nil**.
:menu-function	A function designator or **nil**.

Accessors

popup-menu-button-menu
popup-menu-button-menu-function

Description

The class **popup-menu-button** provides a button with a popup menu, which is displayed when the user clicks on the button.

If *menu-function* is non-nil, it should be function of one argument (the pane) and should return a **menu** object. Otherwise, *menu* should be a **menu** object.

popup-menu-button inherits from **item**, so you can supply *text*, *data* and so on.

Notes

Do not use **popup-menu-button** inside toolbars. Use **toolbar-button** instead.

Examples

```
(example-edit-file "capi/elements/popup-menu-button")
```

See also

menu
toolbar-button

popup-menu-force-popdown *Function*

Summary

Cancels a popup menu.

Package

capi

Signature

popup-menu-force-popdown *popup-menu* **=>** *result*

Arguments

popup-menu⇓ A **menu** displayed using **display-popup-menu**.

Values

result⇓ A boolean.

Description

The function **popup-menu-force-popdown** cancels the menu *popup-menu* if it is currently displayed.

popup-menu should be a popup menu, that is a menu that is displayed using **display-popup-menu**. **popup-menu-force-popdown** pops it down, in the same way that pressing **Cancel** would normally do.

popup-menu-force-popdown can be called from any process. In particular, it can be called from a timer without worrying on which process it is actually executed. For examples of using timers in CAPI, see **20.4 Examples using timers to implement "animation"**.

If *popup-menu* is not displayed, **popup-menu-force-popdown** has no effect.

The result is **t** if the menu is displayed when **popup-menu-force-popdown** is called. Otherwise *result* is **nil**.

Notes

popup-menu-force-popdown can be called from any process.

See also

display-popup-menu

menu
8.13 Displaying menus programmatically

ppd-directory *Variable*

Summary

The directory in which LispWorks looks for PPD files.

Package

capi

Initial Value

nil

Description

The variable ***ppd-directory*** specifies where LispWorks looks for PostScript Printer Definition (PPD) files.

This applies only on Motif.

The directory which is the value of ***ppd-directory*** should contain PPD files (files with extension ppd) either directly, or under subdirectories. The PPD files under each subdirectory are grouped together, with the name of the directory as the group name. PPD files in ***ppd-directory*** itself are grouped under the "Other" group.

See also

16.7 Printing on Motif

print-capi-button *Generic Function*

Summary

Generates the text for a button.

Package

capi

Signature

print-capi-button *button* **=>** *text*

Arguments

button⇓ A **button**.

Values

text A string.

Description

The generic function **print-capi-button** is called by CAPI to generate the text for *button*.

You can add methods for your own button classes.

See also

button

print-collection-item *Generic Function*

Summary

Prints an item as a string.

Package

capi

Signature

print-collection-item *item collection*

Arguments

item⇓	An **item** or an Lisp object.
collection⇓	A **collection** or any Lisp object.

Description

The generic function **print-collection-item** prints *item* as a string. It is used when *item* is known to be an item in *collection*.

An **item** in a collection prints using the first of these which returns non-nil: the item's *text*, the item's *print-function*, the collection's *print-function* or the item's *data*. An **item** not known to be in the collection is printed simply using **print-object**.

The method on **(t collection)** uses the collection's *print-function*.

Examples

```
(setq collection (make-instance
                  'capi:collection
                  :items '(1 2 3 4 5)
                  :print-function #'(lambda (x)
                                      (format nil
                                       "<~A:>"
                                       x))))


(capi:print-collection-item 2 collection)
```

In this example we provide our own **print-collection-item** method:

```
(defclass my-tree-view (capi:tree-view) ())

(defmethod capi:print-collection-item ((item capi:item)
                                        (tree my-tree-view))
  (string-capitalize (svref (capi:item-data item) 0)))

(capi:contain
 (make-instance 'my-tree-view
                :roots
                (list (make-instance 'capi:item
                                     :data
                                     (vector "foo")))))
```

See also

get-collection-item
collection

print-dialog *Function*

Summary

Displays a print dialog and returns a printer object.

Package

capi

Signature

print-dialog &key *screen owner first-page last-page print-selection-p print-pages-p print-copies-p continuation* **=>** *printer*

Arguments

screen⇓	A **screen** or **nil**.
owner⇓	A pane or **nil**.
first-page⇓	A positive integer or **nil**.
last-page⇓	A positive integer or **nil**.
print-selection-p⇓	A generalized boolean.
print-pages-p⇓	A generalized boolean.
print-copies-p⇓	A generalized boolean.
continuation⇓	A function or **nil**.

Values

printer	A printer, or **nil**.

Description

The function **`print-dialog`** displays a print dialog and returns a printer object. The printer object returned will print multiple copies if requested by the user.

If *print-pages-p* is **`t`**, the user can select a range of pages to print. This should always be the case unless the application only produces single page output. If *print-pages* is **`t`**, *first-page* and *last-page* can be used to initialize the page range. For example, they could be set to be the first and last pages of the document.

print-copies-p indicates whether the application handles production of multiple copies for drivers that do not support this function. Currently this should be **`nil`** if the application uses Page Sequential printing and **`t`** if the application uses Page on Demand printing.

If *print-selection-p* is **`t`**, the user is given the option of printing the current selection. Only specify this if the application has a notion of selection and selecting printing functionality is provided.

The dialog is displayed on the current screen unless *screen* specifies otherwise.

owner specifies an owner window for the dialog. See **10 Dialogs: Prompting for Input** for details.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts one argument. *continuation* is called with the values that would normally be returned by **`print-dialog`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`print-dialog`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

Note that the printer object itself is opaque but programmatic setting of some printer options is available via the function **`set-printer-options`**.

Examples

```
(example-edit-file "capi/graphics/metafile")
```

```
(example-edit-file "capi/printing/fit-to-page")
```

```
(example-edit-file "capi/printing/multi-page")
```

```
(example-edit-file "capi/printing/page-on-demand")
```

See also

`print-file`
`print-text`
`set-printer-options`
10 Dialogs: Prompting for Input
16 Printing from the CAPI—the Hardcopy API

print-editor-buffer *Function*

Summary

Prints the contents of an editor buffer to the printer.

Package

capi

Signature

print-editor-buffer *buffer* **&key** *start end printer interactive font*

Arguments

buffer⇓	An editor buffer.
start⇓, *end*⇓	Editor points or **nil**.
printer⇓	A printer or **nil**.
interactive⇓	A boolean.
font⇓	A **font** or a **font-description**, or **nil**.

Description

The function **print-editor-buffer** prints the contents of *buffer* to *printer*, which is the current printer by default.

By default the entire editor buffer is printed, but by specifying *start* and *end* to be editor points, a part of the buffer can be printed. See the *Editor User Guide* for information about editor points.

If *interactive* is **t**, the default value, then a printer dialog is displayed.

font is interpreted as described for **print-text**.

See also

print-file
print-text
10 Dialogs: Prompting for Input
16 Printing from the CAPI—the Hardcopy API

printer-configuration-dialog *Function*

Summary

Displays a dialog allowing the user to configure printers.

Package

capi

Signature

printer-configuration-dialog &key *screen owner*

Arguments

screen⇓	A **screen** **nil**.

owner⇓ A pane or **nil**.

Description

The function **printer-configuration-dialog** displays the printer configuration dialog that allows users to add and configure PostScript printers.

This applies only on Motif.

screen specifies a CAPI screen on which to display the dialog. *owner* controls which interface owns the dialog. If it is specified it should be a currently displayed CAPI interface; it defaults to the current top level interface.

The general options that are available are described under **install-postscript-printer**. In addition, printer-specific options (which are defined in the printer PPD file) are available.

The printers that are visible in the dialog are defined by files in the directories in the list ***printer-search-path***.

See also

install-postscript-printer
printer-search-path
16.7 Printing on Motif

printer-metrics *System Class*

Summary

The type of objects containing printer metrics.

Package

capi

Superclasses

t

Description

Instances of the system class **printer-metrics** are returned by **get-printer-metrics**. The readers for the slots of a **printer-metrics** object are described below.

printer-metrics-device-height and **printer-metrics-device-width** respectively return the height and width of the printable page in the internal units used by the printer driver or printing subsystem of the printer. These functions should not be used to determine the aspect ratio of the printable page as some printers have size units that differ in the x and y directions.

printer-metrics-dpi-x and **printer-metrics-dpi-y** return the number of printer device units per inch in the x and y directions respectively. This typically corresponds to the printer resolution, although in some cases this may not be known. For example, a generic PostScript language compatible driver might always return 300dpi, even though it cannot know the resolution of the printer the PostScript file will actually be printed on.

printer-metrics-height and **printer-metrics-width** respectively return the height and width of the printable area in millimeters.

printer-metrics-left-margin and **printer-metrics-top-margin** respectively return the current left margin and current top margin of the printable area in millimeters.

printer-metrics-max-height and **printer-metrics-max-width** respectively return the greatest possible height and width of the printable area in millimeters.

printer-metrics-min-left-margin and **printer-metrics-min-top-margin** respectively return the smallest possible left margin and top margin of the printable area in millimeters.

printer-metrics-paper-height and **printer-metrics-paper-width** respectively return the height and width of the paper selected for this printer in millimeters.

See also

get-printer-metrics
16 Printing from the CAPI—the Hardcopy API

printer-port *Class*

Summary

An object that **with-print-job** uses when a pane is not supplied.

Package

capi

Superclasses

graphics-port-mixin

Description

The class **printer-port** is the class of the object that **with-print-job** binds its *var* argument to when it is not given a pane.

printer-port is a graphics port, which is described in **13 Drawing - Graphics Ports** and **22 GRAPHICS-PORTS Reference Entries**.

Notes

The phrase "printer port" refers to either to an instance of **printer-port** or an instance of **output-pane** when it is used as the pane argument to with-printer-job.

See also

output-pane
with-print-job

printer-port-handle *Function*

Summary

Returns the underlying handle to a printer port.

Package

capi

Signature

printer-port-handle &optional *port* **=>** *handle*

Arguments

port⇓ A printer port.

Values

handle⇓ Platform-dependent.

Description

The function **printer-port-handle** returns a platform-dependent value which represents the underlying handle to the printer port.

On Microsoft Windows, *handle* is the HDC for the printer device.

If *port* is passed it should be the value bound to *var* in **with-print-job**. If *port* is not supplied it defaults to the current printer port (dynamically bound within **with-print-job**).

See also

with-print-job
16 Printing from the CAPI—the Hardcopy API

printer-port-supports-p *Function*

Summary

Detects if the printer port can support a certain feature.

Package

capi

Signature

printer-port-supports-p *feature* **&optional** *port* **=>** *supportedp***,** *validp*

Arguments

feature⇓ A keyword.

port⇓ A printer port.

Values

supportedp⇓ A boolean.

validp⇓ A boolean.

Description

The function **`printer-port-supports-p`** detects if the printer port can support the feature named by *feature*.

If *port* is passed it should be the value bound to *var* in **`with-print-job`**. If *port* is not supplied it defaults to the current printer port (dynamically bound within **`with-print-job`**).

supportedp indicates if the feature is supported.

validp indicates if the feature was recognized.

Currently the only value of *feature* that is recognized is **`:postscript`** and *supportedp* is true if the printer supports PostScript.

See also

`with-print-job`
16 Printing from the CAPI—the Hardcopy API

printer-search-path *Variable*

Summary

Specifies where to look for printer definition files.

Package

`capi`

Initial Value

`("~/.lispworks-printers/" nil)`

Description

The variable **`*printer-search-path*`** specifies where to look for printer definition files.

This applies only on Motif.

The value is a list containing directory pathname designators specifying where to look for printer definition files. The list can also include the value **`nil`**, which is interpreted as the **`printers`** directory in the LispWorks library.

To find known printers the system loads all files in these directories. If there are duplicate printer definitions, the printer in the first directory takes precedence.

The default path is useful when printing from the Common LispWorks IDE, but applications that want to allow users to use printers should set the list appropriately.

The first path in the ***printer-search-path*** list is regarded as the "local" path. New printers are saved in this path. When the user edits a printer that was found in another directory on ***printer-search-path*** and then tries to save it, the system prompts for whether to overwrite the original or save it in the "local" directory.

The printer files can be copied to other directories, on the same machine, and hence to install printers in different directories.

A printer file can be copied to other machines, provided the printer is installed on the other machine and the PPD file is available in the same path.

See also

16.7 Printing on Motif

print-file *Function*

Summary

Prints the contents of a specified file.

Package

capi

Signature

print-file *file* **&key** *printer interactive font*

Arguments

file⇓	A pathname designator.
printer⇓	A printer or **nil**.
interactive⇓	A boolean.
font⇓	A **font** or a **font-description**, or **nil**.

Description

The function **print-file** prints *file* to *printer*, which defaults to the current printer. If *interactive* is **t**, then a print dialog is displayed. This is the default behavior.

font is interpreted as described for **print-text**.

See also

print-editor-buffer
print-text
16 Printing from the CAPI—the Hardcopy API

print-rich-text-pane *Function*

Summary

Prints the contents of a **rich-text-pane**, on Microsoft Windows.

Package

capi

Signature

print-rich-text-pane *pane* **&key** *jobname printer interactive selection* **=>** *result*

Arguments

pane⇓	A **rich-text-pane**.
jobname⇓	A string, or **nil**.
printer⇓	A printer, or **nil**.
interactive⇓	A boolean.
selection⇓	A boolean.

Values

result	A boolean.

Description

The function **print-rich-text-pane** prints the contents in *pane*.

jobname is the name of the print job. The default value is **nil**, meaning that the name "Document" is used.

printer is the printer to use. The default value is **nil**, meaning that the **current-printer** is used.

interactive, if true, specifies that a **print-dialog** is displayed before printing. The default value of *interactive* is **t**.

selection is a boolean specifying what to print. If true, only the current selection is printed. If **nil**, all the contents of *pane* are printed. The default value is **nil**.

Notes

print-rich-text-pane is supported only on Microsoft Windows.

See also

rich-text-pane
16 Printing from the CAPI—the Hardcopy API

print-text *Function*

Summary

Prints plain text to a printer.

Package

capi

Signature

print-text *line-function* **&key** *printer tab-spacing interactive font*

Arguments

line-function⇓	A function.
printer⇓	A printer or **nil**.
tab-spacing⇓	A positive integer or **nil**.
interactive⇓	A boolean.
font⇓	A **font** or a **font-description**, or **nil**.

Description

The function **print-text** prints plain text to a printer specified by *printer*, and defaulting to the current printer.

line-function is called repeatedly with no arguments to enumerate the lines of text. It should return **nil** when the text is exhausted.

tab-spacing, which defaults to 8, specifies the number of spaces printed when a tab character is encountered.

print-text starts a new page when a line consisting of just a formfeed character (ASCII 12) is found in the text.

If *interactive* is **t**, then a print dialog is displayed. This is the default behavior.

font should be a **gp:font** object, or a Font Description object, or a symbol which is a font alias as defined by **define-font-alias**. The printed text is line wrapped on the assumption that the font is fixed width, so be sure to pass a suitable font. The default value of *font* is a Font Description for a fixed pitch font of size 10.

See also

print-editor-buffer
print-file
16 Printing from the CAPI—the Hardcopy API

process-pending-messages *Function*

Summary

Processes all the pending messages in the current process.

Package

`capi`

Signature

`process-pending-messages` *ignored* **`=> nil`**

Arguments

ignored⇓ This argument is ignored.

Description

The function **`process-pending-messages`** processes all the pending messages in the current process, and then returns **`nil`**. It is useful when your code needs to continuously do something, but also needs to respond to user input or other messages.

ignored is ignored.

See also

4.1 The correct thread for CAPI operations

progress-bar *Class*

Summary

A pane that is used to show progress during a lengthy task.

Package

`capi`

Superclasses

`range-pane`
`titled-object`
`simple-pane`

Description

The class **`progress-bar`** is used to display progress during a lengthy task. It has no interactive behavior.

The **`range-pane`** accessors **`(setf range-start)`** and **`(setf range-end)`** are used to specify integers delimiting the range of values the progress bar can display.

The accessor **(setf range-slug-start)** is used to set an integer value for the progress indicator.

Examples

```
(example-edit-file "capi/elements/progress-bar")
```

```
(example-edit-file "capi/elements/progress-bar-from-background-thread")
```

See also

range-pane
titled-object
3.9.4 Slider, Progress bar and Scroll bar

prompt-for-color *Function*

Summary

Presents a dialog box allowing the user to choose a color.

Package

capi

Signature

prompt-for-color *message* **&key** *color* *colors* *owner* **=>** *result***,** *successp*

Arguments

message⇓	A string.
color⇓	A color specification.
colors⇓	A list.
owner⇓	An owner window.

Values

result	A color specification, or **nil**.
successp	A boolean.

Description

The function **prompt-for-color** pops up a dialog box allowing the user to choose a color.

message supplies a title for the dialog on GTK+ and Motif. On Microsoft Windows *message* is ignored.

color provides the default color in the dialog.

colors is a list of custom color specifications that the user can choose from.

owner specifies an owner window for the dialog. See **10 Dialogs: Prompting for Input** for details.

Notes

For a description of color specifications, see **15.1 Color specs**.

See also

10 Dialogs: Prompting for Input

prompt-for-confirmation *Function*

Summary

Displays a dialog box with a message and **Yes** and **No** buttons.

Package

capi

Signature

prompt-for-confirmation *message* **&key** *screen owner cancel-button default-button continuation* **=>** *result*, *successp*

Arguments

message⇓	A string.
screen⇓	A screen.
owner⇓	An owner window.
cancel-button⇓	A boolean.
default-button⇓	A keyword, or **nil**.
continuation⇓	A function or **nil**.

Values

result	A boolean.
successp	A boolean.

Description

The function **prompt-for-confirmation** displays a dialog box containing *message*, with **Yes** and **No** buttons. When either **Yes** or **No** is pressed, it returns two values:

- A boolean indicating whether **Yes** was pressed.
- **t** (for compatibility with other prompt functions).

cancel-button specifies whether a **Cancel** button also appears on the dialog. When **Cancel** is pressed, **abort** is called and the dialog is dismissed. The default value of *cancel-button* is **nil**.

default-button specifies which button has the input focus when the dialog appears (and is thus selected when the user immediately presses **Return**).The value **:ok** means **Yes**, the value **:cancel** means **Cancel**, and any other value means **No**. The default value of *default-button* is **nil**.

screen specifies a CAPI screen on which to display the dialog. *owner* specifies an owner window for the dialog. See **10 Dialogs: Prompting for Input** for details.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **prompt-for-continuation**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **prompt-for-confirmation** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

Examples

```
(capi:prompt-for-confirmation "Continue?")
```

```
(multiple-value-bind (res success)
    (capi:prompt-for-confirmation  "Yes, No or Cancel"
                                   :cancel-button t)
  (if success
      res
    (abort)))
```

See also

confirm-yes-or-no
10 Dialogs: Prompting for Input

prompt-for-directory *Function*

Summary

Displays a dialog prompting the user for a directory.

Package

capi

Signature

prompt-for-directory *message* **&key** *if-does-not-exist pathname file-package-is-directory pane-args popup-args owner continuation use-file-dialog* **=>** *result*, *successp*

Arguments

message⇓ A string.

if-does-not-exist⇓ One of **:ok**, **:prompt** or **:error**.

pathname⇓ A pathname, or **nil**.

file-package-is-directory⇓
A generalized boolean.

pane-args⇓ Arguments to pass to the pane.

popup-args⇓	Arguments to pass to the confirmer.
owner⇓	An owner window.
continuation⇓	A function or **`nil`**.
use-file-dialog⇓	A generalized boolean.

Values

result	A directory pathname, or **`nil`**.
successp⇓	A boolean.

Description

The function **`prompt-for-directory`** prompts the user for a directory pathname using a dialog box. Like all the prompters, **`prompt-for-directory`** returns two values: the directory pathname and a flag indicating success. *successp* will be **`nil`** if the dialog was cancelled, and **`t`** otherwise.

message is shown in the dialog box.

On Windows and Motif, if *if-does-not-exist* is **`:ok`**, a non-existent directory can be chosen. When set to **`:prompt`**, if a non-existent directory is chosen, the user is prompted for whether the directory should be created. When set to **`:error`**, the user cannot choose a non existent directory. The default value of *if-does-not-exist* is **`:prompt`**.

On Cocoa it is never possible to choose a non-existent directory, and the value of *if-does-not-exist* is ignored.

pathname, if non-nil, supplies an initial directory for the dialog. The default value for *pathname* is **`nil`**, and with this value the dialog initializes with the current working directory.

file-package-is-directory is handled as by **`prompt-for-file`**.

owner specifies an owner window for the dialog. See **10 Dialogs: Prompting for Input** for details.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **`prompt-for-directory`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`prompt-for-directory`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

On Windows, when *use-file-dialog* is true (the default) and the **`"shell-objs"`** module has been loaded (not the default), then the directory prompter looks like the standard file prompters. *use-file-dialog* is ignored on other platforms.

The prompt itself is created by passing an appropriate pane to **`popup-confirmer`**. Arguments can be passed to the **`make-instance`** of the pane and the call to **`popup-confirmer`** using *pane-args* and *popup-args* respectively. Currently, the pane used to create the file prompter is internal to the CAPI.

See also

`popup-confirmer`
`prompt-for-file`
10 Dialogs: Prompting for Input

prompt-for-file *Function*

Summary

Displays a dialog prompting the user for a filename.

Package

capi

Signature

prompt-for-file *message* **&key** *pathname ok-check filter filters if-exists if-does-not-exist file-package-is-directory operation owner pane-args popup-args continuation* **=>** *filename*, *successp*, *filter-name*

Arguments

message⇓ A string or **nil**.

pathname⇓ A pathname designator or **nil**.

ok-check⇓ A function or **nil**.

filter⇓ A string or **nil**.

filters⇓ A property list.

if-exists⇓ One of **:ok** or **:prompt**.

if-does-not-exist⇓ One of **:ok**, **:prompt** or **:error**.

file-package-is-directory⇓

A generalized boolean.

operation⇓ One of **:open** or **:save**.

owner⇓ An owner window.

pane-args⇓ Arguments to pass to the pane.

popup-args⇓ Arguments to pass to the confirmer.

continuation⇓ A function or **nil**.

Values

filename⇓ A pathname or **nil**.

successp⇓ A boolean.

filter-name⇓ A string.

Description

The function **prompt-for-file** prompts the user for a file using a dialog box.

message is shown in the dialog box.

pathname, if non-nil, is a pathname designator providing a default filename for the dialog.

ok-check, if non-nil, should be a function which takes a pathname designator argument and returns a true value if the pathname is valid.

filter specifies the initial filter expression. The default value is **`"*.*"`**. An example filter expression with multiple filters is **"*.LISP;*.LSP"**.

filter is used on all platforms. However on Motif, if *filter* contains multiple file types, only the first of these is used.

On Cocoa **`prompt-for-file`** supports the selection of application bundles as files if they match the filter. For example, they will match if the filter expression contains **`*.app`** or **`*.*`**.

filters is a property list of filter names and filter expressions, presenting filters which the user can select in the dialog. If *filter* is not one of the expressions in *filters*, an extra filter called **`"Files"`** is added for this expression.

On Microsoft Windows the default value of *filters* is:

```
("Lisp Source Files" "*.LISP;*.LSP"
 "Lisp Fasls" "*.OFASL"
 "Text Documents" "*.DOC;*.TXT"
 "Image Files" "*.BMP;*.DIB;*.ICO;*.CUR"
 "All Files" "*.*")
```

The **"Lisp Fasls"** extension may vary depending on the implementation.

On Cocoa and GTK+ the default value of *filters* is:

```
("Lisp Source Files" "*.lisp;*.lsp"
 "Text Documents" "*.txt;*.text"
 "All Files" "*.*")
```

filters is ignored on Motif.

When *if-exists* is **`:ok`**, an existing file can be returned. Otherwise the user is prompted about whether the file can be overwritten. The default for *if-exists* is **`:ok`** when *operation* is **`:open`** and **`:prompt`** when operation is **`:save`**.

When *if-does-not-exist* is **`:ok`**, a non-existent file can be chosen. When it is **`:prompt`**, the user is prompted if a non-existent file is chosen. When it is **`:error`**, the user cannot choose a non-existent file. The default for *if-does-not-exist* is **`:prompt`** if *operation* is **`:open`** and **`:ok`** if *operation* is **`:save`**.

operation chooses the style of dialog used, in LispWorks for Windows only. The default value is **`:open`**.

owner, if non-nil, specifies an owner window for the dialog. See **10 Dialogs: Prompting for Input** for details.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts three arguments. *continuation* is called with the values that would normally be returned by **`prompt-for-file`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`prompt-for-file`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

On Motif, the prompt itself is created by passing an appropriate pane to **`popup-confirmer`**. Arguments can be passed to the **`make-instance`** of the pane and the call to **`popup-confirmer`** using *pane-args* and *popup-args* respectively. Currently, the pane used to create the file prompter is internal to the CAPI. *pane-args* and *popup-args* are ignored on Microsoft Windows.

filename is the full pathname of the file selected, or **`nil`** if the dialog was cancelled.

successp is a flag which is **`nil`** if the dialog was cancelled, and **`t`** otherwise.

On Microsoft Windows **`prompt-for-file`** returns a third value: *filter-name* is the name of the filter that was selected in the dialog.

file-package-is-directory controls how to treat file packages on Cocoa. By default it is **`nil`**, which means that a file package is treated as file. If *file-package-is-directory* is non-nil, the a file package is treated as a directory. *file-package-is-directory* corresponds to the **`treatsFilePackagesAsDirectories`** method of **`NSSavePanel`** in Cocoa. It has no effect on other platforms.

Examples

```
(capi:prompt-for-file "Enter a filename:")
```

```
(capi:prompt-for-file "Enter a filename:"
                      :pathname "/usr/bin/cal")
```

```
(capi:prompt-for-file "Enter a filename:"
                      :ok-check 'probe-file)
```

See also

`popup-confirmer`
`prompt-for-string`
`prompt-for-directory`
10 Dialogs: Prompting for Input

prompt-for-files *Function*

Summary

Displays a dialog which returns multiple filenames.

Package

`capi`

Signature

`prompt-for-files` *message* **`&key`** *pathname ok-check filter filters if-exists if-does-not-exist file-package-is-directory operation owner pane-args popup-args continuation* **`=>`** *filenames*, *successp*, *filter-name*

Arguments

message⇓ A string or **`nil`**.

pathname⇓ A pathname designator or **`nil`**.

ok-check⇓ A function or **`nil`**.

filter⇓ A string or **`nil`**.

filters⇓ A property list.

if-exists⇓ One of **`:ok`** or **`:prompt`**.

if-does-not-exist⇓ One of **`:ok`**, **`:prompt`** or **`:error`**.

file-package-is-directory⇓
A generalized boolean.

operation⇓ One of **:open** or **:save**.

owner⇓ An owner window.

pane-args⇓ Arguments to pass to the pane.

popup-args⇓ Arguments to pass to the confirmer.

continuation⇓ A function or **nil**.

Values

filenames⇓ A list.

successp⇓ A boolean.

filter-name⇓ A string.

Description

The function **prompt-for-files** presents the user with a dialog box similarly to **prompt-for-file**, but in which multiple filenames can be selected.

message, *pathname*, *ok-check*, *filter*, *filters*, *if-exists*, *if-does-not-exist*, *file-package-is-directory*, *operation*, *owner*, *pane-args* and *popup-args* are as for **prompt-for-file**, except on Microsoft Windows where the default value of *filters* is:

```
("MS Word files" "*.doc"
  "HTML files" "*.htm;*.html"
  "Plain Text files" "*.txt;*.text"
  "All files" "*.*")
```

On Cocoa and GTK+ the default value of *filters* is:

```
("Lisp Source Files" "*.lisp;*.lsp"
 "Text Documents" "*.txt;*.text"
 "All Files" "*.*")
```

which is the same default as for **prompt-for-file**.

filenames is a list of filenames, or **nil** if the user cancels the dialog.

successp is a flag which is **nil** if the dialog was cancelled, and **t** otherwise.

filter-name is the name of the filter that was selected in the dialog.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts three arguments. *continuation* is called with the values that would normally be returned by **prompt-for-files**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **prompt-for-files** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

Notes

prompt-for-files is not implemented on Motif.

See also

prompt-for-file

prompt-for-font *Function*

Summary

Presents a dialog box allowing the user to choose a font.

Package

capi

Signature

prompt-for-font *message* **&key** *font owner* **=>** *result*, *successp*

Arguments

message⇓	A string.
font⇓	A font, a font description, or **nil**.
owner⇓	An owner window, or **nil**.

Values

result	A font, or **nil**.
successp	A boolean.

Description

The function **prompt-for-font** displays a dialog box allowing the user to choose a font.

message supplies a title for the dialog.

font, if non-nil, provides defaults for the dialog box. The default value is **nil**.

owner specifies an owner window for the dialog. See **10 Dialogs: Prompting for Input** for details.

For a description of Graphics Ports fonts and font descriptions, see **13.9 Portable font descriptions**.

See also

find-best-font
10 Dialogs: Prompting for Input

prompt-for-form *Function*

Summary

Displays a text input pane and prompts the user for a form.

Package

capi

Signature

prompt-for-form *message* **&key** *package initial-value evaluate quotify ok-check value-function pane-args popup-args continuation* **=>** *result*, *okp*

Arguments

message⇓ A string or **nil**.

package⇓ A package or **nil**.

initial-value⇓ A Lisp object.

evaluate⇓ A generalized boolean.

quotify⇓ A generalized boolean.

ok-check⇓ A function or **nil**.

value-function⇓ A function, or **nil**.

pane-args⇓ Arguments to pass to the pane.

popup-args⇓ Arguments to pass to the confirmer.

continuation⇓ A function or **nil**.

Values

result A Lisp object.

okp A boolean.

Description

The function **prompt-for-form** prompts the user for a form by providing a text input pane that the form can be typed into.

message supplies a title for the dialog.

The form is read in *package* if specified or ***package*** if not. If *evaluate* is non-nil then the result is the evaluation of the form, otherwise it is just the form itself. The printed version of *initial-value* will be placed into the text input pane as a default, unless *quotify*, which defaults to *evaluate*, specifies otherwise. If *value-function* is provided it overrides the default value function which reads the form and evaluates it when required. If *ok-check* is provided it will be passed the entered form and should return **t** if the form is a valid result.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **prompt-for-form**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **prompt-for-form** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

The prompter is created by calling **prompt-for-string**. Arguments can be passed to the **make-instance** of the pane and the call to **popup-confirmer** using *pane-args* and *popup-args* respectively, and an input history can be implemented by supplying a *history-function* or *history-symbol* in *popup-args*.

Examples

Try the following examples, and each time enter **(+ 1 2)** into the input pane.

```
(capi:prompt-for-form "Enter a form:")
```

```
(capi:prompt-for-form "Enter a form:" :evaluate nil)
```

See also

prompt-for-forms
prompt-for-string
popup-confirmer
text-input-pane
10 Dialogs: Prompting for Input

prompt-for-forms *Function*

Summary

Displays a text input pane prompting the user for a number of forms.

Package

capi

Signature

prompt-for-forms *message* **&key** *package initial-value value-function pane-args popup-args continuation* **=>** *result***,** *okp*

Arguments

message⇓	A string or **nil**.
package⇓	A package or **nil**.
initial-value⇓	A list.
value-function⇓	A function, or **nil**.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
continuation⇓	A function or **nil**.

Values

result	A list.
okp	A boolean.

Description

The function **prompt-for-forms** prompts the user for a number of forms by providing a text input pane that the forms can be typed into, and it returns the forms in a list. The forms are read in the specified *package* or ***package*** if not. If *value-function* is provided it overrides the default value function which reads space-separated forms and returns a list of them.

message supplies a title for the dialog.

The printed version of *initial-value* will be placed into the text input pane as a default.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **`prompt-for-forms`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`prompt-for-forms`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

The prompter is created by passing an appropriate pane (in this case a text input pane) to **`popup-confirmer`**. Arguments can be passed to the **`make-instance`** of the pane and the call to **`popup-confirmer`** using *pane-args* and *popup-args* respectively.

Examples

Try the following example, and enter **`1 2 3`** into the input pane.

```
(capi:prompt-for-forms "Enter some forms:")
```

See also

`prompt-for-form`
`prompt-for-string`
`popup-confirmer`
`text-input-pane`

prompt-for-integer *Function*

Summary

Prompts the user for an integer.

Package

`capi`

Signature

`prompt-for-integer` *message* **`&key`** *min max initial-value ok-check pane-args popup-args continuation* **`=>`** *result*, *successp*

Arguments

message⇓	A string.
min⇓	An integer or **`nil`**.
max⇓	An integer or **`nil`**.
initial-value⇓	An integer or **`nil`**.
ok-check⇓	A function or **`nil`**.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
continuation⇓	A function or **`nil`**.

Values

result⇓ An integer or **`nil`**.

successp A boolean.

Description

The function **`prompt-for-integer`** pops up a **`text-input-pane`** and prompts the user for an integer, which is returned in *result*.

message supplies a title for the dialog.

When *min* or *max* are specified the allowable result is constrained accordingly.

initial-value determines the initial value displayed in the dialog. *initial-value* defaults to the value of *min*, or if *min* is **`nil`** then no initial value is displayed.

Further restrictions can be applied by passing an *ok-check* function. *ok-check* should take one argument, the currently entered number, and should return **`t`** if it is valid. If *ok-check* is **`nil`** (the default) then there is no further restriction.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **`prompt-for-integer`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`prompt-for-integer`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

The prompter is created by passing **`text-input-pane`** to **`popup-confirmer`**. Arguments can be passed to the **`make-instance`** of the pane and the call to **`popup-confirmer`** using *pane-args* and *popup-args* respectively.

Examples

```
(capi:prompt-for-integer "Enter an integer:")
```

```
(capi:prompt-for-integer "Enter an integer:" :max 10)
```

```
(capi:prompt-for-integer "Enter an integer:"
                         :min 100 :max 200)
```

```
(capi:prompt-for-integer "Enter an integer:"
                         :ok-check 'evenp)
```

See also

`prompt-for-string`
`popup-confirmer`
`text-input-pane`
10 Dialogs: Prompting for Input

prompt-for-items-from-list *Function*

Summary

Prompts with a choice of items.

Package

capi

Signature

prompt-for-items-from-list *items message* **&key** *pane-args popup-args interaction choice-class continuation* **=>** *result***,** *successp*

Arguments

items⇓	A sequence.
message⇓	A string.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
interaction⇓	One of **:single-selection**, **:multiple-selection**, or **:extended-selection**.
choice-class⇓	A class name.
continuation⇓	A function or **nil**.

Values

result	A list.
successp	A boolean.

Description

The function **prompt-for-items-from-list** is similar to **prompt-with-list**. *interaction* defaults to **:extended-selection**.

See **prompt-with-list** for how *items, message*, *pane-args*, *popup-args*, *interaction*, *choice-class* and *continuation* are used.

See also

prompt-with-list

prompt-for-number *Function*

Summary

Prompts the user for a number.

Package

capi

Signature

prompt-for-number *message* **&key** *min max initial-value ok-check pane-args popup-args continuation* **=>** *result***,** *successp*

Arguments

message⇓	A string.
min⇓	A number or **nil**.
max⇓	A number or **nil**.
initial-value⇓	A number or **nil**.
ok-check⇓	A function or **nil**.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
continuation⇓	A function or **nil**.

Values

result⇓	A number or **nil**.
successp	A boolean.

Description

The function **prompt-for-number** pops up a **text-input-pane** and prompts the user for a number, which is returned in *result*.

The functionality corresponds exactly to that of **prompt-for-integer**, except that all types of numbers are allowed.

See **prompt-for-integer** for how *message*, *min*, *max*, *initial-value*, *ok-check*, *pane-args*, *popup-args*, *continuation* are used.

See also

prompt-for-integer
10 Dialogs: Prompting for Input

prompt-for-string *Function*

Summary

Displays a text input pane and prompts the user for a string.

Package

capi

Signature

prompt-for-string *message* **&key** *pane-args popup-args ok-check value-function text initial-value print-function history-symbol history-function continuation* **=>** *result***,** *okp*

Arguments

message⇓	A string.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
ok-check⇓	A function or **nil**.
value-function⇓	A function or **nil**.
text⇓	A string or **nil**.
initial-value⇓	A string or **nil**.
print-function⇓	A function or **nil**.
history-symbol⇓	A symbol.
history-function⇓	A function or **nil**.
continuation⇓	A function or **nil**.

Values

result⇓	A string or **nil**.
okp⇓	A boolean.

Description

The function **prompt-for-string** prompts the user for a string and returns that string in *result* and a flag *okp* indicating that the dialog was not cancelled. The initial string can either be supplied directly as a string using *text*, or by passing *initial-value* and a *print-function* for that value. *print-function* defaults to **princ-to-string**. The value returned can be converted into a different value by passing a *value-function*, which by default is the identity function. This *value-function* gets passed the text that was entered into the pane, and should return both the value to return and a flag that should be non-nil if the value that was entered is not acceptable. If an *ok-check* is passed, then it should return non-nil if the value about to be returned is acceptable.

prompt-for-string creates an instance of **text-input-pane** or **text-input-choice** depending on the value of *history-function*. Arguments can be passed to the **make-instance** of this pane using *pane-args*. **prompt-for-string**

then passes this pane to **popup-confirmer**. Arguments can be passed to the call to **popup-confirmer** using *popup-args*.

message supplies a title for the dialog.

history-symbol, if non-nil, provides a symbol whose value is used to store an input history, when *history-function* is not supplied. The default value of *history-symbol* is **nil**.

history-function, if supplied, should be a function designator for a function with signature:

history-function **&optional** *push-value*

history-function is called with no argument to obtain the history which is used as the *items* of the **text-input-choice**, and with the latest input to update the history.

The default value of *history-function* is **nil**. In this case, if *history-symbol* is non-nil then a history function is constructed which stores its history in the value of that symbol.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **prompt-for-string**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **prompt-for-string** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

Examples

```
(capi:prompt-for-string "Enter a string:")
```

```
(capi:prompt-for-string
 "Enter an integer:"
 :initial-value 10
 :value-function  #'(lambda (x)
                      (let ((integer
                             (ignore-errors
                              (read-from-string x))))
                      (values integer
                         (not (integerp integer))
                      ))))
```

See also

popup-confirmer
text-input-pane
10 Dialogs: Prompting for Input

prompt-for-symbol *Function*

Summary

Prompts the user for a symbol.

Package

capi

Signature

`prompt-for-symbol` *message* **`&key`** *initial-value symbols package ok-check pane-args popup-args continuation* **`=>`** *result*, *okp*

Arguments

message⇓	A string or **`nil`**.
initial-value⇓	A symbol.
symbols⇓	A list of symbols.
package⇓	A package or **`nil`**.
ok-check⇓	A function, or **`nil`**.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
continuation⇓	A function or **`nil`**.

Values

result	A symbol.
okp	A boolean.

Description

The function **`prompt-for-symbol`** prompts the user for a symbol which they should enter into the pane.

message supplies a title for the dialog.

initial-value, if non-nil, should be a symbol which is initially displayed in the pane.

The symbols that are valid can be constrained in a number of ways.

symbols, if non-nil, should be a list of all valid symbols. The default is **`nil`**, meaning all symbols are valid.

package, if non-nil, is a package in which the symbol must be available. The value **`nil`** means that the value of **`*package*`** is used, and this is the default.

ok-check is a function which when called on a symbol will return non-nil if the symbol is valid.

The prompter is created by calling **`prompt-for-string`**. Arguments can be passed to the **`make-instance`** of the pane and the call to **`popup-confirmer`** using *pane-args* and *popup-args* respectively, and an input history can be implemented by supplying a *history-function* or *history-symbol* in *popup-args*.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **`prompt-for-symbol`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`prompt-for-symbol`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

Examples

```
(capi:prompt-for-symbol "Enter a symbol:")
```

```
(capi:prompt-for-symbol "Enter a symbol:"
                        :package 'cl)
```

```
(capi:prompt-for-symbol "Enter a symbol:"
                        :symbols
                        '(foo bar baz))


(capi:prompt-for-symbol "Enter a symbol:"
                        :ok-check #'(lambda (symbol)
                                      (string< symbol "B")))
```

This last example shows how to implement a symbol prompter with an input history:

```
(defvar *my-history* (list "cdr" "car"))

(capi:prompt-for-symbol "Enter a symbol"
                        :popup-args
                        '(:history-symbol *my-history*))
```

See also

prompt-for-form
prompt-for-string
popup-confirmer
text-input-pane
10 Dialogs: Prompting for Input

prompt-for-value *Function*

Summary

Prompts the user for a form to evaluate.

Package

capi

Signature

prompt-for-value *message* **&key** *package initial-value value-function pane-args popup-args continuation* **=>** *value*, *okp*

Arguments

message⇓	A string or **nil**.
package⇓	A package or **nil**.
initial-value⇓	A symbol.
value-function⇓	A function, or **nil**.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
continuation⇓	A function or **nil**.

Values

value	A Lisp object.

okp A boolean.

Description

The function **prompt-for-value** prompts the user for a form and returns the result of evaluating that form.

The form is read in *package* if specified or ***package*** if not and the result is the evaluation of the form.

If *initial-value* is supplied it provides a default form.

If *value-function* is supplied it overrides the default value function which reads the form and evaluates it.

message supplies a title for the dialog.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **prompt-for-value**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **prompt-for-value** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

The prompter is created by passing a **text-input-pane** to **popup-confirmer**. Arguments can be passed to the **make-instance** of the pane and the call to **popup-confirmer** using *pane-args* and *popup-args* respectively.

Examples

```
(capi:prompt-for-value
 "Square"
 :initial-value '(+ 1 2 3)
 :value-function
 #'(lambda (text)
     (let ((res (eval (read-from-string text))))
       (* res res))))
```

See also

prompt-for-form

prompt-with-list *Function*

Summary

Prompts the user to select an item or items from a **choice**.

Package

capi

Signature

prompt-with-list *items message* **&key** *choice-class interaction selection selected-item selected-items value-function pane-args popup-args continuation buttons callbacks all-button none-button* **=>** *result*, *successp*

Arguments

items⇓ A sequence.

message⇓	A string.
choice-class⇓	A class name.
interaction⇓	One of **`:single-selection`**, **`:multiple-selection`**, or **`:extended-selection`**.
selection⇓	The indexes of the choice's selected items.
selected-item⇓	The selected item for a single selection choice.
selected-items⇓	A list of the selected items.
value-function⇓	A function, or **`nil`**.
pane-args⇓	Arguments to pass to the pane.
popup-args⇓	Arguments to pass to the confirmer.
continuation⇓	A function or **`nil`**.
buttons⇓	A list of strings or the keyword **`:none`**.
callbacks⇓	A list of callback specs.
all-button⇓	A string, **`nil`** or **`t`**.
none-button⇓	A string, **`nil`** or **`t`**.

Values

result⇓	A list.
successp	A boolean.

Description

The function **`prompt-with-list`** prompts the user with a **`choice`**. The user's selection is normally returned by the prompter.

items supplies the items of the **`choice`**.

message supplies a title for the **`choice`**.

choice-class determines the type of **`choice`** used in the dialog. *choice-class* defaults to **`list-panel`**, and must be a subclass of **`choice`**.

interaction determines the interaction style of the **`choice`** in the dialog. By default *interaction* is **`:single-selection`**. For single selection, the dialog has an **OK** and a **Cancel** button, while for other selection styles it has **Yes**, **No** and **Cancel** buttons where **Yes** means accept the selection, **No** means accept a null selection and **Cancel** behaves as normal. Note that *interaction* **`:multiple-selection`** is not supported for lists on macOS.

One of *selection*, *selected-item* or *selected-items* can be used to set the initial selection of the **`choice`**.

The primary returned value is usually the selected items, but a *value-function* can be supplied that gets passed the result and can then return a new result. If *value-function* is **`nil`** (this is the default), then *result* is simply the selection.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **`prompt-with-list`**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **`prompt-with-list`** returns immediately, leaving the dialog on the screen. The **`with-dialog-results`** macro provides a convenient way to create a *continuation* function.

In addition to the choice showing the items, **`prompt-with-list`** can also display a panel of push buttons (the "action buttons") which perform actions related to the choice. Note that these buttons are separated from the "dialog buttons" such as

OK and **Cancel**. The dialog buttons are controlled separately by keywords in *popup-args*.

By default, **`prompt-with-list`** does not display action buttons. However, if *interaction* is **`:multiple-selection`**, the default behavior is to display two action buttons, **All** and **None**. These change the selection to all of the items or none of the items respectively.

When *buttons* is **`:none`**, it specifies no action buttons in any case (including no **All** and **None** buttons). Otherwise *buttons* must be a list of strings specifying additional action buttons. Each of the strings specifies a button, and the string is displayed in the button.

callbacks specifies the callbacks of the buttons. It should be a list of callback specifiers matching the list in *buttons*. Each callback specifier is either a callable (a function or a symbol) which takes one argument, the choice, or a list where the **`car`** is a callable which is called as follows:

```
(apply (car callback-spec) choice (cdr callback-spec))
```

When *all-button* and *none-button* are supplied they override the default behavior of the **All** and **None** buttons. If *all-button* (*none-button*) is **`nil`**, then **All** (**None**) is not displayed. If *all-button* (*none-button*) is non-nil and *buttons* is not **`:none`**, the **All** (**None**) button is displayed, and if the value is string, that string is used instead of the default string.

The prompter is created by passing an appropriate pane (in this case an instance of class *choice-class*) to **`popup-confirmer`**. Arguments can be passed to the **`make-instance`** of the pane and the call to **`popup-confirmer`** using *pane-args* and *popup-args* respectively.

Examples

```
(capi:prompt-with-list
  '(1 2 3 4 5) "Select an item:")


(capi:prompt-with-list
 '(1 2 3 4 5) "Select some items:"
 :interaction :multiple-selection
 :selection '(0 2 4))


(capi:prompt-with-list
 '(1 2 3 4 5) "Select an item:"
 :interaction :multiple-selection
 :choice-class 'capi:button-panel)


(capi:prompt-with-list
  '(1 2 3 4 5) "Select an item:"
  :interaction :multiple-selection
  :choice-class 'capi:button-panel
  :pane-args
  '(:layout-class capi:column-layout))
```

There is a more complex example in:

```
(example-edit-file "capi/choice/prompt-with-buttons")
```

See also

`popup-confirmer`
`list-panel`
`choice`
10 Dialogs: Prompting for Input

prompt-with-list-non-focus *Function*

Summary

Raises a non-focus window.

Package

capi

Signature

prompt-with-list-non-focus *items* **&key** *owner x alternative-x right y alternative-y bottom choice-class vertical-scroll print-function selection selected-item visible-items selection-callback action-callback destroy-callback list-updater gesture-callbacks add-gesture-callbacks alternative-bottom alternative-right widget-name filtering-gesture filtering-toggle* **&allow-other-keys =>** *interface*

Arguments

items⇓ A sequence.

owner⇓ A displayed CAPI pane.

x⇓, *alternative-x*⇓, *right*⇓

Integers, or one of the keywords **:left**, **:right**, **:center** and **:centre**.

y⇓, *alternative-y*⇓, *bottom*⇓

Integers, or one of the keywords **:top**, **:bottom**, **:center** and **:centre**.

choice-class⇓ A subclass of **list-panel**.

vertical-scroll⇓ A boolean.

print-function⇓ A function designator or **nil**.

selection⇓ An integer.

selected-item⇓ An item.

visible-items⇓ A positive integer.

selection-callback⇓ A function designator or **nil**.

action-callback⇓ A function designator or **nil**.

destroy-callback⇓ A function designator or **nil**.

list-updater⇓ A function designator or **nil**.

gesture-callbacks⇓ A list of pairs of the form **(***gesture* **.** *callback***)**.

add-gesture-callbacks⇓

A list of pairs of the form **(***gesture* **.** *callback***)**.

alternative-bottom⇓ An integer, or one of the keywords **:top**, **:bottom**, **:center** and **:centre**, or **t**.

alternative-right⇓ An integer, or one of the keywords **:left**, **:right**, **:center** and **:centre**, or **t**.

widget-name⇓ A string.

filtering-gesture⇓ A Gesture Spec.

filtering-toggle⇓ A Gesture Spec.

Values

interface A **non-focus-list-interface**, or **nil**.

Description

The function **prompt-with-list-non-focus** raises a non-focus window, displaying the items *items* in a list of class *choice-class*, which should be **list-panel** or a subclass.

The non-focus window does not take the input focus, and hence does not see any keyboard input unless this is passed to it by **non-focus-maybe-capture-gesture**. It responds to mouse gestures.

Note that even moving the selection in the list vertically in response to the arrow keys cannot happen without **non-focus-maybe-capture-gesture**.

owner is required, and must be a CAPI pane visible on the screen. The position of the non-focus window is determined relative to *owner*, and the callbacks are invoked in the process of *owner*.

x, y, right, bottom, alternative-x, alternative-y, alternative-right, and *alternative-bottom* are used for positioning the window. *x*, *alternative-right*, *alternative-x* and *right* are the horizontal keywords, and one of them determines the horizontal position as described below. *y*, *alternative-bottom*, *alternative-y* and *bottom* are the vertical keywords, and one of them determines the vertical position. The values **:center** and **:centre** are synonyms here.

x and *y* specify the positioning of the left and top sides of the window, except for **:center**/**:centre**. An integer means offset in pixels from the left or top of *owner*. **:left**, **:right**, **:top** and **:bottom** mean the left/right/top/bottom of *owner*. **:center** means the center of the owner, and in this case it specifies the location of the center of the window in the x or y dimension. *x* must be supplied, unless *right* is supplied. *y* must be supplied, unless *bottom* is supplied.

right and *bottom* override *x* and *y* respectively. They specify the positioning of the right or bottom of the window, except for **:center**/**:centre**, where they are interpreted in the same way as *x* and *y*.

alternative-x, alternative-y, alternative-right, and *alternative-bottom* are used if positioning the window using *x* or *right* and *y* or *bottom* would place it outside of the screen, and are interpreted the same way as the non-alternative keywords. For example, both Editor completion and **text-input-pane** completion specify a *y* coordinate below the text, and *alternative-bottom* above the text. The decision to use the alternative variables is made independently in the horizontal and vertical directions. *alternative-right* and *alternative-bottom* can both take the special value **t**, meaning the *height* or *width* of the **screen**.

The default value of *choice-class* is **list-panel**.

selection or *selected-item* can be used to specify the initially selected item in the list. If neither of these initargs is supplied, the first item is selected.

visible-items specifies the height of the list panel when the filter is not visible. The default value of *visible-items* is 20.

vertical-scroll is supplied to **cl:make-instance** when making the list. The default value of *vertical-scroll* is **t**.

print-function is also supplied to **cl:make-instance** when making the list. The default value of *print-function* is **nil**.

selection-callback, if non-nil, should be a function of two arguments, the selected item and the non-focus interface. *selection-callback* is called (in the process of owner) when an item is selected in the list panel. Note that *callback-type* does not affect the arguments passed to *selection-callback*.

action-callback, if non-nil, should also be a function of two arguments, the selected item and the non-focus interface. *action-callback* is called (in the process of *owner*) when an item is double-clicked in the list panel, or when **Return** is passed to **non-focus-maybe-capture-gesture** (by default, see *gesture-callbacks*). Note that *callback-type* does not affect the arguments passed to *action-callback*.

destroy-callback, if non-nil, should be a function of one argument, the non-focus window (a CAPI interface). *destroy-callback* is called when the non-focus window is destroyed. It is invoked in the process of *owner*.

list-updater, if non-nil, should be a function with signature:

list-updater => *result*

list-updater is called in the process of *owner* whenever **non-focus-update** is called. *result* must be a list of items to put into the list panel, or one of the special values **t** (meaning no effect) and **:destroy** (meaning destroy the non-focus window).

gesture-callbacks and *add-gesture-callbacks* define gesture callbacks which the non-focus window can "capture" (when **non-focus-maybe-capture-gesture** is called). *gesture-callbacks* and *add-gesture-callbacks* should both be a list of pairs of the form **(***gesture* **.** *callback***)**. Each *gesture* must be a gesture specifier, that is an object that **sys:coerce-to-gesture-spec** can coerce to a **sys:gesture-spec**. Each *callback* is either a callable (symbol or function) which takes one argument, the non-focus window, or a list of the form **(***function* **.** *arguments***)**. Note that when it is a list, the window is not automatically passed to the function *function* amongst the arguments *arguments*. The gesture callbacks are used only when **non-focus-maybe-capture-gesture** is called.

add-gesture-callbacks adds more gesture callbacks to those that are implicitly defined for controlling the list panel (see **non-focus-maybe-capture-gesture**). *gesture-callbacks*, if supplied, replaces the gesture callbacks that are implicitly defined for the list panel. In both cases, a gesture callback that is defined explicitly overrides any implicitly define gesture callback.

filtering-gesture defines whether it is possible for the user to add a filter to the non-focus window with a keyboard gesture, and defines that gesture. The gesture is actually a toggle: it destroys a filter that is on, and adds a filter when none is present. When the filter is added, its text is reset and it is always enabled, that is it captures characters and **Backspace**. While the filter is visible, the list panel displays only items that match the filter (see **5.3.6 Filters**). The default value of *filtering-gesture* is a Gesture Spec matching **Control+Return**.

filtering-toggle defines whether it is possible for the user to disable/enable the filter with a keyboard gesture, and defines that gesture. When a filter is visible and enabled, the non-focus window captures characters and **Backspace** (when **non-focus-maybe-capture-gesture** is called) and passes them to the filter. When the filter is visible and disabled, characters and **Backspace** are captured. The default value of *filtering-toggle* is a Gesture Spec matching **Control+Shift+Return**.

widget-name has an effect only on GTK+ and Motif. It defines the widget name of the interface, which can then be used to define resources specific to the non-focus window. Note that the non-focus completers in **editor-pane** and **text-input-pane** use the default *widget-name* which is **"non-focus-list-prompter"**, so defining resources for non-focus-list-prompter will affect them.

If *items* is **nil**, **prompt-with-list-non-focus** returns **nil** without doing anything. Otherwise, it raises the non-focus window and returns the interface, which is of class **non-focus-list-interface**.

The non-focus window is "passive", because it does not see keyboard input. It is the responsibility of the caller to pass any keyboard input that the non-focus window needs to process to the window, by using **non-focus-maybe-capture-gesture**. In general, that should be all keyboard gestures, and **non-focus-maybe-capture-gesture** decides which gestures it wants to process.

The caller can also use **non-focus-terminate**, **non-focus-update**, **non-focus-list-toggle-filter**, **non-focus-list-add-filter**, **non-focus-list-remove-filter** and **non-focus-list-toggle-enable-filter** to control the non-focus window.

See also

list-panel
non-focus-terminate
non-focus-update
non-focus-list-toggle-filter
non-focus-list-toggle-enable-filter
non-focus-maybe-capture-gesture
10.6 In-place completion

prompt-with-message *Function*

Summary

Displays a message dialog, allowing it to be a window-modal sheet on Cocoa.

Package

capi

Signature

prompt-with-message *message* **&key** *owner* *continuation*

Arguments

message⇓ A string.

owner⇓ An owner window, or **nil**.

continuation⇓ A function or **nil**.

Description

The function **prompt-with-message** displays *message* in a dialog owned by *owner*.

If *continuation* is non-nil, then it must be a function with a lambda list that accepts two arguments. *continuation* is called with the values that would normally be returned by **prompt-with-message**. On Cocoa, passing *continuation* causes the dialog to be made as a window-modal sheet and **prompt-with-message** returns immediately, leaving the dialog on the screen. The **with-dialog-results** macro provides a convenient way to create a *continuation* function.

Examples

```
(capi:prompt-with-message
  "No items were deleted.")
```

See also

display-message-for-pane
display-message

push-button *Class*

Summary

A pane that displays either a piece of text or an image and when it is pressed it performs an action.

Package

capi

Superclasses

button
titled-object

Initargs

:alternate-callback
A callback invoked on Microsoft Windows, Cocoa and GTK+ when pressing the mouse button over the button while a platform-specific modifier key is held down.

:press-callback
A callback invoked on Microsoft Windows, GTK+ and Motif when pressing the mouse button over the button.

Accessors

button-alternate-callback
button-press-callback

Description

The class **push-button** inherits most of its behavior from **button**. Note that it is normally best to use a **push-button-panel** rather than make the individual buttons yourself, as the button panel provides functionality for handling groups of buttons. However, push buttons can be used if you need to have more control over the button's behavior.

press-callback, if non-nil, should be a function which is called when the user presses the mouse left button over the push button. The arguments to *press-callback* are as specified by *callback-type*. This initarg is not supported on Cocoa.

alternate-callback, if non-nil, should be a function. On Microsoft Windows and GTK+, it is called instead of *callback* when the button is clicked with the **Control** key held down. On Cocoa, it is called instead of *callback* when the button is clicked with the **Command** key held down. *alternate-callback* is not implemented for Motif or for other classes of **button**.

Notes

callback (from superclass **button**) is the general callback, triggered when the user clicks the button, either by pressing and releasing the mouse button or by a keyboard gesture.

press-callback is called only when the user presses the mouse button.

Examples

```
(setq button (capi:contain
              (make-instance
               'capi:push-button
```

```
                   :text "Press Me"
                   :data '(:some :data)
                   :callback #'(lambda (data interface)
                                 (capi:display-message
                                  "Pressed ~S"
                                  data)))))

(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) nil button)

(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) t button)
```

See also

radio-button
check-button
button-panel
push-button-panel
1.2.1 CAPI elements
3.10 Button elements
12 Creating Panes with Your Own Drawing and Input

push-button-panel *Class*

Summary

A pane containing a group of buttons.

Package

capi

Superclasses

button-panel

Description

The class **push-button-panel** inherits all of its behavior from **button-panel**, which itself inherits most of its behavior from **choice**. Thus, the push button panel can accept items, callbacks, and so on.

Examples

```
(defun test-callback (data interface)
       (capi:display-message
                        "Pressed ~S" data))


(capi:contain (make-instance 'capi:push-button-panel
                             :title "Press a button:"
                             :items
                               '("Press Me" "No, Me")
                             :selection-callback
                               'test-callback))
```

```
(capi:contain (make-instance 'capi:push-button-panel
                             :title "Press a button:"
                             :items
                               '("Press Me" "No, Me")
                             :selection-callback
                               'test-callback
                             :layout-class
                               'capi:column-layout))
```

```
(capi:contain (make-instance 'capi:push-button-panel
                              :title "Press a button:"
                              :items '(1 2 3 4 5 6 7 8 9)
                              :selection-callback
                                'test-callback
                              :layout-class
                                'capi:grid-layout
                              :layout-args
                                '(:columns 3)))
```

There is a further example here:

```
(example-edit-file "capi/buttons/buttons")
```

See also

push-button
radio-button-panel
check-button-panel
5 Choices - panes with items

quit-interface *Function*

Summary

Closes the top level interface containing a specified pane.

Package

capi

Signature

quit-interface *pane* **&key** *force* **=>** *result*

Arguments

pane⇓	A CAPI element.
force⇓	A boolean. The default value is **nil**.

Values

result	**t** if the interface was closed, **nil** otherwise.

Description

The function **`quit-interface`** closes the top level interface containing *pane*, but first it verifies that it is OK to do this by calling the interface's *confirm-destroy-function*. If it is OK to close the interface, it then calls **`destroy`** to do so. If *force* is true, then neither the *confirm-destroy-function* or the *destroy-callback* are called, and the window is just closed immediately.

Notes

`quit-interface` must only be called in the process of the top level interface of *pane*. Menu callbacks on that interface will be called in that process, but otherwise you probably need to use **`execute-with-interface`** or **`apply-in-pane-process`**.

Examples

Here are two examples demonstrating the use of **`quit-interface`** with the *destroy-callback* and the *confirm-destroy-function*.

```
(setq interface (capi:display
                 (make-instance
                  'capi:interface
                  :title "Test Interface"
                  :destroy-callback
                  #'(lambda (interface)
                      (capi:display-message
                       "Quitting ~S" interface)))))


(capi:apply-in-pane-process
 interface 'capi:quit-interface interface)
```

With this second example, the user is prompted as to whether or not to quit the interface.

```
(setq interface (capi:display
                 (make-instance
                  'capi:interface
                  :title "Test Interface"
                  :confirm-destroy-function
                  #'(lambda (interface)
                      (capi:confirm-yes-or-no
                       "Really quit ~S"
                       interface)))))


(capi:apply-in-pane-process
 interface 'capi:quit-interface interface)
```

See also

`destroy`
`display`
`interface`
7 Programming with CAPI Windows

radio-button *Class*

Summary

A button that can be either selected or deselected, but when selecting it any other buttons in its group will be cleared.

Package

capi

Superclasses

button
titled-object

Description

The class **radio-button** inherits most of its behavior from **button**. Note that it is normally best to use a **radio-button-panel** rather than make the individual buttons yourself, as the **button-panel** provides functionality for handling groups of buttons. However, radio buttons are provided in case you need to have more control over the button's behavior.

Examples

```
(setq button (capi:contain
              (make-instance 'capi:radio-button
                             :text "Press Me")))

(capi:apply-in-pane-process
 button #'(setf capi:button-selected) t button)

(capi:apply-in-pane-process
 button #'(setf capi:button-selected) nil button)

(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) nil button)

(capi:apply-in-pane-process
 button #'(setf capi:button-enabled) t button)
```

There is a further example here:

```
(example-edit-file "capi/buttons/buttons")
```

See also

push-button
check-button
button-panel
radio-button-panel
3.10 Button elements

radio-button-panel *Class*

Summary

A pane containing a group of buttons of which only one can be selected at any time.

Package

capi

Superclasses

button-panel

Description

The class **radio-button-panel** inherits all of its behavior from **button-panel**, which itself inherits most of its behavior from **choice**. Thus, the radio button panel can accept items, callbacks, and so forth.

Examples

```
(capi:contain (make-instance
               'capi:radio-button-panel
               :title "Select a color:"
               :items '(:red :green :blue)
               :print-function 'string-capitalize))
```

```
(setq buttons (capi:contain
                (make-instance
                 'capi:radio-button-panel
                 :title "Select a color:"
                 :items '(:red :green :blue)
                 :print-function 'string-capitalize
                  :layout-class 'capi:column-layout)))
```

```
(capi:choice-selected-item buttons)
```

There is a further example here:

```
(example-edit-file "capi/buttons/buttons")
```

See also

radio-button
push-button-panel
check-button-panel
5 Choices - panes with items

raise-interface *Function*

Summary

Raises the interface containing a specified pane to the front of the screen.

Package

capi

Signature

raise-interface *pane*

Arguments

pane⇓ A pane.

Description

The function **raise-interface** raises the window containing *pane* to the front of the screen. To push it to the back use **lower-interface**, and to iconify it use **hide-interface**.

Examples

```
(setq pane (capi:contain
             (make-instance
              'capi:text-input-pane)))

(capi:apply-in-pane-process
 pane 'capi:lower-interface pane)

(capi:apply-in-pane-process
 pane 'capi:raise-interface pane)
```

See also

activate-pane
hide-interface
interface
lower-interface
quit-interface
7.7 Manipulating top-level windows

range-pane *Class*

Summary

A class supporting **progress-bar** and **slider**.

Package

capi

Superclasses

capi-object

Subclasses

progress-bar
scroll-bar
slider

Initargs

:start	An integer specifying the lowest value of the range.
:end	An integer specifying the highest value of the range.
:slug-start	An integer specifying the start of the slug, corresponding to the current value of the range.
:slug-end	An integer specifying the end of the slug.
:callback	Called when the user changes the value.
:orientation	One of **:horizontal** (the default) or **:vertical**.

Accessors

range-start
range-end
range-slug-start
range-slug-end
range-callback
range-orientation

Description

The class **range-pane** exists to support the **progress-bar** and **slider** classes. Consult the reference pages for **progress-bar** and **slider** for further information.

See also

progress-bar
slider
3.9.4 Slider, Progress bar and Scroll bar

range-set-sizes *Function*

Summary

Set values in a **range-pane**.

Package

capi

Signature

range-set-sizes *range-pane* **&key** *start* *end* *slug-start* *slug-end* *redisplay*

Arguments

range-pane⇓ A **range-pane**.

start⇓ A real number or **nil**.

end⇓ A real number or **nil**.

slug-start⇓ A real number or **nil**.

slug-end⇓ A real number or **nil**.

redisplay⇓ A generalized boolean.

Description

The function **range-set-sizes** set the values in the **range-pane** *range-pane* for any value of *start*, *end*, *slug-start* or *slug-end* that is supplied as non-nil.

For each of *start*, *end*, *slug-start* and *slug-end*, if the value is **nil** or not supplied, the corresponding value in *range-pane* is not changed.

If *redisplay* is true (the default) then *range-pane* is redisplayed with the new values.

Notes

The values can be also set individually by the accessors **(setf range-start)** and so on. **range-set-sizes** has the advantage over the accessors that it causes fewer calls to redisplay.

See also

range-pane
3.9.4 Slider, Progress bar and Scroll bar

read-sound-file *Function*

Summary

Reads data from a sound file on Microsoft Windows and Cocoa.

Package

capi

Signature

read-sound-file *source* **=>** *array*

Arguments

source⇓ A pathname designator.

Values

array An array of element type **`(unsigned-byte 8)`**.

Description

The function **`read-sound-file`** reads data from *source* and returns an array of its contents.

Notes

1. **`read-sound-file`** can be called during image building.
2. **`read-sound-file`** is not implemented on GTK+ and Motif.

See also

`load-sound`
18.2 Sounds

record-dependent-object
unrecord-dependent-object *Functions*

Summary

Register or unregister an object for destruction when a **`pinboard-layout`** is destroyed.

Package

`capi`

Signatures

`record-dependent-object` *pinboard-layout object*

`unrecord-dependent-object` *pinboard-layout object*

Arguments

pinboard-layout⇓ A **`pinboard-layout`**.

object⇓ A Lisp object.

Description

The functions **`record-dependent-object`** and **`unrecord-dependent-object`** are part of a mechanism for destroying objects when a **`pinboard-layout`** is destroyed.

`record-dependent-object` records the object *object*, which means that when *pinboard-layout* is destroyed, **`destroy-dependent-object`** is applied to *object*.

`unrecord-dependent-object` removes *object* from the dependents, comparing objects by **`cl:equal`**.

It is possible to record the same object more than once. **`unrecord-dependent-object`** removes one occurrence of object at most. If there is no object, it does nothing.

Notes

These functions are not designed to deal with many calls to **record-dependent-object** and **unrecord-dependent-object**. If you need to deal with many objects, you can either use the *destroy-callback* of **pinboard-layout** (inherited from **output-pane**), or add a single object of your object type (class or structure) and define a **destroy-dependent-object** method for it that will deal with the many objects in an optimal way.

See also

destroy-dependent-object
pinboard-layout

rectangle *Class*

Summary

A **pinboard-object** that draws a rectangle.

Package

capi

Superclasses

pinboard-object

Initargs

:filled	A boolean, default value **nil**.

Accessors

filled

Description

The class **rectangle** provides a simple **pinboard-object** that draws a rectangle.

The rectangle is always drawn with *shape-mode* **:plain** (that is, without anti-aliasing).

filled determines whether the rectangle is filled.

See also

12.3 Creating graphical objects

redisplay-collection-item *Generic Function*

Summary

Redisplays the area in a **collection** that belongs to an item.

Package

capi

Signature

redisplay-collection-item *collection item*

Arguments

collection⇓	A **collection**.
item⇓	A Lisp object that is an item of *collection*.

Description

The generic function **redisplay-collection-item** redisplays *item* in *collection*.

There are methods supplied for **graph-pane** and **tree-view**.

See also

collection

redisplay-element *Function*

Summary

Force redisplay of an **output-pane** or a **pinboard-object**.

Package

capi

Signature

redisplay-element *element* **&optional** *x y width height*

Arguments

element⇓ An **output-pane** or a **pinboard-object**.

x⇓, *y*⇓, *width*⇓, *height*⇓

Positive reals or **nil**. Default **nil**.

Description

The function **redisplay-element** causes *element* to be redisplayed. Redisplaying causes the *display-callback* of *element* to be called. When *element* is **pinboard-object**, the *display-callback* of its **pinboard-layout** is called.

redisplay-element is special in that it can be called from any thread, as opposed to almost all of the other CAPI functions, which must be called from the thread to which *element* belongs.

x, *y*, *width* and *height* specify which part of *element* to redisplay. If *x* or *y* are **nil**, they are set to 0. If *width* is **nil**, it is set to the width of *element* minus *x*, and if *height* is **nil** it is set to the height of *element* minus *y*. Thus if **redisplay-element** is called with only *element*, it redisplays all of it.

Notes

redisplay-element is the same as **gp:invalidate-rectangle**, except that **redisplay-element** is safe to call from any thread, which **gp:invalidate-rectangle** is not.

The call to the *display-callback* is asynchronous, and there is no specific call to the *display-callback* that matches a given call to **redisplay-element**. **redisplay-element** just guarantees that, provided *element* is displayed and nothing is broken, at least one call to the *display-callback* will happen with the given rectangle or a rectangle that contains it.

Examples

This example shows use of **redisplay-element** from a timer:

```
(example-edit-file "capi/graphics/metafile-rotation.lisp")
```

See also

gp:invalidate-rectangle
output-pane
pinboard-object

redisplay-interface *Generic Function*

Summary

Updates the state of an interface.

Package

capi

Signature

redisplay-interface *interface*

Arguments

interface⇓	An **interface**.

Description

The generic function **redisplay-interface** updates the state of the interface *interface*, such as enabling and disabling menus, buttons, and so forth, that might have changed since the last call. When using this as a callback, you can use **:redisplay-interface** instead of the symbol, and then it will get passed the correct arguments regardless of the callback type.

Notes

This method is called by **popup-confirmer** to update its button's enabled state, and so it should be called when state changes in a dialog.

See also

interface
redisplay-menu-bar
redraw-pinboard-layout
display
10 Dialogs: Prompting for Input

redisplay-menu-bar *Function*

Summary

Updates the menu bar of an interface.

Package

capi

Signature

redisplay-menu-bar *interface* **&key** *redo-items*

Arguments

interface⇓	An **interface**.
redo-items⇓	A generalized boolean.

Description

The function **redisplay-menu-bar** updates the menu bar of *interface*, such that menus become enabled and disabled as appropriate.

When *redo-items* is non-nil, **redisplay-menu-bar** redoes the items in **menu** and **menu-component** that have an *items-function*, by calling the *items-function* and setting the items. The default value of *redo-items* is **t**.

Notes

redo-items defaults to **t** in order to ensure that any accelerator associated with any item is up-to-date. When the menu bar contains menus (including sub-menus and menu-components) that have an *items-function*, **redisplay-menu-bar** may take a relatively long time (tens of milliseconds). If it is called often (for example, each time the user types a character), then it is better to call **redisplay-menu-bar** with *redo-items* **nil**.

Compatibility note

This function has been superseded by **redisplay-interface**, which updates the menu bar, but also updates other state objects such as buttons, list panels and so on.

See also

interface
redisplay-interface

redraw-drawing-with-cached-display *Function*

Summary

Redraws a pane with cached display, in particular the areas that were drawn by calls to a *temp-display-callback*.

Package

capi

Signature

redraw-drawing-with-cached-display *pane*

Arguments

pane⇓ An **output-pane**.

Description

The function **redraw-drawing-with-cached-display** redraws the output pane *pane*, in particular the areas that were drawn by calls to the *temp-display-callback*. This has the effect of restoring the display to how it was in the last call to **start-drawing-with-cached-display**.

This function must be called in the scope of **start-drawing-with-cached-display** or **output-pane-free-cached-display**. Calls outside this scope have no effect.

Notes

This redraws only what it thinks needs to be redrawn. To redraw all of the pane, use **update-drawing-with-cached-display** passing only the pane.

See also

start-drawing-with-cached-display
update-drawing-with-cached-display

redraw-pinboard-layout *Function*

Summary

Redraws any pinboard objects within a specified rectangle.

Package

`capi`

Signature

`redraw-pinboard-layout` *pinboard x y width height* **`&optional`** *redisplay*

Arguments

pinboard⇓ A **`pinboard-layout`**.

x⇓, *y*⇓, *width*⇓, *height*⇓

Non-negative integers.

redisplay⇓ A generalized boolean.

Description

The function **`redraw-pinboard-layout`** causes any pinboard objects within the rectangle specified by *x*, *y*, *width* and *height* of the pinboard layout *pinboard* to get redrawn.

If *redisplay* is **`nil`**, then the redisplay will be cached until a later update. The default for *redisplay* is **`t`**.

See also

`pinboard-object`
`redraw-pinboard-object`

redraw-pinboard-object *Function*

Summary

Redraws a specified pinboard object.

Package

`capi`

Signature

`redraw-pinboard-object` *object* **`&optional`** *redisplay*

Arguments

object⇓ A **pinboard-object**.

redisplay⇓ A generalized boolean.

Description

The function **redraw-pinboard-object** causes the pinboard object *object* to be redrawn, unless *redisplay* is **nil** in which case the redisplay will be cached until a later update. The default for *redisplay* is **t**.

Examples

There are examples here:

```
(example-edit-file "capi/graphics/")
```

See also

pinboard-object
pinboard-layout
redraw-pinboard-layout

reinitialize-interface *Generic Function*

Summary

Reinitializes an existing **interface**.

Package

capi

Signature

reinitialize-interface *interface* **&rest** *initargs*

Arguments

interface⇓ An **interface**.

initargs⇓ Initialization arguments for *interface*.

Description

The generic function **reinitialize-interface** reinitializes *interface* (an existing instance of a subclass of **interface**) using *initargs*.

reinitialize-interface is called automatically by **find-interface** when this re-uses an interface.

The applied primary method specialized on **interface** does nothing. You can add methods to specialize on subclasses of **interface** which you define.

See also

find-interface
interface-reuse-p

remove-capi-object-property *Function*

Summary

Removes a property from the property list of an object.

Package

capi

Signature

remove-capi-object-property *object property*

Arguments

object⇓	A **capi-object**.
property⇓	A Lisp object.

Description

The function **remove-capi-object-property** removes the property named by *property* from the property list of *object*.

All CAPI objects contain a property list, similar to the symbol **plist**. The functions **capi-object-property** and **(setf capi-object-property)** are the recommended ways of setting properties, and **remove-capi-object-property** is the way to remove a property.

Examples

```
(setq pane (make-instance 'capi:list-panel
                          :items '(1 2 3)))

(capi:capi-object-property pane 'test-property)

(setf (capi:capi-object-property pane 'test-property)
      "Test")
(capi:capi-object-property pane 'test-property)

(capi:remove-capi-object-property pane 'test-property)
(capi:capi-object-property pane 'test-property)
```

See also

capi-object-property
capi-object
18.5 Object properties and name

remove-items *Generic Function*

Summary

Removes some items from a collection.

Package

capi

Signature

remove-items *collection list-or-predicate*

Arguments

collection⇓ A collection.

list-or-predicate⇓ A list, or a function of one argument returning a boolean value.

Description

The generic function **remove-items** removes from the **collection** *collection* those items determined by *list-or-predicate*.

If *list-or-predicate* is list, then the items removed are those matching some element of *list-or-predicate*, compared by the *test-function* of *collection*. Otherwise, the items removed are those for which the function *list-or-predicate* returns true.

This is logically equivalent to recalculating the collection items and then calling **(setf collection-items)**. However, **remove-items** is more efficient and causes less flickering on screen.

remove-items can only be used when the **collection** has the default *items-get-function* **svref**.

Notes

remove-items cannot be used a **graph-pane** or a **tree-view**.

See also

append-items
collection
replace-items
5 Choices - panes with items

replace-dialog *Function*

Summary

Replaces a replacable dialog.

Package

capi

Signature

replace-dialog *interface* **&rest** *args* **=> nil**

Arguments

interface⇓ An interface.

args⇓ Other arguments as for **display-dialog**.

Description

The function **replace-dialog** displays a dialog in the same way the **display-dialog** does, except that it also destroys the existing dialog.

interface is a CAPI interface to be displayed as a dialog.

The arguments *args* are interpreted the same as the arguments to **display-dialog**, except that *modal* is ignored. **replace-dialog** displays the dialog like **display-dialog**.

See also

display-replacable-dialog

replace-items *Generic Function*

Summary

Replaces some items in a collection.

Package

capi

Signature

replace-items *collection* *items* **&key** *start* *new-selection*

Arguments

collection⇓ A collection.

items⇓ A list.

start⇓ A non-negative integer.

new-selection⇓ A list specifying the selection.

Description

The generic function **replace-items** replaces some items in the **collection** *collection* from *items*. **replace-items** can only be used when the **collection** has the default *items-get-function* **svref**.

start should be a non-negative integer and less than the number of items in *collection*.

Items in *collection* are replaced starting at index *start*, and proceeding until the end of the list *items*, or the end of the items in *collection*. If *items* is too long, the surplus is quietly ignored. **`replace-items`** never alters the number of items in the collection.

If supplied, *new-selection* should be a list of items specifying the new selection in collection. To specify no selection, pass **`nil`**.

If *new-selection* is not supplied, then **`replace-items`** attempts to preserve the selection. If some of the selected items are replaced, then the selection on these items is removed, but if a selected item simply moves, then the selection moves with it.

Notes

`replace-items` cannot be used a **`graph-pane`** or a **`tree-view`**.

See also

`append-items`
`collection`
`remove-items`
5 Choices - panes with items

report-active-component-failure *Generic Function*

Summary

Reports on failures to find or create a component.

Package

`capi`

Signature

`report-active-component-failure` *pane component-name error-string function-name hresult*

Arguments

pane⇓	An **`ole-control-pane`**.
component-name⇓	A string or **`nil`**.
error-string⇓	A string.
function-name⇓	A symbol.
hresult⇓	An integer or **`nil`**.

Description

The generic function **`report-active-component-failure`** is used to report on failures to find or create a component.

component-name is the name of the component it tried to find.

error-string is the error string.

function-name is the name of the function that actually failed.

hresult is the hresult that came back. It may be **`nil`** if the error is that the guid of the named component could not be found.

When the system fails to open the component, it calls **report-active-component-failure**, with the first argument the **ole-control-pane** *pane*. The default method for **ole-control-pane** tries to call **report-active-component-failure** again on its top level interface. The default method on **interface** calls **error**.

You can add your own methods, specializing on subclasses of **ole-control-pane** or subclasses of **interface**.

Notes

This function is implemented only in LispWorks for Windows. Load the functionality by **(require "embed")**.

See also

ole-control-pane

reuse-interfaces-p *Accessor*

Summary

Determines whether global interface re-use is enabled.

Package

capi

Signature

reuse-interfaces-p => *reusep*

(setf reuse-interfaces-p) *reusep* **=>** *reusep*

Arguments

reusep⇓ A boolean.

Values

reusep⇓ A boolean.

Description

The accessor **reuse-interfaces-p** gets and sets a flag that controls whether global interface re-use is enabled.

If *reusep* is **t**, then **locate-interface** and **find-interface** may return existing interfaces. If *reusep* is **nil**, then **locate-interface** returns **nil** and **find-interface** returns a new interface.

See also

find-interface
locate-interface

rich-text-pane *Class*

Summary

A text pane with extended formatting.

Package

capi

Superclasses

simple-pane

Initargs

:character-format	A plist.
:paragraph-format	A plist.
:change-callback	A function called when a change is made.
:protected-callback	A function determining whether the user may edit a protected part of the text, on Microsoft Windows.
:filename	A file to display.
:text	A string or **nil**.
:text-limit	An integer.
:link-callback	Windows only: A function designator, **:open** (the default), **:ignore** or **nil**.

Accessors

rich-text-pane-change-callback
rich-text-pane-limit
rich-text-pane-text

Description

The class **rich-text-pane** provides a text editor which supports character and paragraph formatting of its text.

character-format is the default character format. It is a plist which is interpreted in the same way as the *attributes-plist* argument of **set-rich-text-pane-character-format**. The default value of *character-format* is **nil**.

paragraph-format is the default paragraph format. It is a plist which is interpreted in the same way as the *attributes-plist* argument of **set-rich-text-pane-paragraph-format**. The default value of *paragraph-format* is **nil**.

change-callback, if non-nil, is a function of two arguments: the pane itself, and a keyword denoting the type of change. This second argument is either **:text** or **:selection**. The default value of *change-callback* is **nil**.

protected-callback, if supplied, is called when the user tries to modify protected text. (Text is protected by setting the protected attribute, see **set-rich-text-pane-character-format**.) *protected-callback* must be a function of four arguments: the pane itself, bounding indexes of the protected text, and a boolean which is true when the change would affect the selection. If the change would affect just a single character, this last argument is **nil**. If *protected-callback* returns **nil**, then the change is not performed. If *protected-callback* is not supplied, then the user cannot modify protected text. *protected-*

callback is supported only on Microsoft Windows.

filename, if non-nil, should be a string or pathname naming a file to display in the pane. *filename* takes precedence over *text* if both are non-nil.

text, if non-nil, should be a string which is displayed in the pane if *filename* is **`nil`**.

text-limit, if non-nil, should be an integer which is an upper bound for the length of text displayed in the pane.

link-callback can be used on Windows to control what happens when the user clicks on a link in the text. By default, LispWorks opens the link in the default browser. *link-callback* can be used to change this behavior. **`:open`** and **`nil`** give the default behaviour. **`:ignore`** means that LispWorks ignores gestures for the hyperlink. Otherwise, *link-callback* must be a function designator that takes three arguments: the pane, the gesture that the user entered, and the URL of the hyperlink (a string). The gesture conforms to the syntax of the input model as described in **12.2 Receiving input from the user**. Currently it is always either the keyword **`:motion`**, or a list specifying a button mapping as in **12.2.1.3 Button mappings**. *link-callback* should do any processing that is required, including opening the URL if appropriate. When the cursor is moved outside of a link, *link-callback* is called with getsure **`:motion`** and the URL is **`nil`**.

Notes

1. **`rich-text-pane`** is supported only on Microsoft Windows, and Cocoa in macOS 10.3 and later. Some of its features are supported only on Microsoft Windows, as mentioned above.
2. *change-callback* and *protected-callback* are not yet implemented on Cocoa.
3. The functions that are specific to **`rich-text-pane`** cannot be called before the pane is created. If you need to perform operations on the pane before it appears, and which cannot be performed using the initargs, the best approach is to define an **`:after`** method on **`interface-display`** on the class of the interface containing the **`rich-text-pane`**, and perform the operations inside this method.

Examples

For an example of using **`rich-text-pane`**, see:

```
(example-edit-file "capi/applications/rich-text-editor")
```

See also

`print-rich-text-pane`
`rich-text-pane-character-format`
`rich-text-pane-operation`
`set-rich-text-pane-character-format`
`rich-text-pane-paragraph-format`
`set-rich-text-pane-paragraph-format`
3.6 Displaying rich text

rich-text-pane-character-format *Function*

Summary

Returns the character format.

Package

capi

Signature

rich-text-pane-character-format *pane* **&key** *selection* **=>** *result*

Arguments

pane⇓ A **rich-text-pane**.

selection⇓ Must be **t**. This argument is deprecated.

Values

result⇓ A plist.

Description

The function **rich-text-pane-character-format** returns as a plist the current character attributes for *pane*.

If there is a current selection in the pane, then the attributes are those set for the selected text. If there is no selection, then it gets the "typing attributes", which are applied to characters that are typed by the user. Note that any cursor movement changes these attributes, so their values are ephemeral.

Supplying *selection* is deprecated. If *selection* is **nil** an error is signalled. The default value of *selection* is **t**.

An attribute appears in *result* only if its value is the same over all of the range. Therefore this form:

```
(getf
 (capi:rich-text-pane-character-format pane) :bold
 :unknown)
```

will return:

- **t** if all the selection is bold.
- **nil** if all the selection is not bold.
- **:unknown** if the selection is only partially bold.

For the possible attributes, see **set-rich-text-pane-character-format**.

Compatibility note

The value **nil** for the keyword argument **:selection** is not supported in LispWorks 6.1 and later. See the description above for details of the current behavior with respect to the current selection in the **rich-text-pane**.

See also

rich-text-pane
set-rich-text-pane-character-format

rich-text-pane-operation *Function*

Summary

Gets and sets values and performs various operations on a pane.

Package

capi

Signature

rich-text-pane-operation *pane operation* **&rest** *args* **=>** *result*, *result2*

Arguments

pane⇓ A **rich-text-pane**.

operation⇓ A keyword specifying the operation to perform.

args⇓ The value or values to use, when the operation is setting something.

Values

result⇓ Various, see below.

result2⇓ Returned only for *operation* **:get-selection**, see below.

Description

The function **rich-text-pane-operation** gets and sets values and performs various operations on *pane*.

The valid values of *operation* on Microsoft Windows and Cocoa are:

:pastep, **:cutp** or **:copyp**

result is a boolean indicating whether it is currently possible to perform a **:paste**, **:cut** or **:copy** operation.

:paste, **:cut**, or **:copy**

Performs the indicated operation.

:select-all Selects all the text.

:set-selection *args* should be two integers *start* and *end*. Sets the selection to the region bounded by *start* (inclusive) and *end* (exclusive).

:get-selection Returns as multiple values the bounding indexes of the selection. *result* is the start (inclusive) and *result2* is the end (exclusive). If there is no selection, both values are the index of the insertion point.

:can-undo or **:can-redo**

result is a boolean indicating whether it is currently possible to perform an **:undo** or **:redo** operation.

:undo Undoes the last editing operation. Note that, after typing, it is the whole input, rather than a single character, that is undone. The **:undo** operation may be repeated successively, to undo previous editing operations in turn.

Note: with RichEdit 1.0, **:undo** does not work repeatedly - it only undoes one previous editing operation. See **rich-text-version**.

:redo Undoes the effect of the last **:undo** operation. The **:redo** operation may be repeated successively, to cancel the effect of previous **:undo** operations in turn.

Note: with RichEdit 1.0, **:redo** does not work. See **rich-text-version**.

:get-modified *result* is the value of a boolean modified flag. This flag can be set by the **:set-modified** operation. Also, editing the text sets it to true.

:set-modified Sets the modified flag. The argument is a boolean.

:save-file Saves the text to a file. Details below.

:load-file Loads the text from a file. Details below.

Additionally these values of *operation* are valid on Microsoft Windows, only:

:get-word-wrap Returns a value indicating the word wrap, which can be the keyword **:none**. *result* can also be the keyword **:window** or a CAPI printer object, meaning that the text wraps according to the width of the window or the printer.

:set-word-wrap Sets the word wrap. The argument can be as described for **:get-word-wrap**, and additionally it can be the keyword **:printer**, meaning the **current-printer**.

:hide-selection Specifies whether the selection should be hidden (not highlighted) when *pane* does not have the focus. The argument is a boolean.

For operations **:save-file** and **:load-file**, *args* is a lambda list:

filename **&key** *selection format plain-text*

filename is the file to save or load.

selection is a boolean, with default value **nil**.

format is **nil** or a keyword naming the file format. Values include **:rtf** and **:text** meaning Rich Text Format and text file respectively.

plain-text is a boolean, with default value **nil**.

With *operation* **:save-file**, if *selection* is true, only the current selection is saved. If *selection* is **nil**, all the text is saved. The default value of *format* is **:rtf** and there are two further allowed values, **:rtfnoobjs** and **:textized**. These are like **:rtf** and **:text** except in the way they deal with COM objects. See the documentation for SF_RTFNOOBJS and SF_TEXTIZED in the EM_STREAMOUT entry in the MSDN for details. When saving with *format* **:rtf** or **:rtfnoobjs**, if *plain-text* is true, then keywords that are not common to all languages are ignored. With other values of *format*, *plain-text* has no effect.

With *operation* **:load-file**, if *selection* is true, the unselected text is preserved. If there is a selection, the new text replaces it. If there is no selection, the new text is inserted at the current insertion point. If *selection* is **nil**, all the text is replaced.

The default value of *format* is **nil**, meaning that the RTF signature is relied upon to indicate a Rich Text Format file. If *plain-text* is true, then keywords that are not common to all languages are ignored.

Examples

```
(setq rtp
      (capi:contain
       (make-instance
        'capi:rich-text-pane
        :text (format nil "First paragraph.~%Second paragraph, a little longer.~%Another paragraph,
 which should be long long enough that it spans more than one line. ~%" ))))
```

Set the selection to characters 9 to 18:

```
(capi:rich-text-pane-operation rtp :set-selection 9 18)
```

Write all the text to a file in text format:

```
(capi:rich-text-pane-operation
 rtp :save-file "mydoc.txt" :format :text)
```

Paste:

```
(capi:rich-text-pane-operation rtp :paste)
```

See also

rich-text-pane
rich-text-version

rich-text-pane-paragraph-format *Function*

Summary

Returns the paragraph format.

Package

capi

Signature

rich-text-pane-paragraph-format *pane* **=>** *result*

Arguments

pane⇓ A **rich-text-pane**.

Values

result A plist.

Description

The function **`rich-text-pane-paragraph-format`** returns as a plist the paragraph attributes of the current paragraphs in *pane*.

For the possible attributes, see **`set-rich-text-pane-paragraph-format`**.

See also

`rich-text-pane`

rich-text-version *Function*

Summary

Identifies the version of RichEdit in use, on Microsoft Windows.

Package

`capi`

Signature

`rich-text-version =>` *result*

Values

result⇓ A keyword indicating the version of the RichEdit control in use.

Description

The function **`rich-text-version`** returns the version of RichEdit that is being used to implement **`rich-text-pane`**.

result is **`:rich-edit-2.0`** if RichEdit 2.0 or newer is loaded. Otherwise *result* is **`:rich-edit-1.0`**.

`rich-text-version` is supported only on Microsoft Windows.

See also

`rich-text-pane`

right-angle-line-pinboard-object *Class*

Summary

A subclass of **`pinboard-object`** that displays a line drawn around two edges of the area enclosed by the pinboard object.

Package

`capi`

Superclasses

line-pinboard-object

Initargs

:type	The type of line.

Description

The class **right-angle-line-pinboard-object** is a a subclass of **line-pinboard-object** which displays a line around the edge of the pinboard object rather than diagonally.

type can be one of two values.

:vertical-first	Draw top-left to bottom-left to bottom-right.
:horizontal-first	Draw top-left to top-right to bottom-right.

The main use of this class is to produce graphs with right-angled edges rather than diagonal ones.

Examples

```
(capi:contain
 (make-instance
  'capi:right-angle-line-pinboard-object
  :start-x 20 :start-y 20
  :end-x 280 :end-y 100))

(capi:contain
 (make-instance
  'capi:right-angle-line-pinboard-object
  :start-x 20 :start-y 120
  :end-x 280 :end-y 200
  :type :horizontal-first))
```

See also

pinboard-layout
12.3 Creating graphical objects

row-layout *Class*

Summary

A layout which arranges its children in a row.

Package

capi

Superclasses

grid-layout

Initargs

:ratios	The size ratios between the layout's children.
:adjust	The vertical adjustment for each child.
:gap	The gap between each child.
:uniform-size-p	If **t**, each child in the row has the same width.

Accessors

layout-ratios

Description

The class **row-layout** lays its children out in a row. It inherits the behavior from **grid-layout**. The *description* is a list of the layout's children, and the layout also translates the initargs *ratios*, *adjust*, *gap* and *uniform-size-p* into the grid layout's equivalent arguments *x-ratios*, *y-adjust*, *x-gap* and *x-uniform-size-p*.

description may also contain the keywords **:divider** and **:separator** which create a divider or separator as a child of the **row-layout**. The user can move a divider, but cannot move a separator.

When specifying **:ratios** in a row with **:divider** or **:separator**, you should use **nil** to specify that the divider or separator is given its minimum size.

Examples

```
(setq row (capi:contain
           (make-instance
            'capi:row-layout
            :description
            (list
             (make-instance 'capi:push-button
                            :text "Press me")
             (make-instance 'capi:title-pane
                            :text "Title")
             (make-instance 'capi:list-panel
                            :items '(1 2 3)))
            :adjust :center)))

(capi:apply-in-pane-process
 row #'(setf capi:layout-y-adjust) :bottom row)

(capi:apply-in-pane-process
 row #'(setf capi:layout-y-adjust) :top row)
```

This last example shows a row with a stretchable dummy pane between two other elements which are fixed at their minimum size. Try resizing it:

```
(capi:contain
 (make-instance 'capi:row-layout
                :description
                (list (make-instance 'capi:push-button
                                     :text "foo")
                      nil
                      (make-instance 'capi:push-button
                                     :text "bar"))
                :ratios '(nil 1 nil)))
```

See also

`column-layout`
1.2.1 CAPI elements
5.2 Button panel classes
6 Laying Out CAPI Panes
7 Programming with CAPI Windows
11 Defining Interface Classes - top level windows

screen *Class*

Summary

An object that represents a known monitor screen.

Package

`capi`

Superclasses

`capi-object`

Subclasses

`color-screen`
`mono-screen`

Initargs

`:width`	The width in pixels of the screen.
`:height`	The height in pixels of the screen.
`:number`	The screen number.
`:depth`	The number of color planes in the screen.
`:interfaces`	A list of all of the interfaces visible on the screen.

Readers

`screen-width`
`screen-height`
`screen-number`
`screen-depth`
`screen-interfaces`
`screen-width-in-millimeters`
`screen-height-in-millimeters`

Description

The class **`screen`** represents the screen of a monitor.

When the CAPI initializes itself it creates one or more screen objects and they are then used to specify where a window is to appear. A **`screen`** object can also be queried for information that the program may need to know about the screen that it is working on, such as its width, height and depth.

On Microsoft Windows and Cocoa there is exactly one CAPI screen. When there are multiple monitors, there are several rectangles of pixels within the single CAPI screen.

On Motif, there is one CAPI screen for each X11 screen.

Compatibility note

In LispWorks for Macintosh 4.3 there is one CAPI screen for each Cocoa screen. In LispWorks for Macintosh 4.4 and later, there is exactly one CAPI screen.

Examples

```
(setq screen (capi:convert-to-screen))


(capi:screen-width screen)


(capi:screen-height screen)


(capi:display (make-instance
               'capi:interface :title "Test")
                               :screen screen)


(capi:screen-interfaces screen)
```

See also

convert-to-screen
3.13 Screens
10.4 Dialog Owners
11 Defining Interface Classes - top level windows

screen-active-interface *Function*

Summary

Returns the active interface on a screen.

Package

capi

Signature

screen-active-interface *screen* **=>** *interface*

Arguments

screen⇓ A **screen** or **document-container**.

Values

interface An **interface**, or **nil**.

Description

The function **screen-active-interface** returns the currently active interface on the **screen** *screen*, or **nil** if no CAPI interface is active or if this cannot be determined.

screen-active-interface also works with **document-container**, returning the active interface within the container.

See also

document-container
screen
3.13 Screens

screen-active-p *Function*

Summary

Determines whether a screen is active.

Package

capi

Signature

screen-active-p *screen* **=>** *result*

Arguments

screen⇓ A screen.

Values

result A boolean.

Description

The function **screen-active-p** is the predicate for whether *screen* is active.

Notes

A screen is normally "active". It can become inactive only when it "dies", which can happen on X interface (GTK+ or Motif) when the X connection get broken for any reason.

See also

screen
3.13 Screens

screen-internal-geometries *Function*

Summary

Returns the internal geometries of all the monitors of a screen.

Package

`capi`

Signature

`screen-internal-geometries` *screen* **`=>`** *internal-geometries*

Arguments

screen⇓	A CAPI screen.

Values

internal-geometries⇓	A list of screen rectangles.

Description

The function **`screen-internal-geometries`** returns the internal geometries of all the "monitors" of *screen*. A "monitor" typically corresponds to a physical monitor, but can be anything that the underlying GUI system considers a monitor.

The internal geometry of a monitor is a rectangle which excludes "system areas" like taskbars and global menu bars and so on. Examples of these include the Windows taskbar, the macOS menu bar, and the macOS Dock. See **`screen-internal-geometry`** for information about displaying CAPI windows in system areas.

Each internal geometry is represented as a screen rectangle. A screen rectangle is a list of four numbers: *x* and *y* being the coordinates as offsets from the top-left of the primary monitor, and *width* and *height*.

The first screen rectangle in *internal-geometries* corresponds to the usable area of the primary monitor.

Notes

On GTK+ when using a desktop with separate workspaces, the workspaces may be considered as separate "monitors". When there are multiple real monitors, the values may be incorrect. You can use **`screen-monitor-geometries`** to check the number of monitors, and to check the full size of the monitors.

See also

`pane-screen-internal-geometry`
`virtual-screen-geometry`
`screen-internal-geometry`
`screen-monitor-geometries`
3.13 Screens
4.3 Support for multiple monitors
11.6 Querying and modifying interface geometry

screen-internal-geometry *Function*

Summary

Returns the geometry of the unobscured region of a screen or document container.

Package

`capi`

Signature

`screen-internal-geometry` *screen* **`=>`** *x*, *y*, *width*, *height*

Arguments

screen⇓ A **`screen`**.

Values

x⇓ An integer.

y⇓ An integer.

width⇓ A positive integer.

height⇓ A positive integer.

Description

The function **`screen-internal-geometry`** returns the geometry (as multiple values representing a screen rectangle) of the region of *screen* that can be used to display windows without obstruction. This region excludes "system areas" like menubar and taskbar and so on. Examples of these include the Windows taskbar, the macOS menu bar and the macOS Dock.

x and *y* are the screen rectangle's coordinates as offsets from the top-left of the primary monitor, and *width* and *height* are its dimensions.

On Microsoft Windows **`screen-internal-geometry`** works with **`document-container`**, returning the current size of the container (which may vary over time).

Notes

1. The internal geometry is a snapshot of the unobscured region of a screen. If a system area moves or changes size, then the screen rectangle returned by **`screen-internal-geometry`** changes.
2. It may be possible to display a CAPI window outside the screen's internal geometry, for example under the macOS Dock, but it will be obscured.
3. The primary monitor is that represented by the first screen rectangle in the list returned by **`screen-internal-geometries`**.

See also

`document-container`

pane-screen-internal-geometry
screen
screen-internal-geometries
3.13 Screens
4.3 Support for multiple monitors
11.6 Querying and modifying interface geometry

screen-logical-resolution *Function*

Summary

Returns the logical resolution of *screen*.

Package

capi

Signature

screen-logical-resolution *screen* **=>** *xlogres*, *ylogres*

Arguments

screen⇓	A screen.

Values

xlogres, *ylogres*	Integers representing the logical resolution of *screen* in DPI.

Description

The function **screen-logical-resolution** returns the logical resolution of *screen*, as dots per inch in the x and y directions.

See also

screen
3.13 Screens

screen-monitor-geometries *Function*

Summary

Returns the geometries of all of a screen's monitors.

Package

capi

Signature

`screen-monitor-geometries` *screen* **=>** *monitor-geometries*

Arguments

screen⇓ A CAPI screen.

Values

monitor-geometries⇓ A list of screen rectangles.

Description

The function **`screen-monitor-geometries`** returns the geometries of all the monitors of *screen*. A monitor corresponds to an entity that the host machine regards as a physical monitor. **`screen-monitor-geometries`** ignores software manipulations like the desktop on GTK+.

The monitor geometry is a rectangle which includes all of its display area, including "system areas" like menubar and taskbar and so on. Examples of these include the Windows taskbar, the macOS menu bar and the macOS Dock.

Each monitor geometry screen rectangle is represented by a list of four numbers: the *x* and *y* coordinates as offsets from the top-left of the primary monitor, and the *width* and *height*.

The first screen rectangle in *monitor-geometries* corresponds to the primary monitor.

Notes

1. **`screen-monitor-geometries`** differs from **`screen-internal-geometries`** by returning screen rectangles which include all the monitor areas, and also by ignoring desktop manipulations.

2. You cannot display a CAPI window on the macOS menu bar. You can display a CAPI window in the area occupied by the macOS Dock or the Windows task bar, but the window will be obscured.

See also

`pane-screen-internal-geometry`
`screen-internal-geometries`
`virtual-screen-geometry`
3.13 Screens
4.3 Support for multiple monitors
11.6 Querying and modifying interface geometry

screens *Function*

Summary

Returns the active screens for a library.

Package

`capi`

Signature

screens &optional *library* **=>** *result*

Arguments

library⇓ A library name, a list, or **:any**.

Values

result A list.

Description

The function **screens** returns as a list all the active screens for *library*.

A library name is a keyword naming a library, currently **:win32** on Microsoft Windows, **:gtk** on GTK+, **:motif** on Motif and **:cocoa** on macOS with the native GUI.

library can be a library name, or a list of library names, or the keyword **:any**, meaning all the libraries. The default value of *library* is the result of **default-library**.

See also

default-library
screen
3.13 Screens

scroll *Generic Function*

Summary

Moves the scrollbar and calls the *scroll-callback*.

Package

capi

Signature

scroll *self scroll-dimension scroll-operation scroll-value* **&rest** *options*

Arguments

self⇓ A pane that supports scrolling.

scroll-dimension⇓ **:vertical**, **:horizontal** or **:pan**.

scroll-operation⇓ **:move**, **:step** or **:page**.

scroll-value⇓ An integer, or a list of two integers, or a keyword, or a list of two keywords.

options⇓ A list.

Description

The generic function **scroll** works for panes that support scrolling - these are subclasses of **output-pane** and **layout**.

scroll moves the scrollbar of a scrollable pane *self* according to *scroll-dimension*, *scroll-operation* and *scroll-value*. It then calls the *scroll-callback* (see **output-pane**) with these arguments and *options*.

scroll-dimension determines whether the scrolling is vertical, horizontal or, if the value is **:pan**, in both dimensions.

scroll-operation determines the extent of the scroll. The value **:move** means that the pane scrolls to the position on the scroll range given by *scroll-value*, regardless of the current scroll position. The value **:step** means scroll from the current scroll position by *scroll-value* times the scroll step size. In the case of panes which do their own scrolling the scroll step size is determined by the operating system (OS). In the case of panes for which the CAPI computes the scroll, the scroll step size is as described in **with-geometry**. The value **:page** means scroll from the current scroll position by *scroll-value* times the scroll page size (which is also determined by the OS or the pane's geometry).

scroll-value should be an integer or keyword if *scroll-dimension* is **:horizontal** or **:vertical**. Allowed keyword values are **:start** and **:end**. *scroll-value* should be a list of two integers or keywords representing the horizontal and vertical scroll values if *scroll-dimension* is **:pan**.

options is a list containing arbitrary user data.

Compatibility note

scroll supersedes **set-scroll-position**, which is deprecated and no longer exported. The call:

```
(capi:scroll pane :pan :move (list x y))
```

is equivalent to:

```
(capi:set-scroll-position pane x y)
```

See also

ensure-area-visible
get-scroll-position
output-pane
set-horizontal-scroll-parameters
set-vertical-scroll-parameters
with-geometry
7 Programming with CAPI Windows

scroll-bar *Class*

Summary

A pane which displays a scroll bar.

Package

capi

Superclasses

range-pane
simple-pane
titled-object

Initargs

:line-size	The distance scrolled by the scroll-line gesture.
:page-size	The distance scrolled by clicking inside the scroll bar.
:callback	A function called after a scroll gesture, or **nil**.

Accessors

scroll-bar-line-size
scroll-bar-page-size

Description

The class **scroll-bar** implements panes which display a scroll bar and call a callback when the user scrolls. It is not however the most usual way to add scroll bars - see the note below about **simple-pane**.

line-size is the logical size of a line, and is the distance moved when the user enters a scroll-line gesture, that is clicking on one of the arrow buttons at either end of the scroll bar or using a suitable arrow key. The default value of *line-size* is 1.

page-size is the logical size of a page, and is the distance moved when the user clicks inside the scroll bar. The default value of *page-size* is 10.

callback can be **nil**, meaning there is no callback. This is the default value. Otherwise, is a function of four arguments, the interface containing the scroll-bar, the scroll-bar itself, the mode of scrolling and the amount of scrolling. It has this signature:

callback *interface scroll-bar how where*

how can be one of **:line**, **:page**, **:move**, or **:drag**.

If *how* is **:line**, then *where* is an integer indicating how many lines were scrolled.

If *how* is **:page**, then *where* is an integer indicating how many pages were scrolled.

If *how* is **:move** or **:drag**, then *where* is an integer giving the new location of the *slug-start*, or **:start** or **:end**.

Notes

1. The location of the slug can be found by the **range-pane** accessor **range-slug-start**.
2. Rather than using **scroll-bar**, it is more usual to add scroll bars to a pane by the **simple-pane** initargs **:horizontal-scroll** and **:vertical-scroll**

Examples

```
(defun sb-callback (interface sb how where)
  (declare (ignorable interface))
  (format t "Scrolled ~a where ~a : ~a~%"
          how where (range-slug-start sb)))

(contain
```

```
(make-instance 'capi:scroll-bar
               :callback 'sb-callback
               :page-size 10
               :line-size 2
               :visible-min-width 200))
```

See also

simple-pane
3.9.4 Slider, Progress bar and Scroll bar

scroll-if-not-visible-p *Accessor Generic Function*

Summary

Accesses the *scroll-if-not-visible-p* attribute of a pane.

Package

capi

Signature

scroll-if-not-visible-p *pane* **=>** *value*

(setf scroll-if-not-visible-p) *value* *pane* **=>** *value*

Method signatures

scroll-if-not-visible-p (*pane* **simple-pane)**

(setf scroll-if-not-visible-p) *value* **(***pane* **simple-pane)**

Arguments

pane⇓	A pane.
value⇓	One of **t**, **nil** or **:non-mouse**.

Values

value⇓	One of **t**, **nil** or **:non-mouse**.

Description

The accessor generic function **scroll-if-not-visible-p** gets and sets the *scroll-if-not-visible-p* attribute of *pane*.

value can be one of the following:

t	When *pane* is given the input focus, and it is not fully visible, and its parent can be scrolled to make the pane visible, then the parent is scrolled automatically. This is the default value.
nil	Never scroll the parent to make a pane visible.
:non-mouse	Like **t**, except that it does not scroll when the focus is given as a result of a mouse click in *pane*.

scroll-if-not-visible-p is called by CAPI each time it may need to scroll the parent. The method on **simple-pane** returns a value that is kept internally, and can be set by the default setf method.

You can specialize **scroll-if-not-visible-p** on your classes, but note that it is called often when the user clicks on any pane, so it must be reasonably fast.

The setter sets the *scroll-if-not-visible-p* attribute. It is called when the initarg **:scroll-if-not-visible-p** is used in making a **simple-pane** (or a subclass) instance, and can be called by your program. *value* must be **t**, **nil** or **:non-mouse**.

The method on **simple-pane** sets the internal value that is used by **scroll-if-not-visible-p** on **simple-pane**.

See also

simple-pane
7 Programming with CAPI Windows

search-for-item *Generic Function*

Summary

The generic function **search-for-item** returns the index of an item in a collection.

Package

capi

Signature

search-for-item *collection item*

Arguments

collection⇓	A **collection**.
item⇓	A Lisp object.

Description

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

The search is done by sequentially comparing *item* to each item in *collection* using the collection's *test-function*, which is **cl:eq** by default.

search-for-item is the counterpart function to **get-collection-item** which given an index, finds the appropriate item.

See also

get-collection-item
collection

selection *Function*

Summary

Returns the primary selection.

Package

capi

Signature

selection *self* **&optional** *format* **=>** *result*

Arguments

self⇓	A displayed CAPI pane or interface.
format⇓	A keyword.

Values

result	A string, an **image**, a Lisp object, or **nil**.

Description

The function **selection** returns the contents of the primary selection as a string, or **nil** if there is no selection.

format controls what kind of object is read. The following values of *format* are recognized:

:string	The object is a string. This is the default value.
:image	The object is of type **image**, converted from whatever format the platform supports.
:value	The object is the Lisp value.

When *format* is **:image**, the image returned by **selection** is associated with *self*, so you can free it explicitly with **free-image** or it will be freed automatically when the pane is destroyed.

On Microsoft Windows there is no notion of selection, so this mechanism is internal to Lisp.

Note that X applications may or may not use the primary selection for their paste operations. For instance, Emacs is configurable by the variable **interprogram-paste-function**.

See also

clipboard
free-image
image
selection-empty
set-selection
18.6 Clipboard

selection-empty *Function*

Summary

Determines whether there is a primary selection of a particular kind.

Package

capi

Signature

selection-empty *self* **&optional** *format* **=>** *result*

Arguments

self⇓	A displayed CAPI pane or interface.
format⇓	A keyword.

Values

result	**t** or **nil**.

Description

The function **selection-empty** returns **nil** if there is a primary selection of the kind indicated by *format* associated with *self*, or **t** if there is no such selection.

format controls what kind of object is checked. The following values of *format* are recognized:

:string	The object is a string. This is the default value.
:image	The object is of type **image**, converted from whatever format the platform supports.
:value	The object is the Lisp value.

See also

image
selection
18.6 Clipboard

set-application-interface *Function*

Summary

Specifies the main Cocoa application interface.

Package

capi

Signature

set-application-interface *interface*

Arguments

interface⇓ An object of type **cocoa-default-application-interface**.

Description

The function **set-application-interface** sets *interface* as the main application interface. This interface is used to supply the application menu and receives various callbacks associated with the application.

set-application-interface must be called before any CAPI functions that make the **screen** object (such as **convert-to-screen** and **display**).

interface should not be displayed like a normal interface.

An application can only have one application menu and one dock menu. Because the LispWorks IDE already provides these menus, calling **set-application-interface** while running the LispWorks IDE will add a submenu to the **LispWorks** application menu to contain the *application-menu* and *menu-bar-items* of your application, and you can test them there. Likewise, a submenu will be added to the LispWorks Dock icon menu. Other aspects of the application interface can only be tested when running it standalone.

set-application-interface is only applicable when running under Cocoa.

Examples

```
(example-edit-file "capi/applications/cocoa-application")

(example-edit-file "capi/applications/cocoa-application-single-window")

(example-edit-file "delivery/macos/multiple-window-application")

(example-edit-file "delivery/macos/single-window-application")
```

See also

cocoa-default-application-interface

set-button-panel-enabled-items *Generic Function*

Summary

Sets the enabled state of the items in a button panel.

Package

capi

Signature

set-button-panel-enabled-items *button-panel* **&key** *enable disable set test key*

Arguments

button-panel⇓ A **button-panel**.

enable⇓ A list.

disable⇓ A list.

set⇓ A boolean.

test⇓ A function.

key⇓ A function.

Description

The generic function **set-button-panel-enabled-items** sets the enabled state of the items in *button-panel*. If *set* is **t**, then *enable* is ignored and all items are enabled except those in *disable*. If *set* is **nil**, *disable* is ignored and all items are disabled except those in *enable*. If *set* is not given, the items in *enable* are enabled and the items in *disable* are disabled. If an item is in both lists, it is enabled. A button is in a list when the result of calling *key* on the *data* of the button matches one of the items in the list. A match is defined as a non-nil return value from calling *test*. The default value for *test* is **cl:equal**. *key* defaults to **identity**.

See also

button-panel
redisplay-interface

set-clipboard *Function*

Summary

Sets the contents of the system clipboard.

Package

capi

Signature

set-clipboard *self value* **&optional** *string plist* **=>** *result*

Arguments

self⇓ A displayed CAPI pane or interface.

value⇓ A Lisp object (not necessarily a string) to make available within the local Lisp image.

string⇓ The string representation of *value* to export, or **nil**.

plist⇓ A property list.

Values

result A string, or **nil**.

Description

The function **set-clipboard** sets the contents of the system clipboard associated with *self*.

If *string* is non-nil, then the text on the system clipboard is set to *string*. If *string* is **nil** and *value* is a string, then text on the system clipboard is set to *value*. Otherwise, no text is set on the system clipboard.

In addition, *value* is made available within the local Lisp image when calling **clipboard**.

plist is a plist of additional format/value pairs to export to the system clipboard. The currently supported formats are as described for **clipboard**. You can export more than one format simultaneously.

In Microsoft Windows applications (including LispWorks in Windows emulation mode), the contents of the system clipboard is usually accessed by the user with the **Ctrl+V** gesture.

The X clipboard can be accessed by the **Ctrl+V** gesture in KDE/Gnome emulation, or by running the program **xclipboard** or the Emacs function **x-get-clipboard**. The most likely explanation for apparent inconsistencies after **set-clipboard** is that the pasting application does not use the X clipboard.

In Cocoa applications (including LispWorks), the contents of the system clipboard is usually accessed by the user with the **Command+V** gesture.

Examples

To export an image:

```
(capi:set-clipboard pane nil nil (list :image image))
```

To export an image with a text description:

```
(capi:set-clipboard pane nil nil
                    (list :image image
                          :string "my image"))
```

See also

clipboard
selection
text-input-pane-copy
18.6 Clipboard

set-composition-placement *Function*

Summary

Specifies the placement of the composition window relative to the pane. Composition here mean composing input characters into other characters by an input method.

Package

capi

Signature

`set-composition-placement` *pane x y* **`&key`** *width height force*

Arguments

pane⇓ A pane.

x⇓, *y*⇓, *width*⇓, *height*⇓
Non-negative integers or **`nil`**.

force⇓ A generalized boolean.

Description

The function **`set-composition-placement`** tells the system where to place the composition window in pixel coordinates relative to the pane *pane*.

On systems where the composition text is displayed by the application (rather than by the system, when the composition callback is called with a plist), the placement coordinates are used to place the composition menu when it is raised.

x and *y* are the top left coordinates. If both *width* and *height* are supplied, they specify the dimensions of the composition window. If *force* is supplied with a true value, the coordinates are forced, overriding adjustments that the system may otherwise do.

x, *y* and, when supplied, *width* and *height* must all be positive integers.

Notes

`set-composition-placement` does not raise the composition window. It merely tells the system where to place the composition window when it does appear.

See also

`output-pane`
`output-pane-stop-composition`
12.2.4 Composition of characters

set-confirm-quit-flag *Function*

Summary

Controls the behavior of **`confirm-quit`**.

Package

`capi`

Signature

`set-confirm-quit-flag` *flag*

Arguments

flag⇓ One of **`t`**, **`nil`** or **`:check-editor-files`**.

Description

The function **`set-confirm-quit-flag`** sets a flag which controls the behavior of **`confirm-quit`**.

See **`confirm-quit`** for the meaning of *flag*.

Note: on initialization, the LispWorks IDE sets the flag to the stored value of the option **Tools > Preferences... > Environment > General > Confirm Before Exiting**.

See also

`confirm-quit`

set-default-editor-pane-blink-rate *Function*

Summary

Sets the default cursor blinking rate for editor panes.

Package

`capi`

Signature

`set-default-editor-pane-blink-rate` *blink-rate*

Arguments

blink-rate⇓ A non-negative real number, or **`nil`**.

Description

The function **`set-default-editor-pane-blink-rate`** sets the default to use for the editor pane cursor blinking rate. This default value is used when **`editor-pane-blink-rate`** returns **`nil`**.

Initially the setting is if this call has been made:

```
(set-default-editor-pane-blink-rate nil)
```

This means that the native blink rate will be used.

The argument *blink-rate* is interpreted as a blinking rate as described in **`editor-pane-blink-rate`**.

See also

`editor-pane-blink-rate`
`editor-pane-native-blink-rate`

set-default-interface-prefix-suffix *Function*

Summary

Sets the default suffix and prefix that are added to each interface title.

Package

capi

Signature

set-default-interface-prefix-suffix &key *prefix suffix child-prefix child-suffix* **=>** *prefix*, *suffix*, *child-prefix*, *child-suffix*

Arguments

prefix⇓	A string or **nil**.
suffix⇓	A string or **nil**.
child-prefix⇓	A string or **nil**.
child-suffix⇓	A string or **nil**.

Values

prefix	A string or **nil**.
suffix	A string or **nil**.
child-prefix	A string or **nil**.
child-suffix	A string or **nil**.

Description

The function **set-default-interface-prefix-suffix** sets the global default suffix and prefix that are added to each **interface** title. The prefix and suffix are added by the default method of **interface-extend-title**.

If *prefix*, *suffix*, *child-prefix* or *child-suffix* are supplied, their value must be either a string or **nil**. If any of them is not passed, the corresponding previously set value is not changed.

prefix and *suffix* specify the prefix and suffix to use for interfaces that are children of a **screen** object. These values do not affect *child-prefix* and *child-suffix*.

child-prefix and *child-suffix* specify the prefix and suffix to use for interfaces that are not children of a **screen** object, such as an interface inside a Multiple Document Interface (MDI) window. These values do not affect *prefix* and *suffix*.

The return values are the settings of the prefix, suffix, child prefix and child suffix after the call.

To check the current settings, call **set-default-interface-prefix-suffix** with no arguments. This does not change the current settings.

Before setting the title on a window on the screen, the system calls **interface-extend-title** with the interface and the title of the interface, and uses the result for the actual title. The default method of **interface-extend-title** checks *prefix* and *suffix* (or *child-prefix* and *child-suffix* for MDI) as were set by **set-default-interface-prefix-suffix**, and if they are non-nil adds the value to the title.

set-default-interface-prefix-suffix can be called after some windows are displayed. It automatically updates all current interface windows as if by calling **update-all-interface-titles**.

Examples

If you work in an environment when it is not always obvious on which machine your image is running, you can add the name of the machine to all windows by:

```
(capi:set-default-interface-prefix-suffix
        :suffix (format nil "-- ~a" (machine-instance)))
```

See also

interface-extend-title
update-all-interface-titles
3.3.2.1 Window titles
11.5 Controlling the appearance of the top level window

set-default-use-native-input-method *Function*

Summary

Controls the default of using native input method on GTK+.

Package

capi

Signature

set-default-use-native-input-method &key *output-pane editor-pane* **=> t**

Arguments

output-pane⇓	A boolean.
editor-pane⇓	A boolean.

Description

The function **set-default-use-native-input-method** controls whether the native input method is used by default. Currently it has an effect only on GTK+.

The values of the keyword arguments are booleans. *editor-pane* changes the default for **editor-pane** and subclasses. *output-pane* controls the default for **output-pane** and subclasses, except **editor-pane** and its subclasses.

If a keyword argument is not supplied, the corresponding default is not set.

See also

output-pane
editor-pane

12.2.3 Native input method

set-display-pane-selection *Generic Function*

Summary

Sets the selection in a **display-pane**.

Package

capi

Signature

set-display-pane-selection *pane start end*

Arguments

pane⇓	A **display-pane**.
start⇓, *end*⇓	Bounding indexes for a subsequence of the text of *pane*.

Description

The generic function **set-display-pane-selection** sets the selection in *pane* to be the text bounded by the indexes *start* (inclusive) and *end* (exclusive).

See also

display-pane-selection
display-pane

set-drop-object-supported-formats *Function*

Summary

Sets the list of formats for a drop object.

Package

capi

Signature

set-drop-object-supported-formats *drop-object formats*

Arguments

drop-object⇓	A *drop-object*, as passed to the *drop-callback*.
formats⇓	A list of format keywords.

Description

The function **`set-drop-object-supported-formats`** sets the list of formats that the drop object *drop-object* wants to receive.

The format **`:string`** can be used to receive a string from another application and the **`:filename-list`** format can be used to receive a list of filenames from another application such as the Macintosh Finder or the Windows Explorer.

GTK+ supports dragging of list of URIs. LispWorks uses a list of URIs to pass/receive the data with the format **`:filename-list`**, and also adds the format **`:uris`**. The behavior is as follows:

- For dragging with format **`:filename-list`** (that is, call **`drag-pane-object`** with a plist containing **`:filename-list`**, or including **`:filename-list`** in the value that *drag-callback* returns) the argument must be a list of pathname designators. LispWorks canonicalizes the pathnames and converts them to file URIs.
- For dragging with format **`:uris`**, each value in the list must either a string containing a colon, or a pathname designator. A string containing a colon is passed unchanged. Other it is assumed to be a pathname designator, and is converted to a file URI.
- For dropping with format **`:filename-list`** (that is, calling **`drop-object-get-object`** with **`:filename-list`**), LispWorks converts each file URI to the corresponding filename string (without checking whether it is a proper file name), and discards all other URIs.
- For dropping with format **`:uris`**, LispWorks returns all the URIs as strings.

There is an example of **`:filename-list`** and **`:uris`** here:

```
(example-edit-file "capi/elements/gtk-filename-list-and-uris")
```

On Cocoa and GTK+ the **`:image`** format can be used to receive images. The value passed needs to be an **`image`** obtect.

Any other keyword in *formats* is assumed to be a private format that can only be used to receive objects from with the same Lisp image.

Notes

`set-drop-object-supported-formats` should only be called within a *drop-callback*. See **`simple-pane`** for information about drop callbacks.

Examples

```
(example-edit-file "capi/output-panes/drag-and-drop")
```

```
(example-edit-file "capi/choice/drag-and-drop")
```

```
(example-edit-file "capi/choice/list-panel-drag-images")
```

See also

`drop-object-provides-format`
`simple-pane`
17 Drag and Drop

set-editor-parenthesis-colors *Function*

Summary

Sets the colors that are used for parenthesis coloring.

Package

`capi`

Signature

`set-editor-parenthesis-colors` *colors* **`&key`** *dark-background-colors*

Arguments

colors⇓ A list of colors, **`t`** or **`nil`**.

dark-background-colors⇓
A list of colors or **`nil`**.

Description

The function **`set-editor-parenthesis-colors`** sets the colors that are used for parenthesis coloring in an **`editor-pane`** in Lisp mode.

If *colors* is a non-nil list, each of its elements must be a valid color specification or a defined color alias. See **15 The Color System** for information about color specifications and aliases.

If it is called when CAPI is running, **`set-editor-parenthesis-colors`** checks that the colors are valid. If it is called when CAPI is not running, **`set-editor-parenthesis-colors`** does not check the colors, and a bad color will cause an error later. The colors have an effect only on coloring that happens after the call.

The colors in *colors* are used when the background is light. When the background is dark, a different set of colors is used. This set can be changed by supplying *dark-background-colors*, which should be a list colors. Each color in *dark-background-colors* is paired to a corresponding color in light-background colors (*colors* if it is a non-nil list, or the current list of *colors* is **`nil`** or **`t`**). If there are fewer colors in *dark-background-colors* than in the light-background colors, LispWorks pairs the rest of the light-background color with random light colors. If there are too many colors in *dark-background-colors*, the excess ones are ignored.

If *colors* is **`t`** or **`nil`**, parenthesis coloring is switched on or off, without changing the list of colors.

When parenthesis coloring is off, parentheses are drawn like other characters.

See also

`editor-pane`

set-geometric-hint *Function*

Summary

Sets a hint.

Package

capi

Signature

set-geometric-hint *element key value* **&optional** *override*

Arguments

element⇓	A **simple-pane** or a **pinboard-object**.
key⇓	A geometric hint keyword.
value⇓	A Lisp object.
override⇓	A boolean.

Description

The function **set-geometric-hint** sets the hint associated with *key* to *value* in *element*.

If *override* is **nil**, the value is not changed when there is already a hint for this key. The default is **t**.

See also

set-hint-table
element

set-hint-table *Function*

Summary

Modifies the hint table for an element.

Package

capi

Signature

set-hint-table *element plist*

Arguments

element⇓	A **simple-pane** or a **pinboard-object**.

plist⇓ A plist.

Description

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

This may or may not change the on-screen geometry. To change the geometry of an interface, use **set-top-level-interface-geometry**.

Notes

If a hint keyword is repeated in *plist*, the first value is used.

See also

element
set-geometric-hint
set-top-level-interface-geometry
6 Laying Out CAPI Panes
7 Programming with CAPI Windows

set-horizontal-scroll-parameters
set-vertical-scroll-parameters *Functions*

Summary

Allows programmatic control of the parameters of a horizontal or vertical scroll bar.

Package

capi

Signatures

set-horizontal-scroll-parameters *self* **&key** *min-range max-range slug-position slug-size page-size step-size*

set-vertical-scroll-parameters *self* **&key** *min-range max-range slug-position slug-size page-size step-size*

Arguments

self⇓ A displayed **output-pane** or **layout**.

min-range⇓, *max-range*⇓, *slug-position*⇓, *slug-size*⇓, *page-size*⇓, *step-size*⇓
Reals or **nil**.

Description

The functions **set-horizontal-scroll-parameters** and **set-vertical-scroll-parameters** set the specified parameters of the horizontal or vertical scroll bar of *self*.

self should be a displayed instance of a subclass of **output-pane** (such as **editor-pane**) or **layout** and have a scroll bar.

The other arguments are:

min-range The minimum data coordinate.

max-range The maximum data coordinate.

slug-position The current scroll position.

slug-size The length of the scroll bar slug.

page-size The scroll page size.

step-size The scroll step size.

When one of these keyword arguments is not supplied, the value of the corresponding scroll parameter in *self* is not modified.

See **7.4.2 Scroll values and initialization keywords** for a description of these scroll parameters.

Examples

```
(example-edit-file "capi/output-panes/fixed-origin-scrolling")
```

```
(example-edit-file "capi/output-panes/scrolling-without-bar")
```

```
(example-edit-file "capi/output-panes/coordinate-origin-fixed")
```

See also

`scroll`
`get-horizontal-scroll-parameters`
`get-vertical-scroll-parameters`
`simple-pane`
7 Programming with CAPI Windows
12.4 output-pane scrolling
7.4.2 Scroll values and initialization keywords

set-interactive-break-gestures *Function*

Summary

Sets the break gestures on GTK+ and Motif.

Package

`capi`

Signature

`set-interactive-break-gestures` *gestures* **`=>`** *result*

Arguments

gestures⇓ A list of gesture specifiers, or **`t`**.

Values

result A list.

Description

The function **set-interactive-break-gestures** sets the gestures that can be used to break by typing at an interface.

gestures is a list of gesture specifiers. A gesture specifier is an object that **sys:coerce-to-gesture-spec** can recognize.

When an interface is created, the break gestures are set such that typing any one of them when the interface is on top causes an "interface break". This means that, if the interface process is busy, it tries to break it. In a Listener tool, it tries to break the REPL. Otherwise it tries to find a process that appears busy, and breaks that. In the LispWorks IDE, if there is no busy process it raises the Process Browser tool. Otherwise it breaks the current process.

set-interactive-break-gestures always returns the list of interactive break gestures.

gestures can also be **t**, which means do not change the gestures. This is useful to get the current list.

Notes

1. **set-interactive-break-gestures** has an effect only on GTK+ and Motif.
2. **set-interactive-break-gestures** has no effect on interfaces that are already created.
3. On GTK+ the list can be overridden by the resources file as illustrated in **examples/gtk/gtkrc-break-gestures**

set-interface-pane-name-appearance
set-interface-pane-type-appearance *Functions*

Summary

Set the appearance (foreground, background, font) of panes inside interfaces of a specific type.

Package

capi

Signatures

set-interface-pane-name-appearance *interface-type pane-name* **&key** *font background foreground check-types*

set-interface-pane-type-appearance *interface-type pane-type* **&key** *font background foreground check-types*

Arguments

interface-type⇓ A symbol naming a subtype of **interface**.

pane-name⇓ Any object.

font⇓ A font specification as in **simple-pane**, or **nil** or **:default**, or a function or an fboundp symbol.

background⇓, *foreground*⇓

Color specifications as in **simple-pane**, or **nil** or **:default**, or a function or an fboundp symbol.

check-types⇓ A generalized boolean.

pane-type⇓ A symbol naming a subtype of **simple-pane**.

Description

The function **set-interface-pane-name-appearance** creates a setting such that, when a pane whose **capi-object** *name* is *pane-name* is created inside an interface of type *interface-type*, the pane's font, foreground and background attributes are set to *font*, *foreground* and *background* respectively.

If *font*, *foreground* or *background* is a function or an fboundp symbol, the value to use is the result of calling the function with two arguments: the interface and the pane.

If *font*, *foreground* or *background* is **nil** then the corresponding attribute is set to what it would have been set if **set-interface-pane-name-appearance** was not called at all for this *interface-type* and *pane-name*. See below for the meaning of **:default**.

The function **set-interface-pane-type-appearance** behaves the same as **set-interface-pane-name-appearance**, but the setting is applied to any pane of type *pane-type*.

Each call to **set-interface-pane-name-appearance** with a specific *interface-type* and *pane-name*, or to **set-interface-pane-type-appearance** with a specific *interface-type* and *pane-type*, completely overrides previous calls with the same *interface-type* and *pane-type* or *pane-name*.

When a pane (whose type is a subtype of **simple-pane**) is created (which happens when the interface is displayed by **display**), the settings that were created by **set-interface-pane-type-appearance** and **set-interface-pane-name-appearance** are applied, and override any other settings.

When more than one setting created by **set-interface-pane-type-appearance** or **set-interface-pane-name-appearance** is applicable to a pane, settings created by **set-interface-pane-name-appearance** take precedence over settings created by **set-interface-pane-type-appearance**, and otherwise the more specific settings, according to *interface-type* and *pane-type*, take precedence. The value for each attribute is specified by the setting with the highest precedence where the value is not **nil**.

If the value for an attribute in the highest precedence settings is **:default**, then settings of this attribute of lower precedence are ignored, and the attribute is set to what it would have been set to if none of the settings where created. Setting this for one attribute has no effect on the other attributes.

check-types, which defaults to **t**, controls whether the functions check if *interface-type* is a subtype of **interface**, and if *pane-type* is a subtype of **simple-pane**. Using **:check-types nil** allows you to use these functions before *interface-type* or *pane-type* are defined, at the price of no error checking.

Notes

The settings override any defaults for the matching panes and changes to the **simple-pane** *background*, *foreground* or *font* before the creation of the pane. They can be overridden after the pane is created, for example in a method on **interface-display**.

You can use these functions to customize the LispWorks IDE. For example in the IDE, the type of the interface of the Editor tool is **lw-tools:editor**, and this is also the name of the editor pane inside (but not of the collector-pane or echo-area pane). So you can customize the background of all the Editors in the IDE to red by:

```
(set-interface-pane-name-appearance
   'lw-tools:editor 'lw-tools:editor
   :background :red)
```

Note that this will not affect the pane in the "Output" tab and the echo area. You can use instead:

```
(set-interface-pane-type-appearance
    'lw-tools:editor 'capi:editor-pane
    :background :red)
```

The latter call affects the output and echo-area panes too, because they are subclasses of **editor-pane**. This will override the preferences that are set by the Preferences Dialog in the IDE.

You can use **interface** as *interface-type* to make it applicable to all interfaces, but that may cause undesired effects if it applies to unintended panes. There is also a little overhead associated with settings, though this is probably negligible unless large number of settings are created.

set-interface-pane-name-appearance and **set-interface-pane-type-appearance** will typically be used in your .lispworks initialization file. They can also be useful for adding customization to your application.

See also

simple-pane
15 The Color System
13.9 Portable font descriptions
18.8 Setting the font and colors for specific panes in specific interfaces.

set-list-panel-keyboard-search-reset-time *Function*

Summary

Sets the default length of time before resetting the "last match" in keyboard searching in a **list-panel**.

Package

capi

Signature

set-list-panel-keyboard-search-reset-time *time*

Arguments

time⇓ A positive real number.

Description

The function **set-list-panel-keyboard-search-reset-time** sets the default length of time before resetting the "last match" in keyboard searching in a **list-panel**. The argument *time* specifies this time in seconds.

When the user types a character into a **list-panel**, if there is a "last match" the system searches for a string made of the "last match" followed by the character, otherwise it searches for a string made of the character only. The system sets the "last match" when it matches, and remembers the "last match" for one second by default.
set-list-panel-keyboard-search-reset-time can be used to change the time for which the "last match" is kept.

Notes

When *keyboard-search-callback* returns a third value non-nil, the value that

set-list-panel-keyboard-search-reset-time sets is ignored.

See also

list-panel
list-panel-search-with-function
5.3.9 Searching by keyboard input

set-object-automatic-resize *Function*

Summary

Controls automatic resizing and repositioning of objects in a static layout.

Package

capi

Signature

set-object-automatic-resize *object* **&key** *x-align y-align x-offset y-offset x-ratio y-ratio width-ratio height-ratio aspect-ratio aspect-ratio-y-weight pinboard*

Arguments

object⇓ A **pinboard-object** or a **simple-pane**.

x-align⇓ **nil**, **:left**, **:center** or **:right**.

y-align⇓ **nil**, **:top**, **:center** or **:bottom**.

x-offset⇓ A real number, default value 0.

y-offset⇓ A real number, default value 0.

x-ratio⇓ A positive real number or **nil**.

y-ratio⇓ A positive real number or **nil**.

width-ratio⇓ A positive real number or **nil**.

height-ratio⇓ A positive real number or **nil**.

aspect-ratio⇓ A positive real number, **t** or **nil**.

aspect-ratio-y-weight⇓

A real number, default value 0.5.

pinboard⇓ A **static-layout**, if supplied. This argument is deprecated, and can always be omitted.

Description

The function **set-object-automatic-resize** arranges for *object* to be resized and/or re-positioned automatically when *pinboard* is resized, or removes such a setting.

The value of *aspect-ratio* can be **t**, which means use the current aspect ratio of *object* (that is, its height divided by its width).

object should be either a **pinboard-object** or a **simple-pane** which is (or will be) displayed in a **static-layout**. This *object* will be added to the *description* of the layout by one of its **:description** initarg,

`(setf capi:layout-description)` or **`manipulate-pinboard`**.

pinboard is the layout for *object*. If *pinboard* is already displayed with *object* in its *description*, the argument *pinboard* can be omitted.

When *pinboard* is resized, *object* is resized if either *height-ratio* or *width-ratio* are set.

The new width of *object* is calculated as follows:

- If *width-ratio*, *height-ratio* and *aspect-ratio* are all set, the new width is the width of *pinboard* multiplied by *width-ratio*, and then modified as described below.
- If *width-ratio* is set and either *height-ratio* or *aspect-ratio* is not set, the new width is the width of *pinboard* multiplied by *width-ratio*.
- If *width-ratio* is not set, and both *height-ratio* and *aspect-ratio* are set, the new width is the new height divided by *aspect-ratio*.
- Otherwise, the new width is the same as the old width.

The new height of *object* is calculated as follows:

- If *width-ratio* and *aspect-ratio* are set, the new height is the new width multiplied by the aspect ratio. Note that if *height-ratio* is set, the new width will depend on *height-ratio* too.
- If *height-ratio* is set and either *width-ratio* or *aspect-ratio* are not set, the new height is the height of *pinboard* multiplied by *height-ratio*.
- If *height-ratio* is not set, but both *width-ratio* and *aspect-ratio* are set, the new height is the new width multiplied by *aspect-ratio*.
- Otherwise, the new height is the same as the old height.

If all of *width-ratio*, *height-ratio* and *aspect-ratio* are set, the new width and height of object are calculated as follows:

1. Compute *calculated-width* as the width of *pinboard* multiplied by *width-ratio*, and *calculated-height* as the height of *pinboard* multiplied by *height-ratio*.
2. Compute *aspect-ratio-ratio* as:

   ```
   (/ (/ calculated-height calculated-width) aspect-ratio)
   ```

3. Compute *correction* as:

   ```
   (expt aspect-ratio-ratio aspect-ratio-y-weight)
   ```

4. Compute the new width as *calculated-width* multiplied by *correction*, and the new height as the new width multiplied by *aspect-ratio*.

The result is that if *aspect-ratio-y-weight* is 0, *correction* is 1 and *height-ratio* is effectively ignored, while if *aspect-ratio-y-weight* is 1, *correction* cancels the effect of *width-ratio*. With the default value of 0.5, the resulting position is in the (geometric) middle, and *object* takes a fixed fraction of the area of the pinboard.

After resizing (if needed), *object* is also positioned horizontally if *x-align* is non-nil, and vertically if *y-align* is non-nil.

The new x coordinate of *object* is calculated as follows:

- If *x-ratio* is set, the new x coordinate is the sum of *x-ratio* multiplied by the width of *pinboard* plus *x-offset*, otherwise it is simply *x-offset*.

- The actual value of the x coordinate for *object* is adjusted according to the value of *x-align* such that the left, center or right of *object* align with the new coordinate.

The new y coordinate of object is calculated similarly, using *y-ratio* and *y-offset*, with an adjustment such that the top, center or bottom of *object* aligns with the new coordinate according to *y-align*.

If all of *width-ratio*, *height-ratio*, *x-align* and *y-align* are **nil**, automatic resizing/re-positioning of *object* is removed.

set-object-automatic-resize can be called before *object* is actually displayed, and its effect persists over calls adding and removing *object* to/from **static-layout**s. The effect of **set-object-automatic-resize** also persists if *object* is removed and added again, either to the same layout or another layout.

Repeated calls to **set-object-automatic-resize** set only the values that are passed to **set-object-automatic-resize**. Keys that are not passed are left with their previous value. A call that removes the automatic resizing (because *width-ratio*, *height-ratio*, *x-align* and *y-align* are all **nil**) erases all the values.

set-object-automatic-resize returns **t** if the object is set up for automatic resizing, or **nil** if the object is set up for no automatic resizing.

Notes

1. The initarg **:automatic-resize** can be used to set up automatic resizing in the call to **make-instance**.
2. The name **set-object-automatic-resize** is slightly inaccurate, because this function can alter an object's position without actually changing its size.

Compatibility note

In LispWorks 6.0 the effect of **set-object-automatic-resize** does not persist if the object is removed and then added, to any layout.

In LispWorks 6.0 each call to **set-object-automatic-resize** sets all the values.

Examples

Put an object of fixed size at the top right corner:

```
(set-object-automatic-resize object
                             :x-ratio 1 :x-align :right)
```

Put an object in the bottom-right quadrant:

```
(set-object-automatic-resize
 object
 :x-ratio 0.5 :y-ratio 0.5
 :width-ratio 0.5 :height-ratio 0.5)
```

Put an object with a fixed aspect ratio and object width linear with the width of the layout in the center:

```
(set-object-automatic-resize
 object
 :x-align :center :y-align :center
 :x-ratio 0.5 :y-ratio 0.5
 :aspect-ratio 0.6 :width-ratio 0.1)
```

There is a further example in:

```
(example-edit-file "capi/layouts/automatic-resize")
```

See also

manipulate-pinboard
static-layout
pinboard-object
simple-pane

set-pane-focus *Generic Function*

Summary

Sets the input focus to a pane.

Package

capi

Signature

set-pane-focus *pane*

Arguments

pane⇓ An instance of a subclass of **simple-pane** or **choice**.

Description

The generic function **set-pane-focus** sets the input focus to *pane* or one of its children.

See also

pane-has-focus-p
3.1.5 Focus

set-printer-metrics *Function*

Summary

Sets the metrics in the given printer.

Package

capi

Signature

set-printer-metrics *printer* **&key** *left-margin top-margin width height*

Arguments

printer⇓ A printer.

left-margin⇓ A real or **nil**.

top-margin⇓ A real or **nil**.

width⇓ A real or **nil**.

height⇓ A real or **nil**.

Description

The function **set-printer-metrics** sets the left margin and top margin, and the printable width and printable height, of *printer* to *left-margin*, *top-margin*, *width* and *height* respectively. Values outside the bounds of the printer will be corrected and values that are **nil** cause no change to the corresponding setting.

Examples

To set the margins as large as possible:

```
(let ((metrics (capi:get-printer-metrics printer)))
  (capi:set-printer-metrics printer
     :left-margin 0
     :top-margin 0
     :width
     (capi:printer-metrics-paper-width metrics)
     :height
     (capi:printer-metrics-paper-height metrics)))
```

Actually this sets the margins to the whole paper size, but the printer driver will move these in to take account of the minimum margins of the device.

See also

get-printer-metrics
set-printer-options
print-dialog
16 Printing from the CAPI—the Hardcopy API

set-printer-options *Function*

Summary

Sets various options in the given printer.

Package

capi

Signature

set-printer-options *printer* **&key** *output-file* *first-page* *last-page* *orientation* *copies*

Arguments

printer⇓ A printer.

output-file⇓	A pathname designator or **`nil`**.
first-page⇓	A positive integer or **`nil`**.
last-page⇓	A positive integer or **`nil`**.
orientation⇓	One of **`:landscape`**, **`:portrait`** or **`nil`**.
copies⇓	A positive integer or **`nil`**.

Description

The function **`set-printer-options`** allows some printer options for the current job to be set programmatically. Note that the user can change the various printer options in the dialog displayed by **`print-dialog`**.

printer should be a printer object returned by **`current-printer`** or **`print-dialog`**. *printer* should then be passed to **`with-print-job`** to print using the options specified.

The keyword arguments control which options are set. If a keyword is not passed then the option remains unchanged.

Values of *output-file* are:

`nil`	Print directly to the device.
`t`	Print to a file chosen by the user at printing time.
A pathname	Print to the file given by pathname.

Values of *first-page* are:

`:all`	Print all pages.
An integer	Print from this page to the page given by *last-page*.

Values of *orientation* are:

`:landscape`	Print in landscape mode.
`:portrait`	Print in portrait mode.

Values of *copies*:

An integer	The number of copies to print.

Notes

Printer objects cannot be reused after changing their options or metrics. Call **`current-printer`** after **`set-printer-options`** to get a new printer object containing the latest settings.

Examples

```
;; Print two copies to the current printer.
(let ((printer (capi:current-printer)))
  (capi:set-printer-options printer :copies 2)
  (capi:with-print-job (port :printer printer)
    (print-my-document port)))
```

See also

`print-dialog`

current-printer
with-print-job
16 Printing from the CAPI—the Hardcopy API

set-rich-text-pane-character-format *Function*

Summary

Sets the character format.

Package

capi

Signature

set-rich-text-pane-character-format *pane* **&key** *selection* *attributes-plist* **=>** *result*

Arguments

pane⇓	A **rich-text-pane**.
selection⇓	Must be **t**. This argument is deprecated.
attributes-plist⇓	A plist or **:default**.

Values

result	A plist.

Description

The function **set-rich-text-pane-character-format** sets current character attributes for text in *pane*.

If there is a current selection in the pane, then the attributes are set for the selected text. If there is no selection, then it sets the "typing attributes", which are applied to characters that are typed by the user. Note that any cursor movement changes these attributes, so the setting is ephemeral.

Supplying *selection* is deprecated. If *selection* is **nil** an error is signalled. The default value of *selection* is **t**.

If *attributes-plist* is the symbol **:default** then the default character format of the pane (that is, the value of the **rich-text-pane** initarg **:character-format**) is used. Otherwise *attributes-plist* is a plist of keywords and values. These are the valid keywords on Microsoft Windows and Cocoa:

:bold	A boolean.
:italic	A boolean.
:underline	A boolean.
:face	A string naming a font.
:color	A color spec or alias specifying the foreground color.
:size	The size of the font.

Additionally these *attributes-plist* keywords are valid on Microsoft Windows only:

:strikeout A boolean.

:offset An integer specifying the vertical offset of characters from the line (a positive value makes them superscript and a negative value makes them subscript).

:protected A boolean. See the description of *protected-callback* in **rich-text-pane**.

:charset A cons **(***charset* **.** *pitch-and-family***)** where *charset* has the value of a Microsoft Windows charset identifier, and *pitch-and-family* is the value of **(logior** *pitch family***)** where pitch and family have the value of a Windows pitch and a Windows font family respectively.

Compatibility note

The value **nil** for the keyword argument **:selection** is not supported in LispWorks 6.1 and later. See the description above for details of the current behavior with respect to the current selection in the **rich-text-pane**.

Examples

Note: This example uses some features which are supported only on Microsoft Windows:

```
(defun ok-to-edit-p (pane start end s)
  (declare (ignore pane))
  (capi:prompt-for-confirmation
   (format nil "Editing~:[ ~; selection ~]from ~a to ~a"
           s start end)))

(setq rtp
      (capi:contain
       (make-instance
        'capi:rich-text-pane
        :protected-callback 'ok-to-edit-p
        :character-format
        '(:size 14  :color :red)
        :visible-min-height 300
        :visible-min-width 400
        :paragraph-format
        '(:start-indent 20 :offset -15)
        :text-limit 160
        :text (format nil "First paragraph.~%Second paragraph, a little longer.~%Another paragraph,
 which should be long long enough that it spans more than one line. ~%" ))))
```

Enter some characters in the rich text window and select a range.

Set the selection to blue:

```
(capi:set-rich-text-pane-character-format
 rtp
 :attributes-plist '(:color :blue))
```

Make it protected:

```
(capi:set-rich-text-pane-character-format
 rtp :attributes-plist '(:protected t))
```

Now try to delete a character, and also to delete the selection. In both cases the **ok-to-edit-p** callback is called.

See also

rich-text-pane

rich-text-pane-character-format

set-rich-text-pane-paragraph-format *Function*

Summary

Sets the paragraph format.

Package

capi

Signature

set-rich-text-pane-paragraph-format *pane attributes-plist* **=>** *result*

Arguments

pane⇓	A **rich-text-pane**.
attributes-plist⇓	A plist, or **:default**.

Values

result	A plist.

Description

The function **set-rich-text-pane-paragraph-format** sets paragraph attributes for the current paragraphs in *pane*.

The current paragraphs are those paragraphs which overlap the current selection, or the paragraph containing the insertion point if there is no selection.

If *attributes-plist* is the symbol **:default** then the default paragraph format of *pane* is used. Otherwise *attributes-plist* is a plist of keywords and values. These are the valid keywords on Microsoft Windows and Cocoa:

:alignment	**:left**, **:right** or **:center**.
:start-indent	A number setting the indentation.
:offset-indent	A number modifying the indentation.
:offset	A number setting the relative indentation of subsequent lines in a paragraph.
:right-indent	A number setting the right margin.
:tab-stops	A list of numbers.

Additionally this *attributes-list* keyword is valid on Microsoft Windows, only:

:numbering	**nil**, **t**, **:bullet**, **:arabic**, **:lowercase**, **:uppercase**, **:lower-roman** or **:upper-roman**.

numbering specifies the numbering style. Rich Edit 3.0 supports all the above values of *numbering*. Please note that the Arabic and Roman styles start numbering from zero, and that only **t** and **:bullet** work with versions of Rich Edit before 3.0 (other values of *numbering* are quietly ignored).

start-indent specifies the indentation of the first line of a paragraph. A negative value removes the indentation.

offset-indent takes effect only when *start-indent* is not passed. It specifies an increase in the current indentation. Therefore, a negative value of *offset-indent* decreases the indentation.

offset specifies the offset of the second and following lines relative to the first line of the paragraph. That is, when the indentation of the first line is *indent*, the indentation of the second and subsequent lines is *indent* + *offset*. When *offset* is negative, the second and subsequent lines are indented less than the first line. If *indent* + *offset* is negative, then these lines are not indented.

tab-stops should be a list of numbers specifying the locations of tabs. No more than 32 tabs are allowed.

Examples

```
(setq rtp
      (capi:contain
       (make-instance
        'capi:rich-text-pane
        :visible-min-height 300
        :visible-min-width 400
        :paragraph-format
        '(:start-indent 20 :offset -15)
        :text (format nil "First paragraph.~%Second paragraph, a little longer.~%Another paragraph,
 which should be long long enough that it spans more than one line. ~%" ))))

(capi:set-rich-text-pane-paragraph-format
 rtp '(:offset-indent 30 :numbering :lowercase))
```

See also

rich-text-pane
rich-text-pane-paragraph-format

set-selection *Function*

Summary

Sets the primary selection.

Package

capi

Signature

set-selection *self value* **&optional** *string plist* **=>** *result*

Arguments

self⇓	A displayed CAPI pane or interface.
value⇓	A Lisp object (not necessarily a string) to make available within the local Lisp image.
string⇓	The string representation of *value* to export, or **nil**.
plist⇓	A property list of additional format/value pairs to export.

Values

result A string, or **nil**.

Description

The function **set-selection** sets the primary selection associated with *self*.

If *string* is non-nil, then the text of the primary selection is set to *string*. If *string* is **nil** and *value* is a string, then text of the primary selection is set to *value*. Otherwise, no text is set for the primary selection.

In addition, *value* is made available within the local Lisp image when calling **selection**.

plist is a plist of additional format/value pairs to export to the primary selection. The currently supported formats are as described for **selection**. You can export more than one format simultaneously.

On Microsoft Windows there is no notion of selection, so this mechanism is internal to Lisp.

Note that X applications may or may not use the primary selection for their paste operations. The most likely explanation for apparent inconsistencies after **set-selection** is that the pasting application does not use the primary selection. For instance, Emacs is configurable by the variable interprogram-paste-function.

See also

selection
set-clipboard
18.6 Clipboard

set-text-input-pane-selection *Generic Function*

Summary

Sets the selection in a **text-input-pane**.

Package

capi

Signature

set-text-input-pane-selection *pane start end*

Arguments

pane⇓ A **text-input-pane**.

start⇓, *end*⇓ Bounding indexes for a subsequence of the text of *pane*.

Description

The generic function **set-text-input-pane-selection** sets the selection in *pane* to be the text bounded by the indexes *start* (inclusive) and *end* (exclusive).

See also

text-input-pane-selection
text-input-pane

set-top-level-interface-geometry *Generic Function*

Summary

Sets the geometry of a top level interface.

Package

capi

Signature

set-top-level-interface-geometry *interface* **&key** *x y width height*

Arguments

interface⇓ A CAPI interface.

x⇓, *y*⇓, *width*⇓, *height*⇓

Integers specifying the new geometry.

Description

The generic function **set-top-level-interface-geometry** sets the geometry of a top level interface.

The coordinates of *interface* are modified according to *x*, *y*, *width* and *height*. *interface* should be a top level interface. If a keyword is omitted then that part of the coordinates is not changed.

x and *y* are measured from the top-left of the screen rectangle representing the area of the primary monitor (the primary screen rectangle).

Notes

On Cocoa **set-top-level-interface-geometry** behaves as if an interface toolbar is not present, even if *interface* does contain an interface toolbar.

Examples

```
(setf ii
      (capi:element-interface
       (capi:contain
        (make-instance 'capi:text-input-pane))))

(multiple-value-bind (x y width height)
    (capi:top-level-interface-geometry ii)
  (capi:execute-with-interface
   ii
   'capi:set-top-level-interface-geometry
   ii
   :x (round (+ x (/ width 4)))
   :y y
```

```
    :width (round (* 0.75 width))
    :height height))
```

See also

top-level-interface-p
top-level-interface-geometry
top-level-interface-display-state
interface
7 Programming with CAPI Windows

shell-pane *Class*

Summary

A pane allowing the user to interact with a subprocess.

Package

capi

Superclasses

interactive-pane

Initargs

:command	The command which is run as a subprocess.

Accessors

shell-pane-command

Description

The class **shell-pane** creates an editor in which a subprocess runs.

User input is interpreted as input to the subprocess. In particular, when the user enters **Return** in the last line, the line is sent to the subprocess. The output of the subprocess is displayed in the pane.

The default value of command is **nil**, which means that the actual command is determined as follows:

On Microsoft Windows, **cmd.exe** is run.

On non-Windows platforms, the value of the environment variable **ESHELL** is used if set, and otherwise the environment variable **SHELL** is consulted. If that is not set, then **/bin/csh** (**/bin/sh** on SVR4 platforms) is run.

Examples

This function emulates user input on *pane*:

```
(defun send-keys-to-pane-aux (pane string newline-p)
  (loop for char across string
        do (capi:call-editor pane char))
  (if newline-p
```

```
        (capi:call-editor pane #\Return)))
```

This function trampolines to **send-keys-to-pane-aux** on the right process:

```
(defun send-keys-to-pane (pane string newline-p)
  (capi:apply-in-pane-process pane
                              'send-keys-to-pane-aux
                              pane string newline-p))

(setq sp (capi:contain
          (make-instance 'capi:shell-pane
                         :visible-min-width
                         '(character 60)
                         :visible-min-height
                         '(character 30))))
```

This call emulates the user typing **dir** followed by **Return**:

```
(send-keys-to-pane sp "dir" t)
```

show-interface *Function*

Summary

Brings the interface containing a specified pane onto the screen.

Package

capi

Signature

show-interface *pane*

Arguments

pane⇓ A pane.

Description

The function **show-interface** brings the interface containing *pane* back onto the screen.

To hide the interface use **hide-interface**.

See also

hide-interface
activate-pane
interface
7.7 Manipulating top-level windows

show-pane *Function*

Summary

Restores the specified pane to the screen.

Package

capi

Signature

show-pane *pane* **=>** *pane*

Arguments

pane⇓ An instance of **simple-pane** or a subclass.

Values

pane An instance of **simple-pane** or a subclass.

Description

The function **show-pane** restores the pane *pane* to the screen if it is hidden (for instance by **hide-pane**) or iconified.

See also

hide-pane
show-interface

simple-layout *Class*

Summary

A layout with a single child, and the child is resized to fill the space (where possible).

Package

capi

Superclasses

x-y-adjustable-layout

Subclasses

switchable-layout

Description

The class **simple-layout** is a layout with a single child, and the child is resized to fill the space (where possible).

The description of a **simple-layout** can be either a single child, or a list containing just one child. The simple layout then adopts the size constraints of its child, and lays the child out inside itself.

Examples

```
(capi:contain (make-instance
               'capi:simple-layout
               :description (list (make-instance
                                   'capi:text-input-pane))))
```

See also

layout
row-layout
column-layout

simple-network-pane *Class*

Summary

A graph pane which arranges its nodes in a grid.

Package

capi

Superclasses

graph-pane

Initargs

:x-gap	The horizontal node spacing.
:y-gap	The vertical node spacing.

Description

The class **simple-network-pane** provides a graph which lays out its nodes in a rectangular grid by a simple algorithm.

The default values of *x-gap* and *y-gap* are 200 and 100 respectively.

simple-network-pane is a subclass of **choice**, so for details of its selection handling, see **choice**.

Examples

```
(example-edit-file "capi/graphics/network")
```

simple-pane *Class*

Summary

The class **simple-pane** is the superclass for any elements that actually appear as a native window, and is itself an empty window.

Package

capi

Superclasses

element

Subclasses

display-pane
interface
title-pane
button-panel
list-panel
option-pane
output-pane
progress-bar
slider
text-input-pane
tree-view
toolbar
layout
button

Initargs

:enabled	A boolean controlling whether the pane is enabled.
:background	The background color of the pane.
:foreground	The foreground color of the pane.
:font	The default font for the pane.
:horizontal-scroll	**t**, **:without-bar**, or **nil**. If true the pane can scroll horizontally.
:vertical-scroll	**t**, **:without-bar**, or **nil**. If true the pane can scroll vertically.
:scroll-bar-type	**nil** (the default) or **:always-visible**.
:visible-border	A boolean or a keyword controlling whether the pane has a border, for some pane classes.
:internal-border	A non-negative integer, or **nil**. Controls the width of the internal border.
:cursor	A keyword naming a built-in cursor, or a cursor object, or **nil**.
:pane-menu	Specifies a menu to be raised by the **:post-menu** gesture.
:drop-callback	Specifies a drop callback for **output-pane**, **interface**, **list-panel** or **tree-view**.
:drag-callback	Specifies a drag callback for **list-panel** or **tree-view**.
:automatic-resize	A plist.
:scroll-if-not-visible-p	

Defines whether, when the focus is given to the pane and the pane is not fully visible, the pane's parent is automatically scrolled to show it.

:toolbar-title A string.

:scroll-horizontal-slug-size

Useful only for output-pane and subclasses and for layouts. See **set-horizontal-scroll-parameters**.

:scroll-vertical-slug-size

Useful only for output-pane and subclasses and for layouts. See **set-vertical-scroll-parameters**.

:scroll-start-x Useful only for output-pane and subclasses and for layouts. See **set-horizontal-scroll-parameters**.

:scroll-start-y Useful only for output-pane and subclasses and for layouts. See **set-vertical-scroll-parameters**.

:scroll-width Useful only for output-pane and subclasses and for layouts. See **set-horizontal-scroll-parameters**.

:scroll-height Useful only for output-pane and subclasses and for layouts. See **set-vertical-scroll-parameters**.

:scroll-initial-x Useful only for output-pane and subclasses and for layouts. See **set-horizontal-scroll-parameters**.

:scroll-initial-y Useful only for output-pane and subclasses and for layouts. See **set-vertical-scroll-parameters**.

:scroll-horizontal-step-size

Useful only for output-pane and subclasses and for layouts. See **set-horizontal-scroll-parameters**.

:scroll-vertical-step-size

Useful only for output-pane and subclasses and for layouts. See **set-vertical-scroll-parameters**.

:scroll-horizontal-page-size

Useful only for output-pane and subclasses and for layouts. See **set-horizontal-scroll-parameters**.

:scroll-vertical-page-size

Useful only for output-pane and subclasses and for layouts. See **set-vertical-scroll-parameters**.

Accessors

simple-pane-enabled
simple-pane-background
simple-pane-foreground
simple-pane-font
simple-pane-cursor
simple-pane-scroll-callback
simple-pane-drop-callback
simple-pane-drag-callback

Readers

simple-pane-horizontal-scroll
simple-pane-vertical-scroll

simple-pane-visible-border

Description

enabled determines whether the pane is enabled. The default value is **`t`**. Note that changing the enabled state of a visible pane by **`(setf simple-pane-enabled)`** changes its appearance.

background and *foreground* are colors specified using the Graphics Ports color system. Additionally on Cocoa, the special value **`:transparent`** is supported, which makes the pane's background match that of its parent. The keyword **`:background`** can also be used as the value for *background*, which is generally the same as not specifying *background* at all, except for **`layout`** panes where the initargs **`:background :background`** also forces the pane to have its own native GUI object. You need to do that if you want to make a **`layout`** without a background initially, and change it later using **`(setf simple-pane-background)`**.

font should be a **`font`**, a **font-description**, a font alias, or **`nil`**. If it is not a **`font`**, it is converted to a **`font`** when the pane is created. **`nil`** is converted to the default font, and a **font-description** is converted as if by calling **`find-best-font`**.

pane-menu can be used to specify or create a menu to be displayed when the **`:post-menu`** gesture is received by the pane. It has the default value **`:default`** which means that **`make-pane-popup-menu`** is called to create the menu. For a full description of *pane-menu*, see **8.12 Popup menus for panes**.

Notes

1. *foreground* is ignored for buttons on Windows and Cocoa.
2. On Microsoft Windows *pane-menu* is not supported for **`title-pane`**. See **`title-pane`** for alternative approaches.
3. The *font*, *foreground* and *background* might be overridden by settings created using **`set-interface-pane-name-appearance`** or **`set-interface-pane-type-appearance`**.

Description: Cursor

cursor specifies a cursor for the pane. On Cocoa and GTK+, the *cursor* initarg has an effect only in **`output-pane`** and its subclasses. On other platforms it changes the cursor for other CAPI pane classes, although this may contravene style guidelines.

`nil` means use the default cursor, and this is the default value. *cursor* can also be a cursor object as returned by **`load-cursor`**. The other allowed values are keywords naming built-in cursors which are supported on each platform as shown in the table below.

cursor	Cocoa	Windows	Motif
`:busy`	No	Yes	Yes
`:i-beam`	Yes	Yes	Yes
`:top-left-arrow`	Yes	Yes	Yes
`:h-double-arrow`	Yes	Yes	Yes
`:v-double-arrow`	Yes	Yes	Yes
`:left-side`	Yes	Yes	Yes
`:right-side`	Yes	Yes	Yes
`:top-side`	Yes	Yes	Yes
`:bottom-side`	Yes	Yes	Yes

:wait	No	Yes	Yes
:crosshair	Yes	Yes	Yes
:gc-notification	No	Yes	Yes
:top-left-corner	No	Yes	Yes
:top-right-corner	No	Yes	Yes
:bottom-left-corner	No	Yes	Yes
:bottom-right-corner	No	Yes	Yes
:hand	Yes	Yes	Yes
:fleur	Yes	Yes	Yes
:move	Yes	Yes	Yes
:closed-hand	Yes	No	No
:open-hand	Yes	No	No
:disappearing-item	Yes	No	No

Description: Drag and drop

drop-callback can be specified for a pane that is an instance of **output-pane**, **interface**, **list-panel**, **tree-view** or a subclass of one of these. When the user drags an object over a window, the CAPI first tries to call the *drop-callback* of any pane under the mouse and otherwise calls the *drop-callback* of the top-level interface. The default value of *drop-callback* is **nil**, which means that there is no support for dropping into the pane.

For **editor-pane**, *drop-callback* can be **:default**, which provides support for dropping a string into the pane and inserting the string into the pane's editor buffer.

If *drop-callback* is any other non-nil value, it should be either a list (for simple cases) or function designator (to use all options). When it is a function designator, it needs to have this signature:

drop-callback *pane drop-object stage*

The function *drop-callback* is called by the CAPI at various times such as when the pane is displayed and when the user attempts to drop data into the pane. *pane* is the pane itself, *drop-object* is an object used to communicate information about the current dropping operation (see below) and *stage* is a keyword. *drop-callback* should handle these values of *stage*:

:formats This might occur when the pane is being displayed or might occur each time the user drags or drops an object over the pane. It should call **set-drop-object-supported-formats** with the *drop-object* and a list of formats that the pane wants to receive. Each format is a keyword. The list of the formats must be the same each time it is called.

:enter This occurs when the user drags an object into a pane which is an **output-pane** or **interface** (but not for a pane which is a **list-panel** or **tree-view**). It can query the *drop-object* using **drop-object-provides-format** and **drop-object-allows-drop-effect-p** to discover what the user is dragging. It can also use **drop-object-pane-x** and **drop-object-pane-y** to query the mouse position relative to the pane. It should call **(setf drop-object-drop-effect)** with an effect if it wants to allow the object to be dropped. If this is not called, then the object cannot be dropped into the pane.

:leave This occurs when the user drags an object out of a pane which is an **output-pane** or **interface** (but not for a pane which is a **list-panel** or **tree-view**).

:drag This occurs while the user is dragging an object over the pane. It can query the *drop-object* using **drop-object-provides-format** and **drop-object-allows-drop-effect-p** to discover what the user is dragging. For **output-pane**, it can use **drop-object-pane-x** and **drop-object-pane-y** to query the mouse position relative to the pane. For **list-panel** and **tree-view**, it can use **drop-object-collection-index** or **drop-object-collection-item** to query where the user is attempting to drop the object and can call their setf functions to adjust this position. It should call **(setf drop-object-drop-effect)** with an effect if it wants to allow the object to be dropped. If this is not called, then the object cannot be dropped into the pane. For **output-pane** and **interface**, it might also want to update the pane to indicate where the object will be dropped.

:drop This occurs when the user drops an object over the pane. It can query the *drop-object* as for the **:drag** stage, but can also obtain the object itself using **drop-object-get-object** for one of the formats in the list returned by **drop-object-provides-format**. Once the object is received, it should call **(setf drop-object-drop-effect)** with the effect that has been used by the callback. It should also update the pane to incorporate the object in whatever way the application requires.

When *drop-callback* is a list, it specifies a simple response. The list should be of the form:

(*effects formats drop-stage-callback* **&optional** *checker***)**

Both *effects* and *formats* can be either a list of effects or formats, or an atom which is interpreted as a list of one element. *effects* and *formats* specify which effects and formats are allowed.

For the stages except **:formats**, the first effect of the given effects that the *drop-object* allows is set (by calling **(setf drop-object-drop-effect)**), except when *checker* is supplied. In the latter case, before setting an effect it loops through the formats and calls the checker with three arguments:

funcall *checker pane effect format*

If *checker* returns non-nil it sets the effect. If *checker* returns **nil** for the formats, it goes to the next effect.

In the **:drop** stage, after setting the effect, it gets the object with first format that is provided by the *drop-object*, and then calls the *drop-stage-callback* with four arguments:

funcall *drop-stage-callback pane object x-or-index y-or-placement*

If the pane is a **tree-view** or **list-panel**, the last two arguments are the item index (for **get-collection-item**) and placement (**:above**, **:item**, **:below**), which are the results of **drop-object-collection-index**. Otherwise, the last two arguments are the *x* and *y* (results of **drop-object-pane-x** and **drop-object-pane-y**). It is the responsibility of the *drop-stage-callback* to perform whatever dropping should mean.

drag-callback can be specified for a pane that is an instance of **list-panel** or **tree-view**. The default value of *drag-callback* is **nil**, which means that there is no support for dragging from the pane. Otherwise, it should be a function designator with this signature:

drag-callback *pane info* **=>** *result*

When the user drags items in the pane, the CAPI calls the *drag-callback*. *pane* is the pane itself and *info* is a list of item indices that are being dragged (compare with **choice-selection**).

The *drag-callback* should normally return a plist *result* whose keys are the data formats to be dragged, with a value associated with each format. Formats are arbitrary keywords that must be interpreted by the pane where you intend to drop the values (see the *drop-callback*). The format **`:string`** is understood by some other panes that expect text.

The plist *result* returned by *drag-callback* can contain the key **`:image-function`** with a function *image-function* as value.

This function is used to generate the image that is used in the dragging itself, exactly as the *image-function* in **`drag-pane-object`** is used. On Cocoa, **`tree-view`** and **`list-panel`** ignore this key in *result*.

drag-callback can also be used in top-level interfaces. In this case the second argument *info* is a flag describing the gesture that caused the call. Currently the only value is **`:drag-image`**, which means it was invoked by dragging the *drag-image* (see **`interface`**).

drag-callback is allowed to return the *result* **`:default`** rather than a plist. **`:default`** tells the system to do default dragging if there is any. At the time of writing the only place where there is default dragging is on Cocoa for an interface with an **`:interface-pathname`**. *drag-callback* is allowed to return the *result* **`nil`**, meaning do not do dragging.

On **`output-pane`** you add dragging by adding an entry to the *input-model* and which initiates the dragging by calling **`drag-pane-object`**.

Notes: Drag and drop

If **`:image`** is supplied in the *plist* returned by *drag-callback*, the dragging mechanism automatically frees the **`image`** object as if by **`free-image`** when it no longer needs it.

Description: Scroll

Any simple pane can be made scrollable by specifying **`t`** to **`:horizontal-scroll`** or **`:vertical-scroll`**. By default these values are **`nil`**, but some subclasses of **`simple-pane`** default them to **`t`** where appropriate (for instance **`editor-pane`**s always default to having a vertical scroll bar).

For a pane which is scrollable but does not display a scroll bar, pass the value **`:without-bar`** for **`:horizontal-scroll`** or **`:vertical-scroll`**. See:

```
(example-edit-file "capi/output-panes/scrolling-without-bar.lisp")
```

The height and width of a scrollable simple pane can be specified by the initargs **`:scroll-height`** and **`:scroll-width`**, which have the same meaning as **`:internal-min-height`** and **`:internal-min-width`**. See **6.5.2 Constraint Formats** for more information about height and width initargs.

scroll-bar-type controls the visibility of scroll bars on Cocoa. By default, the visibility of scroll bars depends on the System Preferences, which in newer versions of macOS is to have scroll bars that are not always visible. Supplying **`:always-visible`** causes the scroll bars to be always visible as they used to be.

scroll-if-not-visible-p controls scrolling behavior of the parent when the pane is given the input focus. *scroll-if-not-visible-p* can be **`t`**, **`nil`**, or **`:non-mouse`**. See **`scroll-if-not-visible-p`** for details. When this initarg is supplied, the generic function **`(setf scroll-if-not-visible-p)`** is called with it.

Description: Border

The value for *visible-border* can be any of the following, with the stated meanings where applicable:

`nil`	Has no border.
`t`	Has a border.
`:default`	Use the default for the window type.
`:outline`	Add an outline border.

There are various platform/pane class combinations which do not respond to all values of *visible-border*. For instance, on Windows XP with the default theme, **`text-input-choice`** and **`option-pane`** always have a visible border regardless of the value of *visible-border*, while other classes including **`display-pane`**, **`text-input-pane`**, **`list-panel`**, **`editor-pane`** and **`graph-pane`** have three distinct border styles, with *visible-border* **`:default`** meaning the same as *visible-border* **`t`**.

If *internal-border* is non-nil, it should be a non-negative integer specifying the width of an empty region around the edge of the pane.

Description: Miscellaneous

automatic-resize makes the pane resize automatically. This has an effect only if it is placed inside a **`static-layout`** (including subclasses like **`pinboard-layout`**). The effect is that when the **`static-layout`** is resized then the pane also changes its geometry.

The value of *automatic-resize* defines how the pane's geometry changes. It must be a plist of keywords and values which match the keywords of the function **`set-object-automatic-resize`** and are interpreted in the same way.

If the pane is used in the *toolbar-items* list of an **`interface`**, then *toolbar-title* should be a short string that will be shown near to the pane if required for the toolbar.

Notes: Miscellaneous

1. In order to display a simple pane, it needs to be contained within an interface. In a real application you will will define your interface class, but for debugging and just playing around with pane the two convenience functions **`make-container`** and **`contain`** are provided to create an interface with enough support for that pane. The function **`make-container`** just returns a container for an element, and the function **`contain`** displays an interface created for the pane using **`make-container`**.
2. You can also control automatic resizing of a **`simple-pane`** using **`set-object-automatic-resize`**.

Examples

```
(capi:contain (make-instance 'capi:output-pane
                             :background :red
                             :scroll-width 300
                             :horizontal-scroll t))

(setf ep
      (capi:contain
       (make-instance 'capi:editor-pane
                      :visible-border t)))

(setf (capi:simple-pane-cursor ep) :crosshair)
```

For an example illustrating the use of *drag-callback*, see:

```
(example-edit-file "capi/choice/drag-and-drop")
```

See also

`contain`
`define-font-alias`
`set-object-automatic-resize`
3 General Properties of CAPI Panes
6 Laying Out CAPI Panes
9 Adding Toolbars

13.10.3.2 Transparency and the alpha channel

simple-pane-handle *Function*

Summary

Returns the window handle of a pane.

Package

capi

Signature

simple-pane-handle *pane* **=>** *handle*

Arguments

pane⇓	A pane.

Values

handle⇓	An integer, or **nil**.

Description

The function **simple-pane-handle** returns the handle of *pane* in the system that displays it, if there is an underlying window.

On Microsoft Windows *handle* is the hwnd of *pane*.

On X11/Motif, *handle* is the windowid of the main part of *pane* (type Window in the X library).

If *pane* is not displayed, or if *pane* does not have an underlying window, then *handle* is **nil**. Note that layouts do not always have an underlying window.

Use this function with caution: in general, drawing and moving of CAPI windows should be done through the CAPI.

See also

current-dialog-handle
18.7 Handles

simple-pane-visible-height *Function*

Summary

Gets the visible height of a pane.

Package

capi

Signature

`simple-pane-visible-height` *pane* **=>** *result*

Arguments

pane⇓ A simple pane.

Values

result The height of the visible part of *pane*, or **`nil`**.

Description

The function **`simple-pane-visible-height`** returns the height in pixels of the visible part of *pane*, that is the height of the viewport, not including any borders or scroll bars. If *pane* is not displayed the function returns **`nil`**.

See **6.4.1 Width and height hints** for a description of the visible size of a pane.

See also

`simple-pane-visible-size`
`simple-pane-visible-width`
`with-geometry`
3.8 Accessing pane geometry

simple-pane-visible-size *Function*

Summary

Gets the visible size of a pane.

Package

`capi`

Signature

`simple-pane-visible-size` *pane* **=>** *width*, *height*

Arguments

pane⇓ A simple pane.

Values

width The width of the visible part of *pane*, or **`nil`**.

height The height of the visible part of *pane*, or **`nil`**.

Description

The function **`simple-pane-visible-size`** returns the size in pixels of the visible part of *pane*, that is the width and height of the viewport, not including any borders or scroll bars. If *pane* is not displayed the return values are **`nil`**.

See **6.4.1 Width and height hints** for a description of the visible size of a pane.

See also

`simple-pane-visible-height`
`simple-pane-visible-width`
`with-geometry`
3.8 Accessing pane geometry

simple-pane-visible-width *Function*

Summary

Gets the visible width of a pane.

Package

`capi`

Signature

`simple-pane-visible-width` *pane* **`=>`** *result*

Arguments

pane⇓ A simple pane.

Values

result The width of the visible part of *pane*, or **`nil`**.

Description

The function **`simple-pane-visible-width`** returns the width in pixels of the visible part of *pane*, that is the width of the viewport, not including any borders or scroll bars. If *pane* is not displayed the function returns **`nil`**.

See **6.4.1 Width and height hints** for a description of the visible size of a pane.

See also

`simple-pane-visible-height`
`simple-pane-visible-size`
`with-geometry`
3.8 Accessing pane geometry

simple-pinboard-layout *Class*

Summary

A subclass of **`pinboard-layout`** that can contain just one pinboard object or pane as its child, and it adopts the size constraints of that child.

Package

capi

Superclasses

pinboard-layout
simple-layout

Subclasses

graph-pane

Initargs

:child	The child of the pinboard layout.

Description

The class **simple-pinboard-layout** is normally used to place pinboard objects in a layout by placing the layout inside a **simple-pinboard-layout**, thus displaying the pinboard objects. It inherits all of its layout behavior from **simple-layout**.

Examples

```
(setq column
      (make-instance
       'capi:column-layout
       :description
       (list
        (make-instance
         'capi:image-pinboard-object
         :image
         (example-file "capi/graphics/Setup.bmp"))
        (make-instance
         'capi:item-pinboard-object
         :text "LispWorks"))
       :x-adjust :center))


(capi:contain (make-instance
               'capi:simple-pinboard-layout
               :child column))
```

See also

pinboard-object

simple-print-port *Function*

Summary

Prints the contents of an output pane to a printer.

Package

`capi`

Signature

`simple-print-port` *port* **`&key`** *jobname scale dpi printer drawing-mode interactive background*

Arguments

port⇓ An **`output-pane`**.

jobname⇓ A string or **`nil`**.

scale⇓ A positive real or **`nil`**.

dpi⇓ A positive real or **`nil`**.

printer⇓ A printer or **`nil`**.

drawing-mode⇓ One of **`:compatible`**, **`:quality`** or **`nil`**.

interactive⇓ A boolean.

background⇓ A color in the Graphics Ports color system.

Description

The function **`simple-print-port`** prints the **`output-pane`** specified by *port* to the default printer, unless specified otherwise by *printer*.

If *jobname* is non-nil then it is used to set the name of the job that is seen by the user.

scale and *dpi* are used to determine how to transform the output pane's coordinate space to physical units. Their meaning here is the same as in **`get-page-area`**, except that *scale* may also take the value **`:scale-to-fit`**, in which case the pane is printed as large as possible on a single sheet.

The background color of *port* is ignored, and the value given by *background* is used instead. This defaults to **`:white`**.

drawing-mode should be either **`:compatible`** which causes drawing to be the same as in LispWorks 6.0, or **`:quality`** which causes all the drawing to be transformed properly, and allows control over anti-aliasing on Microsoft Windows and GTK+. The default value of *drawing-mode* is **`:quality`**.

For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

If *interactive* is **`t`**, a print dialog is displayed. This is the default. If *interactive* is **`nil`**, then the document is printed to the current printer without prompting the user.

Examples

```
(example-edit-file "capi/printing/simple-print-port")
```

```
(example-edit-file "capi/printing/multi-page")
```

See also

`print-dialog`
13 Drawing - Graphics Ports
16 Printing from the CAPI—the Hardcopy API

slider *Class*

Summary

A pane with a sliding marker, which allows the user to control a numerical value within a specified range.

Package

`capi`

Superclasses

`range-pane`
`titled-object`
`simple-pane`

Initargs

`:print-function`	A function of two arguments, or a format string.
`:show-value-p`	A generalized boolean.
`:start-point`	A keyword.
`:tick-frequency`	An integer, a ratio or the keyword **`:default`**.

Accessors

`slider-print-function`

Readers

`slider-show-value-p`
`slider-start-point`
`slider-tick-frequency`

Description

The class **`slider`** allows the user to enter a number by moving a marker on a sliding scale to the desired value.

show-value-p determines whether the slider displays the current value, on Microsoft Windows and GTK+. The default value is **`t`**. *show-value-p* is ignored on Cocoa.

start-point specifies which end of the slider is the start point in the range. The values allowed depend on the *orientation* of the slider. For horizontal sliders, *start-point* can take these values:

`:left`	The start point is on the left.
`:right`	The start point is on the right.
`:default`	The start point is at the default side (the left).

For vertical sliders, *start-point* can take these values:

`:top`	The start point is at the top.
`:bottom`	The start point is at the bottom.

:default The start point is at the default position, which is the top on Microsoft Windows and Motif, and the bottom on Cocoa.

tick-frequency specifies the spacing of tick marks drawn on the slider. If *tick-frequency* is **:default**, then the slider may or may not draw tick marks according the OS conventions. If *tick-frequency* is 0, then no tick marks are drawn. If *tick-frequency* is a ratio 1/N for integer N>1, then tick marks are drawn to divide the slider range into N sections. Otherwise *tick-frequency* should be an integer greater than 1 which specifies the spacing of tick marks in units between *start* and *end*. The default value of *tick-frequency* is **:default**.

print-function, when supplied, should be a function with signature:

print-function *pane value* **=>** *result*

where *pane* is the slider pane, *value* is its current value, and *result* is a string or **nil**. When the slider pane displays the current value, it calls *print-function* and displays the value as *result*, unless that is **nil**, in which case the value is printed normally.

As a special case, *print-function* can also be a string, which is used as the format string in a call to **format** with one additional argument, the value, that is:

(format nil *print-function value***)**

and the result of this call to **format** is displayed.

Notes

1. **:print-function** is not implemented on Motif.
2. **:print-function** has no effect on Cocoa because the slider pane never displays the value.
3. Use of the *print-function* is determined when the slider pane is displayed. Setting the *print-function* in a slider that did not have a *print-function* when it was first displayed does not work until the slider is destroyed and displayed again. Therefore, if you want to display a **slider** without a *print-function* but set it later, initially you should supply a *print-function* that always returns **nil**, for example:

```
(make-instance 'capi:slider
               :start 10 :end 34
               :print-function 'false)
```

4. *print-function* is useful for displaying fractional values or values that grow logarithmically (or any other non-linear function), because the actual values in a **slider** are always integers that increase linearly as the slider moves.
5. On Windows the value of a **slider** is displayed (when *show-value-p* is true) in a tooltip that is visible only while the user moves the marker with a mouse.

Compatibility note

In LispWorks 6.0 and earlier versions, ticks are drawn as if *tick-frequency* is **:default**.

Examples

Given the default *start* and *end* of 0 and 100, this gives ticks at 0, 25, 50, 75 and 100:

```
(make-instance 'slider :tick-frequency 25)
```

while this gives ticks at 0, 20, 40, 60, 80 and 100:

```
(make-instance 'slider :tick-frequency 1/5)
```

This example illustrates the use of *print-function* to display fractional and non-linear values ranges:

```
(example-edit-file "capi/elements/slider-print-function")
```

See also

3.9.4 Slider, Progress bar and Scroll bar

sorted-object *Class*

Summary

Defines sorting operations.

Package

capi

Superclasses

standard-object

Subclasses

list-panel

Initargs

:sort-descriptions A list.

Description

The class **sorted-object** defines sorting operations.

sorted-object is an interface for sorting the items in **list-panel** and **list-view**.

Each element of *sort-descriptions* is a sorting description object, as returned by **make-sorting-description**. These define various sorting options and are used by **sorted-object-sort-by** and **sort-object-items-by**.

Notes

The subclass **multi-column-list-panel** supports sortable columns.

See also

list-panel
list-view
make-sorting-description
sort-object-items-by

sorted-object-sort-by
sorted-object-sorted-by

sorted-object-sort-by *Generic Function*

Summary

Sets the sorting type of a **sorted-object**.

Package

capi

Signature

sorted-object-sort-by *pane* *new-sort-type* **&key** *allow-reverse*

Arguments

pane⇓	An instance of **sorted-object** or a subclass.
new-sort-type⇓	The sort type to set.
allow-reverse⇓	A boolean.

Description

The generic function **sorted-object-sort-by** sets the sort type of *pane* to *new-sort-type*.

new-sort-type must match by **cl:equalp** the type of one of the sorting descriptions of *pane*.

If *allow-reverse* is non-nil and the sort type already matches *new-sort-type*, then the sort reverses the order of the *items*. The default value of *allow-reverse* is **t**.

If *pane* is a **list-panel**, then **sorted-object-sort-by** also calls **sort-object-items-by** to sort the items with the new sort type. For your own subclasses of **sorted-object** which are not subclasses of **list-panel**, if you need this behavior define an **:after** method that calls **sort-object-items-by**. You can also define **:after** methods on subclasses of **list-panel** to perform other tasks each time the items are sorted.

See also

list-panel
sort-object-items-by
sorted-object
sorted-object-sorted-by

sorted-object-sorted-by *Function*

Summary

Returns the current sorting type and reverse flag of a **sorted-object**.

Package

capi

Signature

sorted-object-sorted-by *pane* **=>** *sort-type***,** *reversed*

Arguments

pane⇓ An instance of **sorted-object** or a subclass.

Values

sort-type⇓ A sort type.

reversed⇓ A boolean.

Description

The function **sorted-object-sorted-by** returns the current sorting type *sort-type* and reverse flag *reversed* of *pane*.

sort-type is the *type* of one of the sorting descriptions of *pane*. *reversed* is true if the pane is sorted in reverse order and false if it is sorted in normal order.

See also

sorted-object
sorted-object-sort-by

sort-object-items-by *Function*

Summary

Sorts items according to a **sorted-object**.

Package

capi

Signature

sort-object-items-by *sorted-object items* **=>** *result*

Arguments

sorted-object⇓ An instance of **sorted-object** or a subclass.

items⇓ A list.

Values

result A permutation of *items*.

Description

The function **sort-object-items-by** sorts *items* according to the current sort type of *sorted-object*, as set by **sorted-object-sort-by**.

Notes

1. If the sort type is reversed, *items* will be sorted in reverse order.
2. The sorting may be destructive, that is *items* may be modified during a call to **sort-object-items-by**.

See also

sorted-object
sorted-object-sort-by
sorted-object-sorted-by

stacked-tree *Class*

Summary

A pane that displays a tree of items in a "stacked" drawing, where each item has an associated value and child items that represent a fraction of that value. Each item is displayed as a rectangle whose width corresponds to the value. Child items are displayed below the item to make a stack of rectangles.

Package

capi

Superclasses

choice
output-pane

Initargs

:root	An object which is the root of the tree of items, or **nil**.
:item-function	A designator for a function.
:value	A non-negative real or **nil**.
:motion-callback	A designator for a function or **nil**.
:colors	A list of colors.
:color-function	A designator for a function or **nil**.
:item-menu-function	A designator for a function or **nil**.
:highlight	A boolean.
:max-level	A positive real or **nil**.
:empty-tree-string	A string or **nil**.

Accessors

stacked-tree-root
stacked-tree-item-function
stacked-tree-item-menu-function
stacked-tree-empty-tree-string

Description

The class **stacked-tree** is a subclass of **output-pane**, which displays a tree of items in a "stacked" drawing. In a stacked drawing, each item of the tree is represented by a horizontal rectangle. The height of the rectangle is fixed to accommodate the height of the font of the **stacked-tree**, while the width corresponds to the "value" of the item. The children of each item are drawn side-by-side below the item itself, to make a stack of rectangles ("stacked").

Within each item's rectangle, the **stacked-tree** displays a label, consisting of the item's name (the third value of *item-function*, see below) and the percentage of the item's value with respect to the value of the **stacked-tree**. The name and/or percentage are omitted if the rectangle is not wide enough.

root and *item-function* specify the tree that the **stacked-tree** is displaying. *root* can be initialized by the **:root** initarg or set by using **(setf stacked-tree-root)** or **modify-stacked-tree**. Likewise, *item-function* can be initialized by the **:item-function** initarg or set by using **(setf stacked-tree-item-function)** or **modify-stacked-tree**. The **stacked-tree** uses *item-function* to traverse the tree starting from *root*.

item-function must be a designator for a function with two arguments: the **stacked-tree** and an item. It should return three values:

item-value — A **real** or **nil**. If *item-value* is a positive **real**, it specifies the item's value, which affects the width of the rectangle used to represent it. If *item-value* is **nil**, then the **stacked-tree** computes the value as the sum of the values of the *item-children*. If *item-value* is not positive, then the item is ignored.

item-children — A list of items that are the children of the item argument. If *item-children* is **nil** then the item is a leaf item and has no children.

item-name — A string or **nil**. When *item-name* is non-nil, the string representation of it (the result of calling the *print-function* inherited from **collection**) is displayed within the rectangle. Just the rectangle is displayed if *item-name* is **nil**.

Both *root* and elements of *item-children* returned by *item-function* can be any object. The only requirement is that *item-function* returns useful values when called with this object. Thus the tree is completely defined by *root* and by what *item-function* returns.

stacked-tree calls *item-function* on items down the tree until either a leaf item is reached (that is when *item-children* is **nil**), or when the depth of the tree reaches *max-level*, if that is non-nil.

Note: Currently there is nothing else to stop the descent down the tree, so you must either have a finite tree, that is your *item-function* must return **nil** as the *item-children* at some level on every branch, or you must supply a non-nil *max-level*.

If *value* is non-nil, it specifies the value on which to base the percentage computations when displaying items. If *value* is **nil** or not specified, it defaults to the *item-value* of *root*, which is the natural value in many cases, but not always. For example, the Profiler tool in the LispWorks IDE uses a *value* that is the number of times that the profiling was done, while the *item-value* of its *root* is the sum of the number of times that each process was profiled, which will be much larger when you profile more than one process.

color-function or *colors* specify the background color used for each displayed rectangle.

If *color-function* is non-nil, then *colors* is ignored. *color-function* is called for each item, the first time the item is displayed, with two arguments: the **stacked-tree** and the item. It must return a color specification (a color-spec or a recognized symbol, see **15 The Color System**), which is then used as the background color of the rectangle for the item.

If *color-function* is **nil**, then *colors* is used. *colors* defaults to a plausible list of colors, so it does not need to be specified. If it supplied, it must be a list of color specifications. The **stacked-tree** selects a random color from this list for each item the first time the item is displayed.

Note: If you do not specify *colors* or *color-function*, then the **stacked-tree** automatically uses darker colors when the window is running with a dark theme. If *color-function* is non-nil, then after a color mode switch, *color-function* is called again for each item that is displyed. *color-function* can use **top-level-interface-dark-mode-p** on the top-level interface of the **stacked-tree** to decide whether it is dark mode or not, but it is probably better to set something inside the **top-level-interface-color-mode-callback** of the interface. If you supply *colors*, then it defines a fixed set that does not change. In this case, you probably want to also set the foreground, so the the color of the text does not chnage either.

If *motion-callback* is non-nil, it is called when the user moves the mouse over the **stacked-tree**, with three arguments: the **stacked-tree**, the item associated with the rectangle at the mouse position or **nil** if the mouse is not over any rectangle, and a vector specifying the coordinates of the item (or **nil** if the item is **nil**). The vector contains eight elements:

0,1,2,3: x, y, width, height

x, y, width, height of the item's rectangle in internal coordinates. Note that the rectangle may have only a partial overlap with the visible area, meaning that only part of it is visible.

4: label-offset. — The horizontal offset in pixels of the beginning of the label from the left side of the rectangle, that is the label's left side is x + label-offset.

5: label-draw-width — The width in pixels that is available to display the label. This is always smaller than the width by a few pixels, and if the rectangle is not visible, may be much smaller or 0.

6: label-width — The width in pixels of the label that should be displayed (as returned by **get-string-extent** when called with the label).

7: percent-width — The width in pixels that is required to display the percentage for the item.

If *highlight* is non-nil, when the user moves the mouse over the **stacked-tree**, the rectangle under the mouse is highlighted.

Note: Both *motion-callback* and *highlight* are implemented by defining the **:motion** gesture in the *input-model* of the **stacked-tree**. If you supply an *input-model* containing **:motion** (see **output-pane**), then this will override the internal one, so *motion-callback* will never be called and *highlight* will not have any effect.

empty-tree-string, if non-nil, should be a string. The default is "Empty STACKED-TREE displayer". It is displayed in the **stacked-tree** if you set *root* to **nil**, or when a non-positive *item-value* is returned when *item-function* is called on *root*.

If *item-menu-function* is non-nil, it is called when the context menu needs to be raised (normally by right-click of the mouse), with two arguments: the **stacked-tree** and the selected item (or **nil** if none is selected). It should return a **menu**, **menu-component** or **nil**. If *item-menu-function* returns a **menu**, then it is used as the context menu. If it returns a **menu-component**, LispWorks makes a menu containing the component followed by the default **stacked-tree** menu (described later). If it returns **nil**, LispWorks raises the default **stacked-tree** menu. If *item-menu-function* is **nil**, LispWorks also raises the default **stacked-tree** menu.

Note: *item-menu-function* is called from the **make-pane-popup-menu** method of **stacked-tree**. You can completely override this by using the **:pane-menu** initarg (see **8.12 Popup menus for panes**), or by defining your own **make-pane-popup-menu** method specializing on **stacked-tree** and your own **interface** class.

Note: When the menu is raised as a result of a mouse click within a rectangle that is associated with an item then this item is selected while the menu is visible. When the menu has been dismissed, if the contents and the selection of the **stacked-tree** are still the same, then the selection goes back to the item that was selected before the mouse click.

Description: capi:output-pane features

Some features of **`stacked-tree`** are inherited from **`output-pane`** as described here.

If you supply a *display-callback* then it will be called after the **`stacked-tree`** has drawn what it wants to draw.

If you supply a *resize-callback,* then the **`stacked-tree`** ensures that the selected item is visible after calling your callback.

`stacked-tree` forces *coordinate-origin* to be **`:fixed-graphics`**.

The **`stacked-tree`** has default initargs for **`:draw-with-buffer`**, **`:horizontal-scroll`** and **`:vertical-scroll`** (all **`t`**). If you override any of these you will affect its behavior.

The **`stacked-tree`** implements its user input interaction (see below) using the *input-model* of **`output-pane`**. If you supply the **`:input-model`** initarg, its value will be appended before the internal input-model of **`stacked-tree`**, so your callbacks will override the internal ones. Note that this affects all interaction, including selection of an item. Your input-model callbacks can use **`stacked-tree-item-at-point`** to find the item at the x,y coordinates.

Description: capi:choice features

Some features of **`stacked-tree`** are inherited from **`choice`** as described here.

The *interaction* of **`stacked-tree`** is always **`:single-selection`**. Setting the *items* signals an error.

choice-selection and **`choice-selected-item`** can be used in the usual way, including setting them. When the selection is set, the **`stacked-tree`** ensures that the selected item is visible.

The *selection-callback* and *action-callback* (inherited from **`callbacks`**) can be used, and are called due to the *input-model* as described above.

Description: Mouse interaction

In the following discussion, *root-width* is the width in pixels of the rectangle used to display *root*. Whenever *root* is changed (and initially), *root-width* is set such that width of the rectangle used to display *root* is the visible width of the **`stacked-tree`**.

Moving the mouse over a **`stacked-tree`** calls *motion-callback* if it is non-nil, and highlights the item under the mouse if *highlight* is non-nil.

Left-click selects the item that was clicked.

Left-double-click on a item changes the *root-width* such that the width of the clicked item's rectangle matches the visible width of the **`stacked-tree`**, and scrolls horizontally such that the item's rectangle starts at the left of the **`stacked-tree`**.

Left-click and drag pans the **`stacked-tree`**, scrolling it such that the clicked point follows the mouse.

Description: Keyboard interaction

The arrow keys change the selected item in the direction indicated if possible. The **`Down`** key moves to the first child of the currently selected item (if any). The **`Left`** and **`Right`** keys move to the item at the same depth if there is any, which may be on a completely different branch of the tree.

The following gestures are also available:

`Ctrl-+`, **`Ctrl--`**: Zoom in, zoom out.

Zooming increases or decreases the *root-width*. It does not affect the vertical dimension.

`Ctrl-i`, **`Ctrl-o`**: Zoom in and out in large steps.

Zoom like **`Ctrl-+`** and **`Ctrl--`**, but in larger steps.

`Return`, **`Ctrl-Return`**: Action callback, alternative action callback.

See **`callbacks`**.

`Ctrl-r`: Reset *root-width*.

Reset the *root-width* to its initial value, so the root of the tree has the visible width of the **`stacked-tree`** at the time it was first displayed, and scroll the root to the left of the **`stacked-tree`**.

`Ctrl-b`, **`Ctrl-f`**: Go backwards, Go forwards.

Go to the previous or next state of the display. Whenever the *root-width* changes or the user left-clicks, the **`stacked-tree`** records the current state of the display, including the *root-width* and scroll position. It uses a ring of length 50 for this record. **`Ctrl-b`** and **`Ctrl-f`** rotate around this ring.

`Ctrl->`, **`Ctrl-<`**: Increment font size, decrement font size.

Try to increment or decrement the font size by one point, and if this fails then try changing the font size by two points. If the font size changes then the height of the rectangles is adjusted to fit the new font height.

Description: context menu

The **`stacked-tree`** context menu contains items to perform the operations listed for keyboard interaction above. It is intended mainly as a way for the user to find the keyboard interaction shortcut. Note that if you override the input-model, and you redefine some of the keys, the menu will be confusing and you should replace it by your own menu.

Notes

The **`stacked-tree`** is useful when the values of an item's children sum to the value of the item itself or less. If the values of the children sum to more than the value of the item, they will overflow to the right of the item and clash with the children of the item's next sibling.

The **`stacked-tree`** is used in the **Stacked Tree** tab of the Profiler tool in the LispWorks IDE.

When **`(setf stacked-tree-root)`** or **`modify-stacked-tree`** is used to set the *root* of a **`stacked-tree`** that is already displayed, it immediately computes an internal representation by traversing the tree. This means that if the tree is big, this operation may take enough time to cause a noticeable delay.

See also

`modify-stacked-tree`
`stacked-tree-item-at-point`
`stacked-tree-zoom-by-factor`
`stacked-tree-width-ratio`
`stacked-tree-history-backward`
`stacked-tree-history-backward`
`stacked-tree-decrease-font-height`
`stacked-tree-decrease-font-height`
`stacked-tree-default-color-function`

stacked-tree-decrease-font-height
stacked-tree-increase-font-height *Functions*

Summary

Decrease or increase the font size in a **stacked-tree**.

Package

capi

Signatures

stacked-tree-decrease-font-height *stacked-tree* **&rest** *ignore*

stacked-tree-increase-font-height *stacked-tree* **&rest** *ignore*

Arguments

stacked-tree⇓ A **stacked-tree**.

ignore⇓ Ignored extra arguments.

Description

The functions **stacked-tree-increase-font-height** and **stacked-tree-decrease-font-height** try to increase/decrease the point size of the font in *stacked-tree*. They add/subtract 1 from the size of the current font, and try to find a font with the new size. If this does not work, they add/subtract 2 and try again. If they find a new font, they set the font in *stacked-tree* to the new font. The heights of the rectangles are adjusted to fit the new font height.

stacked-tree-increase-font-height and **stacked-tree-decrease-font-height** are used by the **Ctrl->** and **Ctrl-<** gestures and you can use them to implement your gestures. The **&rest** *ignore* means that you can use these functions in the input-model directly.

See also

stacked-tree

stacked-tree-default-color-function *Function*

Summary

Returns a color like the default algorithm of **stacked-tree**.

Package

capi

Signature

stacked-tree-default-color-function *stacked-tree* *item* **=>** *color*

Arguments

stacked-tree⇓ A **stacked-tree**.

item⇓ Any object.

Values

color A color specification.

Description

The function **stacked-tree-default-color-function** returns a color for *item* using the same algorithm that **stacked-tree** uses if you do not specify *color-function* or *colors*.

stacked-tree-default-color-function is useful when you want to associate some items with a fixed color. Your code will be something like:

```
(defun my-stacked-tree-color-function (pane node)
  (let ((key (my-get-a-key-from-node node))
        (hash-table (my-find-caching-table)))
    (or (gethash key hash-table)
        (setf (gethash key hash-table)
              (stacked-tree-default-color-function
               pane node)))))
```

Notes

The Profiler tool in the LispWorks IDE uses **stacked-tree-default-color-function** to make all occurences of the same function in the tree have the same color even though the items are not **eq**.

Currently **stacked-tree-default-color-function** actually ignores *stacked-tree* and *item* and returns a random color.

See also

stacked-tree

stacked-tree-history-forward
stacked-tree-history-backward *Functions*

Summary

Go forwards or backwards in the history of a **stacked-tree**.

Package

capi

Signatures

stacked-tree-history-forward *stacked-tree* **&rest** *ignore*

stacked-tree-history-backward *stacked-tree* **&rest** *ignore*

Arguments

stacked-tree⇓ A **stacked-tree**.

ignore⇓ Ignored extra arguments.

Description

A **stacked-tree** has a ring of 50 elements in which it records the *root-width* and scroll position before each change of the *root-width*, and before each user left-click. The function **stacked-tree-history-backward** goes to the previous record of *stacked-tree*, and the function **stacked-tree-history-backward** goes to the next record. Going to the previous/next record means changing the *root-width* and scroll position to their recorded values, and making this record the current one.

Notes

The meaning of *root-width* is explained in **stacked-tree**.

stacked-tree-history-forward and **stacked-tree-history-backward** are used by the **Ctrl-b** and **Ctrl-f** gestures and you can use them to implement your own gestures. The **&rest** *ignore* means that you can use these functions in the input-model directly.

See also

stacked-tree

stacked-tree-item-at-point *Function*

Summary

Return the item whose rectangle is displayed at a given point.

Package

capi

Signature

stacked-tree-item-at-point *stacked-tree x y* **=>** *item*

Arguments

stacked-tree⇓ A **stacked-tree**.

x⇓, *y*⇓ **real**s.

Values

item⇓ An object.

Description

The function **stacked-tree-item-at-point** returns the item that is associated with the rectangle containing the point specified by *x* and *y* in *stacked-tree*. *x* and *y* are internal coordinates that include the scroll position, like the coordinates that are passed to the callbacks.

item is either the root of *stacked-tree* or one of the *item-children* that is returned by the *item-function* of *stacked-tree*.

See also

stacked-tree

stacked-tree-width-ratio *Accessor*

Summary

The horizontal scale of a **stacked-tree**.

Package

capi

Signature

stacked-tree-width-ratio *stacked-tree* **=>** *width-ratio*

setf (stacked-tree-width-ratio *stacked-tree***)** *width-ratio* **=>** *width-ratio*

Arguments

stacked-tree⇓	A **stacked-tree**.
width-ratio⇓	A non-negative **real**.

Values

width-ratio⇓	A non-negative **real**.

Description

The accessor **stacked-tree-width-ratio** accesses the *width-ratio* of *stacked-tree*, which is the ratio between the width of the root rectangle now and when the root was set.

The default action of the **Ctrl-r** gesture is effectively the same as setting **stacked-tree-width-ratio** to 1 and scrolling to the top left.

Note that *width-ratio* is not affected by changes in the width of the **stacked-tree** after the root has been set.

See also

stacked-tree
stacked-tree-zoom-by-factor

stacked-tree-zoom-by-factor *Function*

Summary

Zoom the horizontal scale of a **stacked-tree**.

Package

capi

Signature

stacked-tree-zoom-by-factor *stacked-tree factor* **=>** *width-ratio*

Arguments

stacked-tree⇓	A **stacked-tree**.
factor⇓	A non-negative **real**.

Values

width-ratio⇓	A **real**.

Description

The function **stacked-tree-zoom-by-factor** expands the horizontal dimension of *stacked-tree* by *factor*. If *factor* is between 0 and 1, the horizontal dimension contracts.

This is the same operation as is done by the keyboard gestures **Ctrl--**, **Ctrl-+**, **Ctrl-i** and **Ctrl-o** and you can use it to implement your own gestures.

The returned *width-ratio* is the value returned by **stacked-tree-width-ratio**.

Notes

Evaluating the form:

```
(stacked-tree-zoom-by-factor stacked-tree factor)
```

is equivalent to:

```
(setf (stacked-tree-width-ratio stacked-tree)
      (* (stacked-tree-width-ratio stacked-tree)
         factor))
```

See also

stacked-tree
stacked-tree-width-ratio

start-drawing-with-cached-display *Function*

Summary

Temporarily replaces an output pane's *display-callback* such that it draws from the cached display and optionally adds further drawing.

Package

`capi`

Signature

`start-drawing-with-cached-display` *pane temp-display-callback* **`&key`** *automatic-cancel resize-automatic-cancel user-info from-display-p*

Arguments

pane⇓ An **`output-pane`**.

temp-display-callback⇓
A function designator, or **`nil`**.

automatic-cancel⇓ **`nil`**, **`t`** or a designator for a function of one argument.

resize-automatic-cancel⇓
`nil`, **`t`** or a designator for function of one argument.

user-info⇓ A Lisp object.

from-display-p⇓ A boolean.

Description

The function **`start-drawing-with-cached-display`** caches the display of the output pane *pane* (by calling **`output-pane-cache-display`** with *pane* and *from-display-p*, which defaults to **`nil`**), remembers the current *display-callback*, and replaces the *display-callback* with a callback that first uses the cached display to redraw the area and then uses *temp-display-callback* (if non-nil) to draw additional arbitrary drawing. *temp-display-callback* has the same signature as the *display-callback* of *pane*:

`temp-display-callback` *pane x y width height*

The arguments that will be passed to *temp-display-callback* are determined by calls to **`update-drawing-with-cached-display`** or **`update-drawing-with-cached-display-from-points`**. These functions should be called whenever the temporary display needs to be updated.

The effect of **`start-drawing-with-cached-display`** is undone by any call to **`output-pane-free-cached-display`** (implicit or explicit). Since **`output-pane-cache-display`**, and hence **`start-drawing-with-cached-display`** itself, makes an implicit call to **`output-pane-free-cached-display`**, it is not essential to call **`output-pane-free-cached-display`** between calls. However, the cached display can be quite large, so it is normally better to call **`output-pane-free-cached-display`** as soon as the cache is no longer needed.

If *automatic-cancel* is true then the cached drawing is automatically cancelled (by an implicit call to **`output-pane-free-cached-display`**) when the pane loses the focus or is resized. This is useful when a cached display is used temporarily, for example during drag and drop. If the cached display needs to survive longer, pass **`:automatic-cancel nil`**. The default value of *automatic-cancel* is true. If *automatic-cancel* is a designator for function, it is called with *pane* after the cached displayed is canceled.

resize-automatic-cancel, which defaults to *automatic-cancel*, has the same effect as as *automatic-cancel* but controls what happens when the window is resized rather than when it loses the focus.

user-info is an arbitrary value which will be returned by calls to **`output-pane-cached-display-user-info`** and the call to **`output-pane-free-cached-display`**. It is useful for keeping information during an operation that uses the cached display, for example drag and drop.

Notes

1. The most natural usage of this function is in the **`:press`** input model handler, with a matching **`output-pane-free-cached-display`** call in the **`:release`** handler, to temporarily draw something on top of the permanent display while the user drags the mouse.
2. **`start-drawing-with-cached-display`** and its associated functions (**`update-drawing-with-cached-display`** and **`update-drawing-with-cached-display-from-points`**) use the cached display functions (**`output-pane-cache-display`**, **`output-pane-draw-from-cached-display`**, and **`output-pane-free-cached-display`**). Calling the cached display functions in the scope of **`start-drawing-with-cached-display`** and **`output-pane-free-cached-display`** would confuse them.

Examples

This file shows how to use **`start-drawing-with-cached-display`** in the **`:press`** input model handler:

```
(example-edit-file "capi/output-panes/cached-display")
```

See also

`output-pane-cache-display`
`output-pane-free-cached-display`
`output-pane-cached-display-user-info`
`redraw-drawing-with-cached-display`
`update-drawing-with-cached-display`
`update-drawing-with-cached-display-from-points`
12.5 Transient display on output-pane and subclasses

start-gc-monitor *Function*

Summary

Starts a Lisp Monitor window.

Package

`capi`

Signature

`start-gc-monitor` *screen* **`=>`** *result*

Arguments

screen⇓ A screen.

Values

result⇓ A boolean.

Description

The function **start-gc-monitor** starts a Lisp Monitor window (otherwise known as the GC or Garbage Collector monitor) on the screen *screen*.

result is **t** if it started a Lisp monitor, and **nil** if a Lisp monitor was already running on *screen*.

Note that this works only on Motif. There is no Lisp Monitor window on other platforms.

On Motif, **start-gc-monitor** is called automatically when the LispWorks IDE starts, but you can call **stop-gc-monitor** and **start-gc-monitor** any time.

See also

stop-gc-monitor

start-pane-drag-operation
pane-drag-operation-update
end-pane-drag-operation *Functions*

Summary

Implement a simple dragging operation, which means the pane scrolls as much as the user drags.

Package

capi

Signatures

start-pane-drag-operation *pane x y* **&key** *override-cursor*

pane-drag-operation-update *pane x y*

end-pane-drag-operation *pane x y*

Arguments

pane⇓ A simple-pane with scrollbar(s).

x⇓, *y*⇓ Integers.

override-cursor⇓ A cursor specification or **nil**.

Description

The functions **start-pane-drag-operation**, **pane-drag-operation-update** and **end-pane-drag-operation** together implement a simple dragging operation, which means that *pane* scrolls as much as the user move the cursor. The scrolling happens by a call to **scroll** with the appropriate parameters, in the dimension(s) for which *pane* has scrollbar(s).

start-pane-drag-operation initializes the dragging operation on *pane*. If *override-cursor* cursor is non-nil, the overriding cursor is set internally (not affecting the value that **interface-override-cursor** accesses). *override-cursor* defaults to **:move**.

pane-drag-operation-update performs the dragging operation and calls **scroll** with the appropriate arguments to scroll *pane*, in the direction(s) that the pane has scrollbar(s). *pane* is scrolled based on the difference between the values of *x* and *y* in the calls to **pane-drag-operation-update** and **start-pane-drag-operation**.

end-pane-drag-operation stops the dragging operation, and resets the override cursor to the value of that **interface-override-cursor** accesses. It ignores *x* and *y*.

If **pane-drag-operation-update** or **end-pane-drag-operation** are called without a preceding call to **start-pane-drag-operation** or after a call to **end-pane-drag-operation** without following call to **start-pane-drag-operation** they do nothing.

Notes

These functions are intended to be used as callbacks in input model of output-pane and its subclasses.

Examples

```
(example-edit-file
   "capi/graphics/tracking-pinboard-layout.lisp")
```

See also

scroll
output-pane
simple-pane

static-layout *Class*

Summary

A layout that allows its children to be positioned anywhere within itself.

Package

capi

Superclasses

layout

Subclasses

pinboard-layout

Initargs

:fit-size-to-children

A generalized boolean.

Description

The class **static-layout** is a layout that allows its children to be positioned anywhere within itself.

When a **static-layout** lays out its children, it positions them at the *x* and *y* specified as hints (using **:x** and **:y**), and sizes them to their minimum size (which can be specified using **:visible-min-width** and **:visible-max-width**).

If *fit-size-to-children* is true, the **static-layout** is made sufficiently large to accommodate all of its children, and grows and modifies its scrollbars (if they exist) if necessary when a child is added. This is the default behavior. Otherwise the static layout has a minimum size of one pixel by one pixel which is not affected by the size of its children. If you need the sizing capabilities, then use the class **simple-layout** which surrounds a single child, and adopts the size constraints of that child.

Examples

Here is an example of a static layout placing simple panes at arbitrary positions inside itself.

```
(capi:contain
 (make-instance
  'capi:static-layout
  :description
  (list (make-instance
         'capi:text-input-pane
         :x 20
         :y 100)
        (make-instance
         'capi:push-button-panel
         :x 30
         :y 200
         :items '(1 2 3))))
 :best-width 300 :best-height 300)
```

There are further examples in **20 Self-contained examples**.

See also

pinboard-layout

static-layout-child-geometry *Accessor*

Summary

Gets or sets the geometry of a child in a **static-layout**.

Package

capi

Signature

static-layout-child-geometry *pinboard-object-or-pane* **=>** *x*, *y*, *width*, *height*

setf (static-layout-child-geometry *pinboard-object-or-pane***) (values** *x y width height***) =>** *x*, *y*, *width*, *height*

Arguments

pinboard-object-or-pane⇓

A **pinboard-object** or a pane.

x⇓, *y*⇓, *width*⇓, *height*⇓

Integers.

Values

x⇓, *y*⇓, *width*⇓, *height*⇓

Integers.

Description

The accessor **static-layout-child-geometry** returns as multiple values *x*, *y*, *width* and *height* the geometry of *pinboard-object-or-pane* inside its parent **static-layout**. The setter can be used with all four values at the same time.

The setter can be used be used to set only some of the values, by using **t** for values that need not change. For example, changing the x coordinate to 100 and the width to 50 without affecting the vertical dimension:

```
(setf (static-layout-child-geometry pinboard-object) (values 100 t 50 t))
```

The values that **static-layout-child-geometry** gets or sets are the same as the values that **static-layout-child-position** and **static-layout-child-size** get and set. The setter is more efficient than using the setters of **static-layout-child-position** and **static-layout-child-size** sequentially, and does only one redisplay.

static-layout-child-position *Accessor Generic Function*

Summary

Gets and sets the location of an object inside its parent **static-layout**.

Package

capi

Signature

static-layout-child-position *self* **=>** *x*, *y*

setf (static-layout-child-position *self***) (values** *x y***) =>** *x*, *y*

Arguments

self⇓ A **pinboard-object** or a pane.

x⇓, *y*⇓ Non-negative integers.

Values

x⇓, *y*⇓ Non-negative integers.

Description

The accessor generic function **static-layout-child-position** returns as multiple values *x*, *y* the horizontal and vertical coordinates of *self* inside its parent **static-layout**.

There is also a **setf** expansion which sets the location of *self* in its parent.

Examples

```
(let* ((po (make-instance 'capi:item-pinboard-object
                          :text "5x5" :x 5 :y 5
                          :graphics-args
                          '(:background :red)))
       (pl (capi:contain
            (make-instance 'capi:pinboard-layout
                           :description (list po)
                           :visible-min-width 200
                           :visible-min-height 200))))
  (capi:execute-with-interface
   (capi:element-interface pl)
   #'(lambda (po)
       (dotimes (x 20)
         (mp:wait-processing-events 1)
         (let ((new-x (* (1+ x) 10))
               (new-y (* 5 (+ 2 x))))
           (setf (capi:item-text po)
                 (format nil "~ax~a" new-x new-y))
           (setf (capi:static-layout-child-position po)
                 (values new-x new-y)))))
   po))
```

See also

static-layout
static-layout-child-size

static-layout-child-size *Accessor Generic Function*

Summary

Gets and sets the size of an object inside its parent **static-layout**.

Package

capi

Signature

static-layout-child-size *self* **=>** *width***,** *height*

setf (static-layout-child-size *self***) (values** *width height***) =>** *width***,** *height*

Arguments

self⇓ A **pinboard-object** or a pane.

width⇓, *height*⇓ Positive integers.

Values

width⇓, *height*⇓ Positive integers.

Description

The accessor generic function **static-layout-child-size** returns as multiple values *width*, *height* the dimensions of *self*.

There is also a **setf** expansion which sets the dimensions of *self*.

Examples

```
(let* ((po (make-instance 'capi:pinboard-object
                          :x 5 :y 5
                          :width 5 :height 5
                          :graphics-args
                          '(:background :red)))
       (pl (capi:contain
            (make-instance 'capi:pinboard-layout
                           :description (list po)
                           :visible-min-width 200
                           :visible-min-height 200))))
  (capi:execute-with-interface
   (capi:element-interface pl)
   #'(lambda(po)
       (dotimes (x 20)
         (mp:wait-processing-events 1)
         (let ((new-x (* (1+ x) 10))
               (new-y (* 5 (+ 2 x))))
           (setf (capi:static-layout-child-size po)
                 (values new-x new-y)))))
   po))
```

See also

static-layout
static-layout-child-position

stop-gc-monitor *Function*

Summary

Stop a Lisp Monitor.

Package

capi

Signature

`stop-gc-monitor` *screen* **=>** *result*

Arguments

screen⇓ A screen.

Values

result⇓ A boolean.

Description

The function **`stop-gc-monitor`** stops the Lisp Monitor window on the screen *screen*.

result is **`t`** if it stopped a Lisp monitor, and **`nil`** if there was no Lisp monitor running on *screen*.

Note that this works only on Motif. The Lisp monitor can be restarted with **`start-gc-monitor`**.

See also

`start-gc-monitor`

stop-sound *Function*

Summary

Stops a sound from playing.

Package

`capi`

Signature

`stop-sound` *sound*

Arguments

sound⇓ A sound object returned by **`load-sound`**.

Description

The function **`stop-sound`** stops the sound *sound* from playing.

See also

`play-sound`
18.2 Sounds

switchable-layout *Class*

Summary

A layout which displays only one of its children at a time, and supports switching to another child.

Package

capi

Superclasses

simple-layout

Initargs

:visible-child The currently visible pane from the children.

:combine-child-constraints
A generalized boolean.

Accessors

switchable-layout-visible-child

Readers

switchable-layout-combine-child-constraints

Description

The class **switchable-layout** is a subclass of **simple-layout** which displays only one of its children at a time, and provides functionality for switching the displayed child to one of the other children.

The layout's *description* contains a list of its children. The argument *visible-child* specifies the initially visible child (which defaults to the first of the children).

switchable-layout inherits most of its layout behavior from **simple-layout** as it only ever lays out one child at a time.

combine-child-constraints influences the initial size of the layout. When *combine-child-constraints* is **nil** the constraints of the switchable layout depend only on its currently visible child pane. Switching to a different child pane might cause the layout to resize. When *combine-child-constraints* is non-nil, the constraints depend on all of the child panes, including those that are not visible. This might increase the time taken to create the switchable layout initially, but can prevent unexpected resizing later. The default value of *combine-child-constraints* is **nil**.

Examples

```
(setq children (list
                (make-instance 'capi:push-button
                               :text "Press Me")
                (make-instance 'capi:list-panel
                               :items '(1 2 3 4 5))))

(setq layout (capi:contain
              (make-instance
```

```
                  'capi:switchable-layout
                  :description children)))

    (capi:apply-in-pane-process
     layout #'(setf capi:switchable-layout-visible-child)
     (second children) layout)

    (capi:apply-in-pane-process
      layout #'(setf capi:switchable-layout-visible-child)
      (first children) layout)
```

Here is a further example:

```
(example-edit-file "capi/layouts/switchable")
```

See also

simple-layout
switchable-layout-switchable-children
6 Laying Out CAPI Panes
7 Programming with CAPI Windows
9.9.1 Changing a non-standard toolbar dynamically

switchable-layout-switchable-children *Generic Function*

Summary

Finds the switchable children of a **switchable-layout**.

Package

capi

Signature

switchable-layout-switchable-children *switchable-layout* **=>** *result*

Arguments

switchable-layout⇓ An instance of **switchable-layout** or a subclass.

Values

result A list of panes.

Description

The generic function **switchable-layout-switchable-children** returns as a list all the children of *switchable-layout* that could be made visible by calling the **switchable-layout** accessor **(setf switchable-layout-visible-child)**.

See also

switchable-layout

tab-layout *Class*

Summary

Displays multiple tabs and a pane which shows the main contents. The user can select a tab, which affects what is displayed in the pane.

Package

capi

Superclasses

choice
layout

Initargs

:description	The main layout description.
:items	Specifies the tabs of the tab layout.
:visible-child-function	Returns the visible child for a given selection in switchable mode.
:combine-child-constraints	A generalized boolean which influences the initial size of the layout.
:print-function	The function used to print a name on each tab.
:callback-type	The type of data passed to the callback function in callback mode.
:selection-callback	The function called when a tab is selected, in callback mode.
:image-function	Returns an image for an item, on Microsoft Windows.
:image-lists	A plist of keywords and **image-list** objects, on Microsoft Windows.

Accessors

tab-layout-visible-child-function

Readers

tab-layout-combine-child-constraints
tab-layout-image-function

Description

The class **tab-layout** displays multiple tabs and a pane which shows the main contents. The user can select a tab, which what affects is displayed in the pane.

tab-layout is a subclass of **choice**. Most importantly it inherits **choice**'s *selection* and *selection-callback* behavior, and its *print-function* (which is used to determine the string that appear in each tab), and its *items* behavior (which in turn derives from **collection**).

tab-layout has two modes:

Switchable mode — Selecting a different tab causes a different pane to be displayed.

Callback mode — Selecting a tab merely calls a callback. This callback is responsible for make any required change.

The mode of a **`tab-layout`** is determined by the initarg **`:visible-child-function`**. A non-nil value specifies switchable mode, **`nil`** specifies callback mode.

In switchable mode, selecting on a tab causes a call to the function *visible-child-function* (after doing the *selection-callback*) with the selected item as a single argument. *visible-child-function* must return a pane, which is then displayed. The pane that is returned by *visible-child-function* must not be displayed elsewhere, but can be any pane. Repeated calls with the same item should return the same pane, otherwise it will create a new pane each time the tab is selected.

In callback mode there is only one pane, which you must specify by the initarg **`:description`** (which is inherited from **`layout`**). In this case the *selection-callback* must perform any changes that are needed.

In either mode *combine-child-constraints* influences the initial size of the layout. When *combine-child-constraints* is **`nil`** the constraints of the tab layout depend only on its currently visible tab. Switching to a different tab might cause the layout to resize. When *combine-child-constraints* is non-nil, the constraints depend on all of the tabs, including those that are not visible. This might increase the time taken to create the tab layout initially, but can prevent unexpected resizing later. The default value of *combine-child-constraints* is **`nil`**.

If *image-lists* is specified, it should be a plist containing the keyword **`:normal`** as a key. The corresponding value should be an **`image-list`** object. No other keys are supported at the present time. The **`image-list`** associated with the **`:normal`** key is used with the *image-function* to specify an image to display in each tab.

The *image-function* is called on an item to return an image associated with the item. It can return one of the following:

A pathname or string — This specifies the filename of a file suitable for loading with **`load-image`**. Currently this must be a bitmap file.

A symbol — The symbol must have been previously registered by means of a call to **`register-image-translation`**.

An **`image`** object — For example, as returned by **`load-image`**.

An image locator object — This allowing a single bitmap to be created which contains several button images side by side. See **`make-image-locator`** for more information. On Microsoft Windows, it also allows access to bitmaps stored as resources in a DLL.

An integer — This is a zero-based index into the tab-layout's **`image-list`**. This is generally only useful if the image list is created explicitly. See **`image-list`** for more details.

Notes

image-lists and *image-function* are implemented only on Microsoft Windows.

Examples

The following example shows the use of the switchable mode of **`tab-layout`**. Each tab is linked to an output pane by pairing them in the *items* list.

```
(defun switchable-tab-layout ()
  (let* ((red-pane (make-instance
                    'capi:output-pane
                    :background :red))
```

```
            (blue-pane (make-instance
                        'capi:output-pane
                        :background :blue))
            (tl (make-instance
                 'capi:tab-layout
                 :items
                 (list (list "Red" red-pane)
                       (list "Blue" blue-pane))
                 :print-function 'car
                 :visible-child-function 'second)))
       (capi:contain tl)))
```

```
(switchable-tab-layout)
```

Here is an example of the callback mode of **`tab-layout`**, which uses the selection of a tab to change the nodes of a graph pane through the *selection-callback*.

```
(defun non-switchable-tab-layout (tabs)
  (let* ((gp (make-instance
              'capi:graph-pane))
         (tl (make-instance
              'capi:tab-layout
              :description (list gp)
              :items tabs
              :visible-child-function nil
              :print-function
              (lambda (x)
                (format nil "~R" x))
              :callback-type :data
              :selection-callback
              #'(lambda (data)
                  (setf (capi:graph-pane-roots gp)
                        (list data))))))
    (capi:contain tl)))
```

```
(non-switchable-tab-layout '(1 2 4 5 6))
```

See also

callbacks
simple-layout
switchable-layout
tab-layout-panes
tab-layout-visible-child
6.6.2 Tab layouts
7 Programming with CAPI Windows

tab-layout-panes *Function*

Summary

Returns the panes in a **`tab-layout`**.

Package

capi

Signature

`tab-layout-panes` *tab-layout* **`=>`** *panes*

Arguments

tab-layout⇓ A **`tab-layout`**.

Values

panes A list.

Description

The function **`tab-layout-panes`** returns the panes in a **`tab-layout`**. Note that this is not necessarily the same as the items of *tab-layout,* since *visible-child-function* and/or *key* may be specified.

See also

`tab-layout`
6.6.2 Tab layouts

tab-layout-visible-child *Function*

Summary

Returns the visible child in a **`tab-layout`**.

Package

`capi`

Signature

`tab-layout-visible-child` *tab-layout* **`=>`** *result*

Arguments

tab-layout⇓ A **`tab-layout`**.

Values

result A pane.

Description

The function **`tab-layout-visible-child`** returns the currently-visible pane in *tab-layout*.

See also

`tab-layout`
6.6.2 Tab layouts

text-input-choice *Class*

Summary

This pane consists of a text input area, and a button. Clicking on the button displays a list of editable strings, and selecting one of the strings automatically pastes it into the text input area.

Package

`capi`

Superclasses

`choice`
`text-input-pane`

Initargs

`:visible-items-count`
An integer specifying the maximum length of the list, or the symbol **`:default`**.

`:popup-callback` A function called just before the list appears, or **`nil`**.

Description

The class **`text-input-choice`** behaves in the same way as a **`text-input-pane`**, but has additional functionality. The element inherits from **`choice`**, and the choice *items* are used as the items to display when the user clicks on the button.

The *callback* is called when the user presses the **`Return`** key.

The *selection-callback* is called when the user selects an item in the list.

Notes

The user can edit the items in a **`text-input-choice`**. For an element with similar functionality which does not allow editing, see **`option-pane`**.

Compatibility note

In LispWorks 6.0 and earlier versions the **`text-input-pane`** initarg value *enabled* **`:read-only`** is not supported for **`text-input-choice`** on Microsoft Windows. This restriction is removed for LispWorks 6.1 and later versions.

Examples

```
(example-edit-file "capi/elements/text-input-choice")
```

See also

`choice`
`option-pane`
`text-input-pane`
5 Choices - panes with items
9.7.1 Toolbar items other than buttons with images

text-input-pane *Class*

Summary

The class **text-input-pane** is a pane for entering a single line of text.

Package

capi

Superclasses

titled-object
simple-pane

Subclasses

multi-line-text-input-pane
password-pane
text-input-choice

Initargs

:text	The text in the pane.
:caret-position	The position of the caret in the text (from 0).
:max-characters	The maximum number of characters allowed.
:enabled	Controls the enabled state of the pane.
:callback	A function usually called when the user presses **Return**.
:callback-type	The type of arguments to *callback*.
:change-callback	A function called when a change is made.
:change-callback-type	The type of arguments to *change-callback*.
:text-change-callback	A function designator.
:confirm-change-function	A function called to validate a change. Implemented for Motif only.
:gesture-callbacks	A list of pairs **(***gesture* **.** *callback***)**.
:completion-function	A function called to complete the text.
:in-place-completion-function	A function designator.
:file-completion	**t**, **nil** or a pathname designator.
:in-place-filter	A boolean.
:directories-only	A boolean.
:ignore-file-suffices	A list of strings or the keyword **:default**.

:complete-do-action

A boolean.

:navigation-callback

A function called when certain keyboard gestures occur in the pane.

:editing-callback A function called when editing starts or stops.

:buttons A plist specifying buttons to add, or **t** or **nil**.

:search-field Along with the next four initargs, this is implemented only on Cocoa. It specifies that the pane has "recent-items", which also means using **NSSearchField**.

:recent-items See **:search-field** above.

:recent-items-name See **:search-field** above.

:maximum-recent-items

See **:search-field** above.

:recent-items-mode See **:search-field** above.

Accessors

text-input-pane-text
text-input-pane-max-characters
text-input-pane-enabled
text-input-pane-callback
text-input-pane-confirm-change-function
text-input-pane-change-callback
text-input-pane-completion-function
text-input-pane-navigation-callback
text-input-pane-editing-callback
text-input-pane-buttons-enabled

Readers

text-input-pane-caret-position

Description

The class **text-input-pane** provides a great deal of flexibility in its handling of the text being entered. It starts with the initial text and caret-position specified by the arguments *text* and *caret-position* respectively. It limits the number of characters entered with the *max-characters* argument (which defaults to **nil**, meaning there is no maximum).

If *enabled* is **nil**, the pane is disabled. If *enabled* is **:read-only**, then the pane shows the text and allows it to be selected without it being editable. In this case the visual appearance varies between window systems, but often the text can be copied and the caret position altered. If *enabled* is any other true value, then the pane is fully enabled. The default value of *enabled* is **t**.

You can programmatically get and set the selection and caret position by **set-text-input-pane-selection**, **text-input-pane-selected-text**, **text-input-pane-selection** and **text-input-pane-caret-position**. You can programmatically perform standard edit operations by using **text-input-pane-paste**, **text-input-pane-copy**, **text-input-pane-cut** and **text-input-pane-delete**. You can programmatically invoke the completion functions by **text-input-pane-complete-text** and **text-input-pane-in-place-complete**.

For more than one line of input, use **multi-line-text-input-pane**.

Description: Callbacks

callback, if non-nil, is called when the user presses **Return**, unless *navigation-callback* is non-nil, in which case *navigation-*

callback is called instead. If the pane has "recent-items" (implemented only on Cocoa) then the timing of calls to *callback* is modified: see the discussion of *recent-items* below for the details.

When the *text* or *caret-position* is changed, the callback *change-callback* is called with the *text*, the pane itself, the interface and the *caret-position*. The arguments that are passed to the *change-callback* can be altered by specifying the *change-callback-type* (see the **`callbacks`** class for details of possible values).

With the Motif implementation it is possible to check changes that the user makes to the **`text-input-pane`** by providing a *confirm-change-function* which gets passed the new text, the pane itself, its interface and the new caret position, and which should return non-nil if it is OK to make the change. If **`nil`** is returned, then the pane will be unaltered and a beep will be signalled to indicate that the new values were invalid.

gesture-callbacks provides callbacks to perform for specific keyboard gestures. Each *gesture* must be an object that **`sys:coerce-to-gesture-spec`** can coerce to a **`sys:gesture-spec`**. Each *callback* can be a callable (symbol or function) which takes one argument, the pane. Alternatively each *callback* can be a list of the form **`(`***function arguments***`)`**. Note that in this case, the pane itself is not automatically passed to the *function* amongst *arguments*.

When the user enters a gesture that matches *gesture* in any pair amongst *gesture-callbacks*, the *callback* is executed and the gesture is not processed any more.

text-change-callback is a change callback (see *change-callback*) that is called only when the text in the pane changes. In contrast, *change-callback* is also called when the caret moves. If both *text-change-callback* and *change-callback* are supplied, only *text-change-callback* is invoked.

Notes: Callbacks

1. *change-callback* is potentially called more than once for each user gesture.
2. The interaction of in-place completion is implemented using *gesture-callbacks*. Gestures which you define explicitly by *gesture-callbacks* override the gestures which are defined implicitly by the in-place completion mechanism.
3. For gestures that change the text, *text-change-callback* is probably better than *gesture-callbacks*.

Description: Completion

A *completion-function* can be specified which will get called when the completion gesture is made by the user (by pressing the **`Tab`** key) or when **`text-input-pane-complete-text`** is called. The function should have signature:

`completion-function` *pane string* **`=>`** *completions*, *start*, *end*

where *pane* is the **`text-input-pane`** itself and *string* is the string to complete. When completion is invoked *completion-function* is called with *pane* and a string containing the text of pane to the left of the cursor.

The *completion-function* is called with the pane and the text to complete and should return either **`nil`**, the completed text as a string or a list *completions* of candidate completions. In the latter case, the CAPI will prompt the user for the completion they wish, and this will become the new text. In addition, the *completion-function* can return two more values, *start* and *end*, which specify a range in the text that is to be replaced if the completion is successful.

When *complete-do-action* is non-nil, completion of the text in the pane automatically invokes *callback* (if *callback* is non-nil). The default value of *complete-do-action* is **`nil`**.

in-place-completion-function tells the pane to do in-place completion and specifies the function to use. The function should have signature:

`in-place-completion-function` *pane string* **`=>`** *completions*, *start*, *end*

where *pane* is the **`text-input-pane`** itself and *string* is the string to complete. When in-place completion is invoked *in-*

place-completion-function is called with *pane* and a string containing the text of pane to the left of the cursor.

completions needs to be a list of strings that are possible completions, a single string that is a unique completion, or the symbol **`:destroy`**. **`:destroy`** means that the in-place completion needs to stop and close the in-place window. In addition, the completion function can return two more values, *start* and *end*, which specify a range in the text that is to be replaced if the completion is successful. The function is called repeatedly whenever there is a change to the text that should be completed.

The default value of *in-place-completion-function* is **`nil`**.

file-completion, if non-nil, tells the pane to do file completion using an in-place window. The user invokes In-place completion or file completion by pressing the **`Up`** or **`Down`** key. See **10.6 In-place completion** for more details of the user interaction.

If *file-completion* is a pathname designator, its location is used as the root path for the completion.

The default value of *file-completion* is **`nil`**.

in-place-filter takes effect only when either *in-place-completion-function* or *file-completion* is non-nil. If *in-place-filter* is **`t`** then the in-place window can have a filter. Note that the filter needs to requested by a user gesture. **`Control+Return`** is the default in-place filter gesture. The default value of *in-place-filter* is **`t`**.

directories-only takes effect only if *file-completion* is used. If *directories-only* is **`t`** then in-place completion shows only directories. The default value of *directories-only* is **`nil`**.

ignore-file-suffices takes effect only if *file-completion* is used. It tells in-place completion to ignore files whose file namestring (the result of **`cl:file-namestring`**) ends with any of the strings in the list *ignore-file-suffices*. If *ignore-file-suffices* is **`:default`**, then completion uses the default value, which is the value of **`editor:*ignorable-file-suffices*`** (see **`config/a-dot-lispworks.lisp`**).

Notes: Completion

1. If *in-place-completion-function* needs some dynamic information, it can put it in a property of the pane (using **`capi-object-property`**).
2. For dynamic control over whether there is an in-place completion or not, specify an *in-place-completion-function* that simply returns the keyword **`:destroy`** when there should be no completion.
3. The initarg **`:file-completion`** overrides **`:in-place-completion-function`**.
4. The in-place completion mechanism uses *gesture-callbacks* to implement the functionality.
5. **`:in-place-filter`** can be used to specify that the in-place window can have a filter.
6. The behavior of in-place completion is somewhat different from other completion.
7. The initargs **`:directories-only`** and **`:ignore-file-suffices`** can be used to change the behavior of the completion.

Description: Editing and navigation callbacks

navigation-callback, if non-nil, is a function that will be called when certain navigation gestures are used in the **`text-input-pane`**. The function is called with two arguments, the pane itself, and one of the following keywords:

`:tab-forward`	**`Tab`** was pressed.
`:tab-backward`	**`Tab Backwards`** (usually **`Shift+Tab`**) was pressed.
`:return`	**`Return`** was pressed.

:shift-return **Shift+Return** was pressed.

:enter **Enter** was pressed.

:shift-enter **Shift+Enter** was pressed.

When *navigation-callback* is non-nil, it is called instead of *callback* when **Return** is pressed. *callback* is still called via an OK button if there is one (see *buttons* below).

navigation-callback is implemented only on Microsoft Windows and Cocoa.

editing-callback, if non-nil, is a function of two arguments:

editing-callback *pane type*

pane is the **text-input-pane** and *type* is a keyword. *editing-callback* is called with *type* **:start** when the user starts editing and *type* **:end** when the user stops editing. In general, this occurs when the focus changes, but on Cocoa *type* **:start** is passed when the first change is made to the text.

Notes: Editing and navigation callbacks

Enter is the key usually found on the numeric keypad.

Description: Buttons

buttons specifies toolbar buttons which appear next to the pane and facilitate user actions on it. It also specifies the position of the buttons relative to the pane. This feature appears in the LispWorks IDE, for example the Class box of the Class Browser.

The allowed keys and values of the plist *buttons* are:

:ok A boolean or a plist, default value **t**. If true, a button which calls *callback* appears. If the value is a plist then this plist supplies details for the button, as described below.

:cancel A boolean or a plist, default value **nil**. If true, a button which calls *cancel-function* appears. A plist value is interpreted as for **:ok** and can also contain the key **:accelerator** which specifies an accelerator used for the button. There is no default accelerator.

:completion A boolean or a plist. If true, a button which calls *completion-function* appears. The default value is **t** if *completion-function* is non-nil, and **nil** otherwise. A plist value is interpreted as for **:ok**.

:browse-file A keyword or a plist. If true, a button which invokes **prompt-for-file** appears. If the value is **:save** or **:open** then it is passed as the operation argument to **prompt-for-file**, replacing the text in the pane if successful. If the value is a plist, then it supplies details for the button, as described below, and can also contain the keywords **:message** to specify a message for the file prompter; **:pathname** to specify the default pathname of the file prompter (defaults to the text in the **text-input-pane**), **:directory** to use **prompt-for-directory** rather than **prompt-for-file**, or any of the keywords **:ok-check**, **:filter**, **:filters**, **:if-exists**, **:if-does-not-exist**, **:operation**, **:owner**, **:pane-args** or **:popup-args** which are passed directly to **prompt-for-file** or **prompt-for-directory**.

:cancel-function A function that expects the pane as its single argument. The default is a function which sets *text* to the empty string.

:help Specifies a help button. The value must be a plist containing either keys **:function** and optionally **:arguments**, or the keys **:title**, **:message** and optionally **:dialog-p**.

If *function* is supplied, when the user presses the help button it calls:

```
(apply function pane arguments)
```

where *pane* is the **text-input-pane**. *title*, *message* and *dialog-p* are ignored in this case.

Otherwise when the user presses the help button it opens a window with title *title* displaying the string *message* in a **display-pane**. The message can be long, and can include newlines. The window is owned by the pane, but is not modal, so the user can interact with the pane while the help window is displayed. If *dialog-p* is true, the help window is raised as a dialog. The default value for *dialog-p* is **nil**. *function* and *arguments* are ignored in this case.

The plist can contain other keys as described below.

:orientation The value is either **:horizontal** or **:vertical**. *orientation* controls the orientation of the toolbar. This is useful for **multi-line-text-input-pane**. The default value is **:horizontal**.

:adjust The value is **:top**, **:center**, **:centre** or **:bottom**. *adjust* controls how the buttons are adjusted vertically relative to the text input pane. This is useful for **multi-line-text-input-pane**. The default value is **:center**.

:position The value is **:top**, **:bottom**, **:left** or **:right**. *position* determines whether the buttons appear above, below, left or right of the text input pane. If **:position** is not supplied, then the buttons appear to the right of the pane.

The value **nil** for *buttons* means there are no buttons - this is the default. When *buttons* is true the buttons appear or not according to their specified values or their default values.

All of the button plists (for **:ok**, **:cancel**, **:help** and so on) can contain the following keys and values in addition to those mentioned above:

:enabled A value that controls whether the button is enabled. (See the reader **text-input-pane-buttons-enabled**).

:image The image to use for the button. This should be either a pathname or string naming an image file to load, a symbol giving the id of an image registered with **register-image-translation**, an **image** object as returned by **load-image** or an **external-image**. The default image is one of the symbols **ok-button**, **cancel-button** or **complete-button**, which are pre-registered image identifiers corresponding to each button.

:help-key The *help-key* used to find a tooltip for the button.

The reader **text-input-pane-buttons-enabled** returns a list containing keywords such as**:ok**, **:cancel** and **:completion**, one for each corresponding button (as specified by *buttons*) that is currently enabled.

The writer **(setf text-input-pane-buttons-enabled)** takes a list of keywords as described for the reader and sets the enabled state of the buttons, enabling each button if it appears in the list and disabling it otherwise. The value **t** can also be passed: this enables all the buttons.

Description: Search field and recent items

If *search-field* is a string and *recent-items-name* is not supplied, then the value *search-field* is used as the name. See the discussion of *recent-items* below.

If any of *search-field*, *recent-items* or *recent-items-name* is supplied and is non-nil, the pane uses **`NSSearchField`**, and also has "recent items". An **`NSSearchField`** has a different appearance from **`text-input-pane`**, can display recent items menu, and its input behavior is a little different too.

If *recent-items* is non-nil, it must be a list of strings, or **`t`**. When it is a list of strings, it specifies the initial list of "recent items". When it is **`t`**, it simply specifies that the pane should handle recent items.

If *recent-items-name* is non-nil, it should be a string. The string specifies the autosave name of the pane. When a pane has an autosave name, Cocoa remembers the list of recent items for pane with the same autosave name and same application. The record persists between invocations of the application.

If *recent-items-name* is not supplied or is **`nil`**, and *search-field* is a string, it is used instead as the name.

The maximum number of recent items defaults to 50 and can be controlled by the initarg value *maximum-recent-items*. The value 0 can be used to switch off the "recent items" feature, including the menu.

The recent items list can be read and set by **`text-input-pane-recent-items`**, or modified by any of **`text-input-pane-replace-recent-items`**, **`text-input-pane-delete-recent-items`**, **`text-input-pane-append-recent-items`**, **`text-input-pane-prepend-recent-items`** and **`text-input-pane-set-recent-items`**.

The input behavior of **`text-input-pane`** with "recent items" is the same is that of other **`text-input-pane`**s except for the timing of calls to *callback*. Note that this refers to the function that is passed with the initarg **`:callback`**, so *change-callback* is not affected.

By default, each time the user types a character it causes a scheduling of *callback* some short time later. If the user types another character before the callback, it is re-scheduled later. The result is that as long as the user types, there are no callbacks, but once the user stops a callback is generated.

The behavior of *callback* can be controlled by the initarg value *recent-items-mode*, which can be one of **`:explicit`**, **`:delayed`** or **`:immediate`**. **`:explicit`** gives the same behavior as a normal **`text-input-pane`**, **`:delayed`** is the default described above, and **`:immediate`** means doing a callback immediately after each character. In addition, when the user selects an item from the recent items menu or clicks its **Cancel** button, the *callback* is called. In the case of the **Cancel** button, the string would be empty.

Examples

```
(capi:contain (make-instance 'capi:text-input-pane
                             :text "Hello world"))


(setq tip (capi:contain
                       (make-instance
                        'capi:text-input-pane
                        :enabled nil)))

(capi:apply-in-pane-process
  tip #'(setf capi:text-input-pane-enabled) t tip)

(capi:apply-in-pane-process
 tip #'(setf capi:text-input-pane-enabled) nil tip)

(capi:apply-in-pane-process
 tip #'(setf capi:text-input-pane-text) "New text" tip)


(capi:contain (make-instance
               'capi:text-input-pane
               :text "Hello world"
               :callback #'(lambda (text interface)
                             (capi:display-message
```

```
                                   "Interface ~S's text: ~S"
                                   interface text))))
```

This example uses a plist value for the *buttons* key **:cancel** to specify that the Cancel button is initially disabled:

```
(capi:contain
 (make-instance 'capi:text-input-pane
                :buttons
                '(:ok t :cancel (:enabled nil))))
```

This example shows how to specify a Help button which displays a help message:

```
(defvar *help-message* "A long help message.")

(capi:contain
 (make-instance 'capi:text-input-pane
                :buttons
                `(:help
                  (:title "help window"
                   :message ,*help-message*))))
```

This example shows to specify a button which prompts for a directory:

```
(capi:contain
 (make-instance 'capi:text-input-pane
                :buttons
                '(:browse-file (:directory t
                                :image :std-file-open)
                  :ok nil))
  :title "Enter a directory path")
```

This example illustrates the use of *gesture-callbacks*. **Ctrl+e** moves the cursor to the end of the input, **Ctrl+a** moves it to the start, and **Ctrl+6** does something else:

```
(capi:contain
 (make-instance
  'capi:text-input-pane
  :gesture-callbacks
  (list
   (cons
    "Ctrl-e"
    #'(lambda (tip)
        (setf (capi:text-input-pane-caret-position tip)
              (length (capi:text-input-pane-text tip)))))
   (cons
    "Ctrl-a"
    #'(lambda (tip)
        (setf (capi:text-input-pane-caret-position tip)
              0)))
   (cons
    "Ctrl-6" 'do-something-else))))
```

There is a further example here:

```
(example-edit-file "capi/elements/text-input-pane")
```

See also

display-pane
editor-pane

multi-line-text-input-pane
set-text-input-pane-selection
text-input-choice
text-input-pane
text-input-pane-complete-text
text-input-pane-copy
text-input-pane-cut
text-input-pane-delete
text-input-pane-in-place-complete
text-input-pane-paste
text-input-pane-selected-text
text-input-pane-selection
title-pane
3.5.2 Text input panes
3.1.4.1 Controlling Mnemonics
3.5 Displaying and entering text
19.3.2 Matching resources for GTK+
9 Adding Toolbars
9.7.1 Toolbar items other than buttons with images
10.6 In-place completion

text-input-pane-append-recent-items
text-input-pane-delete-recent-items
text-input-pane-prepend-recent-items
text-input-pane-replace-recent-items *Functions*

Summary

Modifies the recent items list in a **text-input-pane** on Cocoa.

Package

capi

Signatures

text-input-pane-append-recent-items *text-input-pane* **&rest** *strings*

text-input-pane-delete-recent-items *text-input-pane* **&rest** *strings*

text-input-pane-prepend-recent-items *text-input-pane* **&rest** *strings*

text-input-pane-replace-recent-items *text-input-pane* **&rest** *strings*

Arguments

text-input-pane⇓	A **text-input-pane** with recent items.
strings⇓	Strings.

Description

These functions modify the recent items list in *text-input-pane*, which must have recent-items (see **text-input-pane** initargs **:search-field**, **:recent-items** and **:recent-items-name**).

text-input-pane-append-recent-items appends *strings* at the end of the recent items, using **text-input-pane-set-recent-items** with *where* = **:end**.

text-input-pane-delete-recent-items deletes from the recent items any item that matches any of *strings* (compared using **cl:string-equal**), using **text-input-pane-set-recent-items** with *where* = **:delete**.

text-input-pane-prepend-recent-items prepends *strings* at the beginning of the recent items, using **text-input-pane-set-recent-items** with *where* = **:start**.

text-input-pane-replace-recent-items uses **text-input-pane-set-recent-items** with *where* = **:replace**, replacing the recent items in the pane by *strings*. It has the same effect as **(setf text-input-pane-recent-items)**, but takes the strings as **&rest** arguments.

None of these function return a meaningful value.

Notes

text-input-pane-append-recent-items, **text-input-pane-delete-recent-items**, **text-input-pane-prepend-recent-items** and **text-input-pane-replace-recent-items** are implemented only on Cocoa.

See also

text-input-pane
text-input-pane-set-recent-items

text-input-pane-complete-text *Function*

Summary

Calls the *completion-function* in a **text-input-pane**.

Package

capi

Signature

text-input-pane-complete-text *pane* **=>** *result*

Arguments

pane⇓ A **text-input-pane**.

Values

result⇓ A string, or **nil**.

Description

The function **text-input-pane-complete-text** calls the *completion-function* of *pane* with the current *text*. If this call is successful, then the *text* of *pane* is set to the result, and **text-input-pane-complete-text** returns this result. Otherwise, *result* is **nil**.

Note: the *completion-function* may return a list of completion candidates, in which case **`text-input-pane-complete-text`** prompts the user to select one of the candidates.

See also

`text-input-pane`

text-input-pane-copy *Function*

Summary

Copies the selected text in a **`text-input-pane`** to the clipboard.

Package

`capi`

Signature

`text-input-pane-copy` *text-input-pane*

Arguments

text-input-pane⇓ An instance of **`text-input-pane`** or a subclass.

Description

The function **`text-input-pane-copy`** performs the clipboard copy operation on the selected text in *text-input-pane*. It does nothing if there is no selection.

See also

`clipboard`
`text-input-pane`
`text-input-pane-selection`
`text-input-pane-cut`
`text-input-pane-delete`
`text-input-pane-paste`

text-input-pane-cut *Function*

Summary

Cuts the selected text in a **`text-input-pane`** to the clipboard.

Package

`capi`

Signature

text-input-pane-cut *text-input-pane*

Arguments

text-input-pane⇓ An instance of **text-input-pane** or a subclass.

Description

The function **text-input-pane-cut** performs the clipboard cut operation on the selected text in *text-input-pane*. It does nothing if there is no selection.

See also

clipboard
text-input-pane
text-input-pane-selection
text-input-pane-copy
text-input-pane-delete
text-input-pane-paste

text-input-pane-delete *Function*

Summary

Deletes the selected text in a **text-input-pane**.

Package

capi

Signature

text-input-pane-delete *text-input-pane*

Arguments

text-input-pane⇓ An instance of **text-input-pane** or a subclass.

Description

The function **text-input-pane-delete** deletes the selected text in *text-input-pane*. It does nothing if there is no selection.

See also

clipboard
text-input-pane
text-input-pane-selection
text-input-pane-cut
text-input-pane-copy
text-input-pane-paste

text-input-pane-in-place-complete *Function*

Summary

Raises the non-focus completion window.

Package

capi

Signature

text-input-pane-in-place-complete *text-input-pane*

Arguments

text-input-pane⇓ A **text-input-pane**.

Description

The function **text-input-pane-in-place-complete** raises the non-focus completion window.

The pane *text-input-pane* must have been made with either *in-place-completion-function* or *file-completion*. See the description of this functionality in **text-input-pane**.

See also

text-input-pane

text-input-pane-paste *Function*

Summary

Pastes the clipboard text into a **text-input-pane**.

Package

capi

Signature

text-input-pane-paste *text-input-pane*

Arguments

text-input-pane⇓ An instance of **text-input-pane** or a subclass.

Description

The function **text-input-pane-paste** performs the clipboard paste operation on *text-input-pane*, replacing any selected text.

See also

clipboard
text-input-pane
text-input-pane-selection
text-input-pane-cut
text-input-pane-copy
text-input-pane-delete

text-input-pane-recent-items *Accessor*

Summary

Gets and sets the recent items in a **text-input-pane** on Cocoa.

Package

capi

Signature

text-input-pane-recent-items *text-input-pane* **=>** *list-of-strings*

(setf text-input-pane-recent-items) *list-of-strings* *text-input-pane* **=>** *list-of-strings*

Arguments

text-input-pane⇓ A **text-input-pane** with recent items.

list-of-strings⇓ A list of strings.

Values

list-of-strings⇓ A list of strings.

Description

The accessor **text-input-pane-recent-items** gets and sets the recent items in *text-input-pane*, which must have recent-items. (see **text-input-pane** initargs **:search-field**, **:recent-items** and **:recent-items-name**).

The value *list-of-strings* passed to **(setf text-input-pane-recent-items)** must be a list of strings.

Notes

text-input-pane-recent-items is implemented only on Cocoa.

text-input-pane-recent-items does not work properly before the pane is displayed.

See also

text-input-pane
text-input-pane-set-recent-items

text-input-pane-selected-text *Function*

Summary

Returns the selected text in a **text-input-pane**.

Package

capi

Signature

text-input-pane-selected-text *text-input-pane* **=>** *result*

Arguments

text-input-pane⇓ An instance of **text-input-pane** or a subclass.

Values

result A string or **nil**.

Description

The function **text-input-pane-selected-text** returns the selected text in *text-input-pane*, or **nil** if there is no selection.

See also

text-input-pane
text-input-pane-selection
text-input-pane-selection-p

text-input-pane-selection *Function*

Summary

Returns the bounds of the selection in a **text-input-pane**.

Package

capi

Signature

text-input-pane-selection *pane* **=>** *start***,** *end*

Arguments

pane⇓ A **text-input-pane**.

Values

start⇓, *end*⇓ Non-negative integers.

Description

The function **`text-input-pane-selection`** returns as multiple values the bounding indexes of the selection in *pane*. That is, *start* is the inclusive index of the first selected character, and *end* is one greater than the index of the last selected character.

If there is no selection, then both *start* and *end* are the caret position in *pane*.

See also

`set-text-input-pane-selection`
`text-input-pane`
`text-input-pane-selected-text`
`text-input-pane-selection-p`

text-input-pane-selection-p *Function*

Summary

Returns true if there is selected text in a **`text-input-pane`**.

Package

`capi`

Signature

`text-input-pane-selection-p` *pane* **`=>`** *selectionp*

Arguments

pane⇓ A **`text-input-pane`**.

Values

selectionp A boolean.

Description

The function **`text-input-pane-selection-p`** returns **`t`** if there is a selected region in *pane* and **`nil`** otherwise.

See also

`set-text-input-pane-selection`
`text-input-pane`
`text-input-pane-selected-text`
`text-input-pane-selection`

text-input-pane-set-recent-items *Function*

Summary

Sets the recent items in a **text-input-pane**.

Package

capi

Signature

text-input-pane-set-recent-items *text-input-pane strings where*

Arguments

text-input-pane⇓ A **text-input-pane** with recent items.

strings⇓ A list of strings.

where⇓ One of the keywords **:replace**, **:delete**, **:start** and **:end**, or a non-negative integer.

Description

The function **text-input-pane-set-recent-items** sets the recent items in *text-input-pane*, which must have recent items, that is it must have been created with one of the keyword arguments **:search-field**, **:recent-items** or **:recent-items-name**. *strings* must be a list of strings.

text-input-pane-set-recent-items modifies the recent items according to the argument *where*, which can one of:

:replace The strings replace the recent items in the text-input-pane.

:delete Delete from the recent items any item that matches any of the string (using cl:string-equal).

:start Insert the strings at the beginning of the recent items.

:end Insert the strings at the end of the recent items.

A non-negative integer

Insert the strings at the position indicated by the value. 0 means the same as **:start**. If the integer is greater than the length of the current recent items list, the strings are inserted in the end of the list.

In all cases, if any of the strings is already in the recent-items list (as compared by **cl:string-equal**), it is first deleted from the list. This means that passing strings that already exist just moves them around in the list.

Notes

text-input-pane-set-recent-items is a little more efficient than using **text-input-pane-recent-items** and **(setf text-input-pane-recent-items)** but the different is unlikely to be significant.

text-input-pane-set-recent-items does not return a meaningful value.

See also

text-input-pane
text-input-pane-replace-recent-items
text-input-pane-delete-recent-items
text-input-pane-append-recent-items
text-input-pane-prepend-recent-items

text-input-range *Class*

Summary

The class **text-input-range** is a pane for entering a number in a given range. Typically there are up and down buttons at the side which can used to quickly adjust the value.

Package

capi

Superclasses

titled-object
simple-pane

Initargs

:start	An integer specifying the lowest possible value in the range.
:end	An integer specifying the highest possible value in the range.
:wraps-p	A generalized boolean.
:value	An integer specifying the current value in the pane.
:callback	A function called when the value is changed by the user.
:change-callback	A function called called when the user edits the text in the pane.
:callback-type	The type of arguments passed to the callback.

Accessors

text-input-range-start
text-input-range-end
text-input-range-wraps-p
text-input-range-value
text-input-range-callback
text-input-range-change-callback
text-input-range-callback-type

Description

The class **text-input-range** provides numeric input of integers in a given range (some systems refer to this a spinner or spin-box).

The range is controlled by the **:start** and **:end** initargs. *start* defaults to **0** and *end* defaults to **10**. The initial value is set with the argument *value* (which defaults to **0**).

wraps-p controls what happens if the user presses the up or down button until the start or end is reached. If *wraps-p* is **nil**, then it stops at the limit. If *wraps-p* is true then it wraps around to the other end. The default value of *wraps-p* is **nil**.

callback, if non-nil, should be a function to be called whenever the value is changed by the user. The arguments to callback are specified by *callback-type* (see the **callbacks** class for details of possible values, noting that the "data" is the value and the "item" is the pane itself). The default *callback-type* is **(:item :data)**. Note that, if the value is changed by the user editing the text, then *change-callback*, if supplied, is called as well.

change-callback, if non-nil, should be a function of four arguments, to be called when the user edits the text in the pane. It should have this signature:

change-callback *string pane interface caret-position*

where the arguments are interpreted just as for the change-callback of **text-input-pane**. Note that editing of the text may or may not change the value in the **text-input-range** (that is, what **text-input-range-value** returns). If the value does change, then *callback* is called too.

Notes

On Cocoa, *change-callback* is not called for a cursor move only.

Examples

```
(capi:contain
 (make-instance 'capi:text-input-range
                :start 0
                :end 100
                :value 42))
```

```
(example-edit-file "capi/elements/text-input-range")
```

See also

text-input-pane
text-input-choice
option-pane

titled-menu-object *Class*

Summary

A deprecated class retained only for backward compatibility.

Package

capi

Superclasses

menu-object

Subclasses

menu

menu-component
menu-item

Description

The class **titled-menu-object** is deprecated, and left only for backward compatibility. Use **menu-object** instead.

See also

menu-object

titled-object *Abstract Class*

Summary

A mixin class which provides support for decorating a pane with a title and a message.

Package

capi

Superclasses

standard-object

Subclasses

interface
layout
title-pane
display-pane
text-input-pane
toolbar
button-panel
list-panel
option-pane
progress-bar
output-pane
slider

Initargs

:title	A title string for the pane (or **nil**).
:title-args	Initargs to the title **make-instance**.
:title-font	The font used for the title.
:title-position	The position of the title.
:title-adjust	How to adjust the title relative to the pane.
:title-gap	The gap between the title and the pane.
:message	A message string for the pane (or **nil**).
:mnemonic-title	A string specifying the title and a mnemonic. Applies only to the subclasses specified below.

:message-gap The gap between the message and the pane.

Accessors

titled-object-title
titled-object-title-font
titled-object-message
titled-object-message-font

Description

The abstract class **titled-object** is a mixin class which provides support for decorating a pane with a title (a piece of text positioned next to the pane) and with a message (a piece of text below the pane).

The titled pane makes its title decoration from a **title-pane** and the message decoration from a **message-pane**.

The *text* of the **title-pane** is passed via the **titled-object** initarg *title* and the *text* of the **message-pane** is passed via the **titled-object** initarg *message*.

The initargs and font for the **title-pane** are passed via the **titled-object** initargs *title-args* and *title-font* respectively.

title-gap specifies the size in pixels of the gap between the title and the pane. The default value of *title-gap* is 3.

For subclasses other than **interface**, the font used for the *message* can be found by **titled-object-message-font** and set by **(setf titled-object-message-font)**.

message-gap specifies the size in pixels of the gap between the message and the pane. The default value of *message-gap* is 3.

The message is always placed below the pane, but the title's position can be adjusted by specifying *title-position* which can be any of the following.

:left Place the title to the left of the pane.

:right Place the title to the right of the pane.

:top Place the title above the pane.

:bottom Place the title below the pane.

:frame Place the title in a frame (like a groupbox) around the pane.

The *title-adjust* slot is used to adjust the title so that it is left justified, right justified or centered. The value of *title-adjust* can be any of the values accepted by the function **pane-adjusted-offset**, which are **:left**, **:right**, **:top**, **:bottom**, **:center** and **:centre**.

Note: *title-adjust* cannot handle both x and y. It is designed for cases like this:

```
(capi:contain
 (make-instance 'capi:list-panel
                :items '(1 2 3 4 5)
                :title "Temp"
                :title-position :left
                :title-adjust :center
                :title-args
                '(:visible-min-width (:character 12))))
```

mnemonic-title offers an alternate way to provide the pane's title, and with a mnemonic. It takes effect only for **button-panel**, **list-panel**, **list-view**, **option-pane**, **output-pane**, **progress-bar**, **scroll-bar**, **slider**, **text-input-pane**, **text-input-range**, **tree-view** and their subclasses, and is interpreted as described for **menu**.

Note: titles and mnemonic titles can now be added in a **grid-layout**.

Compatibility note

titled-object corresponds to the LispWorks 4.1 class **titled-pane**. For backwards compatibility the accessors **titled-pane-title** and **titled-pane-message**, including setf methods, are provided. These simply trampoline to **titled-object-title** and **titled-object-message**, and may not be supported in future releases.

Examples

Try each of these examples to see some of the effects that titled panes can produce. Note that **text-input-pane** is a subclass of **titled-object**, and that it has a default *title-position* of **:left**.

```
(capi:contain (make-instance 'capi:text-input-pane))
```

```
(capi:contain (make-instance 'capi:text-input-pane
                              :title "Enter some text:"))
```

```
(capi:contain (make-instance
               'capi:text-input-pane
               :title "Enter some text:"
               :title-position :top))
```

```
(capi:contain (make-instance 'capi:text-input-pane
                              :title "Enter some text:"
                              :title-position :top
                              :title-adjust :center))
```

```
(capi:contain (make-instance 'capi:text-input-pane
                             :title "Enter some text:"
                             :title-position :top
                             :title-adjust :right))
```

```
(capi:contain (make-instance 'capi:text-input-pane
                              :message "A message"))
```

```
(capi:contain (make-instance 'capi:text-input-pane
                              :message "A message"
                              :title "Enter some text:"))
```

```
(capi:contain (make-instance 'capi:text-input-pane
                             :title "Enter some text:"
                             :title-args
                              '(:foreground :red)))
```

See also

message-pane
title-pane
3.1.4.1 Controlling Mnemonics
3.3 Specifying titles

titled-pinboard-object *Class*

Summary

A pinboard object with a title.

Package

capi

Superclasses

pinboard-object
titled-object

Subclasses

image-pinboard-object

Description

The class **titled-pinboard-object** provides a pinboard object with a title. The title is regarded as part of the object in geometry calculations.

Notes

titled-pinboard-object does not allow the value **:frame** for the **titled-object** initarg *title-position*. The values **:top**, **:bottom**, **:left** and **:right** are allowed.

Examples

This example creates three instances of **titled-pinboard-object** and one of **item-pinboard-object**, all with with a yellow background. Note that:

1. The title does not have the yellow background in the **titled-pinboard-object**, as opposed to the **item-pinboard-object**. To specify the title background, we pass it in the *title-args*.
2. The width of the title area is determined by the title, but passing **:visible-min-width** (and other geometric hints) can be used to override this.
3. Setting the **titled-object-title** of the **titled-pinboard-object** does not reset its width.

```
(setq tpo1 (make-instance 'capi:titled-pinboard-object
                           :graphics-args
                           '(:background :yellow)
                           :x 10 :y 10
                           :width 150 :height 20
                           :title "Short"
                           :title-position :left
                           :title-args
                           '(:background :red ))
      tpo2  (make-instance 'capi:titled-pinboard-object
                           :graphics-args
                           '(:background :yellow)
                           :x 10 :y 40
```

```
                              :width 150 :height 20
                              :title "Long title"
                              :title-position :left)
         tpo3  (make-instance 'capi:titled-pinboard-object
                              :graphics-args
                              '(:background :yellow)
                              :x 10 :y 70
                              :width 150 :height 20
                              :title "Short"
                              :title-position :left
                              :title-args
                              '(:visible-min-width 100))
         ipo  (make-instance 'capi:item-pinboard-object
                             :graphics-args
                             '(:background :yellow)
                             :x 10 :y 100
                             :width 150 :height 20
                             :text "Item Pinboard" ))
   (setq pl (capi:contain
             (make-instance 'capi:pinboard-layout
                            :visible-min-width  200
                            :visible-min-height 200
                            :description
                            (list tpo1 tpo2 tpo3 ipo))))

   (capi:apply-in-pane-process
    pl
    #'(lambda()
        (setf (capi:titled-object-title tpo1)
              "Longer...")))
```

See also

item-pinboard-object
12.3 Creating graphical objects

title-pane *Class*

Summary

This class provides a pane that displays a single line of text.

Package

capi

Superclasses

titled-object
simple-pane

Subclasses

message-pane

Initargs

:text The text to appear in the title pane.

Accessors

title-pane-text

Description

The class **title-pane** provides a pane that displays a single line of text.

The most common use of **title-pane** is as a title decoration for a pane, and so the class **titled-object** is provided as a class that supports placing title panes around itself.

A **title-pane** with *text* "Title" is created automatically when a **titled-object** is created with *title* "Title".

By default, a **title-pane** is constrained so that it cannot resize (that is, the values of *visible-max-width* and *visible-max-height* are **t**). This can be overridden by passing **:visible-max-width nil** or **:visible-max-height nil**.

Notes

title-pane does not support the **:pane-menu** initarg on Microsoft Windows. If you need interaction, use **display-pane** or **text-input-pane** with **:pane-menu** and **:enabled :read-only**.

Examples

```
(setq title-pane (capi:contain
                  (make-instance
                   'capi:title-pane
                   :text "This is a title pane")))

(capi:apply-in-pane-process
 title-pane #'(setf capi:title-pane-text)
 "New title" title-pane)
```

See also

display-pane
text-input-pane
editor-pane
3 General Properties of CAPI Panes

toolbar *Class*

Summary

This class provides a pane containing toolbar buttons and panes.

Package

capi

Superclasses

collection
simple-pane
titled-object
toolbar-object

Initargs

:dividerp	If **t**, a divider line is drawn above the toolbar, to separate it from the menu bar. The default value is **nil**.
:images	A list of images.
:callbacks	A list of callback functions.
:names	A list of names.
:texts	A list of strings.
:tooltips	A list of tooltip strings used on Microsoft Windows.
:button-width	The width of the toolbar buttons.
:button-height	The height of the toolbar buttons.
:stretch-text-p	A generalized boolean.
:image-width	The width of images in the toolbar.
:image-height	The height of images in the toolbar.
:default-image-set	An optional **image-set** object which can be used to specify images. See **5.10.4 image-list, image-set and image-locator** for more details.
:flatp	A generalized boolean.

Readers

toolbar-flat-p

Description

The class **toolbar** inherits from **collection**, and therefore has a list of *items*. It behaves in a similar manner to **push-button-panel**, which inherits from **choice**.

The *items* argument may be used to specify a mixture of **toolbar-button**s and **toolbar-component**s, or it may contain arbitrary objects as items. The list may also contain CAPI panes, which will appear within the toolbar. This is typically used with **text-input-pane**, **option-pane**, and **text-input-choice**.

For items that are not toolbar buttons or toolbar components, a toolbar button is automatically created, using the appropriate elements of the *images*, *callbacks*, *names*, *texts* and *tooltips* lists. If no image is specified, the item itself is used as the image. For more information on acceptable values for *images*, see **toolbar-button**.

Each of the *images*, *callbacks*, *names*, *texts* and *tooltips* lists should be in one-to-one correspondence with the items. Elements of these lists corresponding to **toolbar-button** items or **toolbar-component** items are ignored.

Note: :tooltips is now deprecated. Use the **interface** *help-callback* with *help-key* **:tooltip** instead.

All toolbar buttons within the item list behave as push buttons. However, toolbar button components may have **:single-selection** or **:multiple-selection** interaction. See **toolbar-component** for further details.

button-width and *button-height* specify the size of each button in the toolbar. If a button contains text and *stretch-text-p* is true, then the button stretches to the width of the toolbar if needed.

images, if supplied, must specify images all of the same size.

image-width and *image-height* must match the sub-image dimensions in *default-image-set* or the dimensions of the *images*.

flatp specifies whether the toolbar is 'flat' on Cocoa. If *flatp* is true, then the buttons do not have a visible outline until the user moves the mouse over them. *flatp* is only implemented on Cocoa. (On Microsoft Windows, all toolbars are flat. On Motif, no toolbar is flat.) The default value of *flatp* is **:default**.

Notes

1. **text-input-pane**, **option-pane**, and **text-input-choice** and so on cannot contain titles when embedded in a **toolbar**.
2. Rather than creating a **toolbar** explicitly you can add an *interface toolbar* by supplying the **interface** initarg **:toolbar-items**. This has the advantages that the toolbar is automatically positioned correctly within the window and has platform-standard behavior such as folding on Cocoa.

See also

collection
image-set
push-button-panel
toolbar-component
5.10.4 image-list, image-set and image-locator
9.9 Non-standard toolbars
13.10 Working with images

toolbar-button *Class*

Summary

This class is used to create instances of toolbar buttons.

Package

capi

Superclasses

item
toolbar-object

Initargs

:callback	A function that is called when the user presses the toolbar button and *popup-interface* is non-nil.
:image	Specifies the image to use for the toolbar button.
:selected-image	Specifies the image to use for the toolbar button when it is selected.
:tooltip	An optional string which is displayed, on Microsoft Windows, when the mouse moves over the button. **:tooltip** is deprecated.
:help-key	An object used for lookup of help. Default value **t**.
:remapped	Links the button to a menu item.
:dropdown-menu	A **menu** or **nil**.
:dropdown-menu-function	A function of no arguments, or **nil**.
:dropdown-menu-kind	One of the keywords **:button**, **:only** and **:delayed**.
:popup-interface	An **interface** or **nil**.

Accessors

toolbar-button-image
toolbar-button-selected-image
toolbar-button-dropdown-menu
toolbar-button-dropdown-menu-function
toolbar-button-dropdown-menu-kind
toolbar-button-popup-interface

Readers

help-key

Description

The class **toolbar-button** is used to create instances of toolbar buttons.

Toolbar buttons may be placed within toolbars and toolbar components. However, there is usually no need to create toolbar buttons explicitly; instead, the *callbacks* and *images* arguments to **toolbar** or **toolbar-component** can be used. To add tooltips, use the **interface** *help-callback* with *help-key* **:tooltip**.

In addition, an **interface** can have its own toolbar buttons, specified by its *toolbar-items*. No **toolbar** object is explicitly needed in that situation.

image and *selected-image* may each be one of the following:

A pathname or string — This specifies the filename of a file suitable for loading with **load-image**. Currently this must be a bitmap file.

A symbol — The symbol must either have been previously registered by means of a call to **register-image-translation**, or be one of the following symbols, which map to standard images: **:std-cut**, **:std-copy**, **:std-paste**, **:std-undo**, **:std-redo**, **:std-delete**, **:std-file-new**, **:std-file-open**, **:std-file-save**, **:std-print**, **:std-print-pre**, **:std-properties**, **:std-help**, **:std-find** and **:std-replace**.

On Microsoft Windows, the following symbols are also recognized for view images: **:view-large-icons**, **:view-small-icons**, **:view-list**, **:view-details**, **:view-sort-name**, **:view-sort-size**, **:view-sort-date**, **:view-sort-type**, **:view-parent-folder**, **:view-net-connect**, **:view-net-disconnect** and **:view-new-folder**.

Also on Microsoft Windows, these symbols are recognized for history images: **:hist-back**, **:hist-forward**, **:hist-favorites**, **:hist-addtofavorites** and **:hist-viewtree**.

An **image** object — For example, as returned by **load-image**.

An image locator object.

This allows a single bitmap to be created which contains several button images side by side. See **make-image-locator** for more information. On Microsoft Windows, this also allows access to bitmaps stored as resources in a DLL.

An integer — This is a zero-based index into the *default-image-set* of the toolbar or toolbar component in which the toolbar button is used.

Each image should be of the correct size for the toolbar. By default, this is 16 pixels wide and 16 pixels high.

help-key is interpreted as described for **element**.

remapped, if non-nil, should match the *name* of a **menu-item** in the same interface as the button. Then, the action of pressing the button is remapped to selecting that **menu-item** and calling its *callback*. The default value of *remapped* is **nil**.

Toolbar buttons can be made with an associated dropdown menu by passing the **:dropdown-menu** or **:dropdown-menu-function** initargs.

If *dropdown-menu* is non-nil then it should be a **menu** object to display for the button.

If *dropdown-menu-function* is non-nil then it should be a function which will be called with the **toolbar-button** as its single argument. It should return a **menu** object to display for the button.

dropdown-menu-kind can have the following values:

:button There is a separate smaller button for the dropdown menu next to the main button.

:only There is no main button, only the smaller button for the dropdown.

:delayed There is only one button and the menu is displayed when the user holds the mouse down over the button for some short delay. If the user clicks on the button then the normal *callback* is called.

Note: *dropdown-menu-kind* is not supported for toolbar buttons in the **interface** *toolbar-items* list.

popup-interface, if non-nil, should be an **interface**. When the user clicks on the toolbar button, the interface *popup-interface* is displayed near to the button. The normal *callback* is not called, but you can detect when the interface appears by using its *activate-callback*. *popup-interface* is useful for popping up windows with more complex interaction than a menu can provide. The default value of *popup-interface* is **nil**.

Note: *popup-interface* is not supported for toolbar buttons in the **interface** *toolbar-items* list.

Toolbar buttons can display text, which should be in the *data* or *text* slot inherited from **item**.

Note: display of text in toolbar buttons is implemented only on Motif and Cocoa.

Examples

A callback function:

```
(defun do-redo (data interface)
  (declare (ignorable data interface))
  (capi:display-message "Doing Redo"))
```

A simple interface:

```
(capi:define-interface redo ()
  ()
  (:panes
   (toolbar
    capi:toolbar
    :items
    (list
     (make-instance
      'capi:toolbar-component
      :items
      (list (make-instance
             'capi:toolbar-button
             ;; remap it to the menu item
             :remapped  'redo-menu-item
             :image :std-redo))))))
  (:menu-bar a-menu)
  (:menus
   (a-menu
    "A menu"
```

```
        (("Redo" :name 'redo-menu-item
                 :selection-callback 'do-redo
                 :accelerator "accelerator-y"))))
      (:layouts
       (main
        capi:row-layout
        '(toolbar)))
      (:default-initargs
       :title "Redo"))
```

In this interface, pressing the toolbar button invokes the menu item callback:

```
(capi:display (make-instance 'redo))
```

This last example illustrates the use of **`:selected-image`**.

```
(capi:contain
 (make-instance
  'capi:toolbar
  :items
  (list
   (make-instance
    'capi:toolbar-component
    :interaction :multiple-selection
    :items
    (list (make-instance 'capi:toolbar-button
                         :image 0
                         :selected-image 1))
    ))))
```

See also

item
make-image-locator
menu-item
toolbar
toolbar-component
3.12 Tooltips
9 Adding Toolbars
13.10 Working with images

toolbar-component *Class*

Summary

A toolbar component is used to group several toolbar buttons together. Each component is separated from the surrounding components and buttons. Toolbar components are choices, and may be used to implement toolbars on which groups of buttons have single-selection or multiple-selection functionality.

Package

capi

Superclasses

toolbar-object

choice

Initargs

:images	A list of images, in one-to-one correspondence with the items.
:callbacks	A list of callback functions, in one-to-one correspondence with the items.
:names	A list of names, in one-to-one correspondence with the items.
:texts	A list of strings, in one-to-one correspondence with the items.
:tooltips	A list of tooltip strings, in one-to-one correspondence with the items.
:default-image-set	An optional **image-set** object which can be used to specify images. See **5.10.4 image-list, image-set and image-locator** for more details.
:selection-function	A function to dynamically compute the selection.
:selected-item-function	A function to dynamically compute the selected item.
:selected-items-function	A function to dynamically compute the selected items.

Description

The class **toolbar-component** inherits from **choice**, and hence has a list of *items*. Its behavior is broadly similar to **button-panel**.

items may be used to specify a mixture of **toolbar-button**s and **toolbar-component**s, or may contain arbitrary objects as items. The list may also contain CAPI panes, which will appear within the toolbar. This is typically used with **text-input-pane**, **option-pane**, and **text-input-choice**.

For items that are not toolbar buttons or toolbar components, a toolbar button is automatically created, using the appropriate elements of the *images*, *callbacks*, *names*, *texts* and *tooltips* lists. If no image is specified, the item itself is used as the image. For more information on acceptable values for images, see **toolbar-button**. Elements of *images*, *callbacks*, *names*, *texts* and *tooltips* corresponding to **toolbar-button** items or **toolbar-component** items are ignored.

No more than one of *selection-function*, *selected-item-function* and *selected-items-function* should be non-nil. Each defaults to **nil**. If one of these is non-nil, it should be a function which is called before the **toolbar-component** is displayed and when **update-toolbar** is called and which determines which items are selected. The function takes a single argument, which is the **interface** of the **toolbar-component**.

selection-function, if non-nil, should return a list of indices suitable for passing to the **choice** accessor **(setf choice-selection)** .

selected-item-function, if non-nil, should return an object which is an item in the **toolbar-component**, or is equal to such an item when compared by the **toolbar-component**'s *test-function* and *key-function*.

selected-items-function, if non-nil, should return a list of such objects.

Examples

```
(example-edit-file "capi/elements/toolbar")
```

See also

toolbar

toolbar-button
3.12 Tooltips
9 Adding Toolbars
13.10 Working with images

toolbar-object *Class*

Summary

This is a common superclass of all toolbar objects.

Package

capi

Superclasses

standard-object

Subclasses

toolbar
toolbar-button
toolbar-component

Initargs

:enabled	If **t**, the toolbar object is enabled.
:enabled-function	A function determining the enabled state.

Accessors

simple-pane-enabled
toolbar-object-enabled-function

Description

The class **toolbar-object** is a common superclass of all toolbar objects.

Any toolbar object may be disabled, by setting its *enabled* slot to **nil**. Disabling a toolbar or toolbar component prevents the user from interacting with any buttons contained in it.

All toolbar objects may also have an *enabled-function* specified. This is called whenever **update-toolbar** is called. If it returns **t**, the toolbar object will be enabled; if it returns **nil**, the object will be disabled.

Notes

The function *enabled-function* should not display a dialog or do anything that may cause the system to hang. In general this means interacting with anything outside the Lisp image, including files, databases and so on.

See also

toolbar
toolbar-button

toolbar-component
update-toolbar
9 Adding Toolbars

top-level-interface *Generic Function*

Summary

Returns the top level interface containing a specified pane.

Package

capi

Signature

top-level-interface *pane*

Arguments

pane⇓ A **simple-pane** or a **pinboard-object**.

Description

The generic function **top-level-interface** returns the top level interface that contains *pane*.

See also

top-level-interface-p
interface
element
3.7 Hierarchy of panes

top-level-interface-color-mode *Accessor*

Summary

A value that indicates the color mode of a top level interface.

Package

capi

Signature

top-level-interface-color-mode *interface* **=>** *color-mode*

setf (top-level-interface-color-mode *interface***)** *color-mode* **=>** *color-mode*

Arguments

interface⇓	An **interface** instance.
color-mode	**nil**, a keyword or a string.

Values

color-mode	**nil**, a keyword or a string.

Description

The accessor **top-level-interface-color-mode** reads or sets the Appearance of *interface* on Cocoa. **top-level-interface-color-mode** has no effect on other platforms.

If *color-mode* is **nil** then *interface* is displayed in the Appearance specified by the System Preferences.

Otherwise, when *color-mode* is non-nil, it specifies that *interface* has its own Appearance, overriding the System Preferences.

When *color-mode* is a keyword, it must be one of the keywords in the following table, and it is mapped to the specified Cocoa appearance name.

Color mode keywords mapping to Cocoa appearances

Keyword	Cocoa appearance name
:light or **:aqua**	**NSAppearanceNameAqua**
:dark or **:dark-aqua**	**NSAppearanceNameDarkAqua**

Any other keyword will signal an error.

When *color-mode* is a string, it specifies the name of a Cocoa appearance, and it is looked up by calling the **appearanceNamed:** method of the Cocoa **NSAppearance** class. It is your responsibility to pass a valid string. If **appearanceNamed:** fails to find the appearance, a warning is signalled and the *color-mode* is ignored.

top-level-interface-color-mode-callback is called when macOS changes the Appearance.

top-level-interface-color-mode returns the desired color mode. Call **top-level-interface-dark-mode-p** to determine if *interface* is currently in dark mode.

Examples

For an example of using *color-mode* and *color-mode-callback*, see:

```
(example-edit-file "capi/applications/interface-color-mode")
```

See also

top-level-interface-color-mode-callback
top-level-interface-dark-mode-p

top-level-interface-dark-mode-p *Function*

Summary

Determines if a top level interface is displayed in dark mode.

Package

capi

Signature

top-level-interface-dark-mode-p *interface* **=>** *dark-mode-p*

Arguments

interface⇓	An **interface** instance.

Values

dark-mode-p	A Boolean.

Description

The function **top-level-interface-dark-mode-p** returns true if *interface* is currently displayed in dark mode and false otherwise. If *interface* is not displayed, **top-level-interface-dark-mode-p** returns false.

On Cocoa, *interface* is in dark mode if the name of its effective Appearance contains "dark" (ignoring case). That works for the standard appearances, but may not work for user defined ones. On GTK+ and Microsoft Windows, *interface* is in dark mode if its default background color is dark, which is checked summing the RGB values and comparing with 1.5.

Examples

```
(example-edit-file "capi/applications/interface-color-mode")
```

See also

interface
top-level-interface-color-mode

top-level-interface-display-state *Generic Function*

Summary

Returns a value which indicates how the top level interface is displayed.

Package

capi

Signature

`top-level-interface-display-state` *interface* **`=>`** *display-state*

Arguments

interface⇓	A top level interface or dialog window.

Values

display-state	One of **`:normal`**, **`:maximized`**, **`:iconic`**, **`:hidden`** or **`:full-screen`**.

Description

Top level interfaces and dialogs can be manipulated by the user, such as being iconifed or maximized. The program can manipulate these windows too. The generic function **`top-level-interface-display-state`** returns a value that indicates the current state of the interface *interface*. The following values can be returned:

`:normal`	The window is visible and has its normal size.
`:maximized`	The window is visible and has been maximized.
`:iconic`	The window is visible as an icon.
`:hidden`	The window is not visible.
`:full-screen`	The window is full screen (only supported on macOS 10.7 and later). This value is only applicable when the *window-styles* list contains the keyword **`:can-full-screen`**.

These values can also be passed as the **`:display-state`** initarg when making a top level interface.

In addition, the function **`(setf top-level-interface-display-state)`** can be used to change the state of a top level interface. The value can be set to one of the above, or to **`:restore`** if the current state is **`:iconic`** or **`:hidden`**. When set to **`:restore`**, the state will become **`:normal`** or **`:maximized`** depending on how the interface was visible in the past.

See also

`top-level-interface-p`
`top-level-interface-geometry`
`set-top-level-interface-geometry`
`interface`
7 Programming with CAPI Windows

top-level-interface-geometry *Function*

Summary

Returns the geometry of the top level interface.

Package

`capi`

Signature

top-level-interface-geometry *interface* **=>** *tx*, *ty*, *twidth*, *theight*

Arguments

interface⇓ An interface.

Values

tx⇓, *ty*⇓, *twidth*, *theight*

Integers.

Description

The function **top-level-interface-geometry** returns the coordinates of the given interface in a form suitable for use as the **:best-x**, **:best-y**, **:best-width** and **:best-height** initargs to **interface**. The value of *interface* should be a top level interface.

tx and *ty* are measured from the top-left of the screen rectangle representing the area of the primary monitor (the primary screen rectangle).

Notes

On Cocoa, the result does not account for the size of the interface toolbar, if present in *interface*.

Examples

```
;; Define and display an interface.
(capi:define-interface test ()
  ()
  (:panes (panel capi:list-panel)))

(setq int (capi:display (make-instance 'test)))
;; Now manually position the interface somewhere.

;; Find where the interface is.
(multiple-value-setq (tx ty twidth theight)
 (capi:top-level-interface-geometry int))

;; Now manually close the interface.
;; Create a new interface in the same place.
(setq int
      (capi:display
       (make-instance
        'test
        :best-x tx
        :best-y ty
        :best-width twidth
        :best-height theight)))
```

See also

top-level-interface-p
top-level-interface-display-state
set-top-level-interface-geometry
interface
4.3 Support for multiple monitors

7 Programming with CAPI Windows
11.6 Querying and modifying interface geometry

top-level-interface-geometry-key *Generic Function*

Summary

Determines where the geometry of an interface is saved.

Package

`capi`

Signature

`top-level-interface-geometry-key` *interface* **`=>`** *key*, *product-name*

Arguments

interface⇓	A top level interface.

Values

key⇓	A symbol.
product-name⇓	A symbol, a string or a list of strings.

Description

The generic function **`top-level-interface-geometry-key`** returns as multiple values a key and a product name, which determine where the geometry of *interface* is saved. The saved geometry is used when displaying a future instance.

The supplied method on **`interface`** returns the class name of *interface* as *key*, and **`nil`** as *product-name*. You can define methods for your interfaces and products.

key must be a symbol.

product-name is used to derive the *product-registry-path*.

product-name can be a symbol which was previously defined to have a registry path by **`(setf sys:product-registry-path)`**.

product-name can alternatively be a string, which is taken directly as *product-registry-path*.

product-name can alternatively be a list of strings, denoting multiple path components. These are concatenated together with the appropriate separator for the platform to give *product-registry-path*.

The geometry of *interface* is saved at the path which is constructed by concatenating (with appropriate separators) these values:

user-path
product-registry-path
`"Environment"`
`(symbol-package #KEY)`
`(symbol-name #KEY)`

where *user-path* is the registry branch HKEY_CURRENT_USER on Microsoft Windows and the home directory on other platforms.

Note: for your interface classes for which you want the geometry to be saved, define a method on **`top-level-interface-save-geometry-p`**.

Note: in an image delivered at delivery level 5, symbol names are removed by default. This breaks the saved geometry mechanism as the registry path is constructed using **`symbol-name`**. To make this work in a level 5 delivered image, explicitly keep the symbol *key*. See the *Delivery User Guide* for details.

See also

`top-level-interface-save-geometry-p`
11.6 Querying and modifying interface geometry

top-level-interface-p *Generic Function*

Summary

The predicate for top level interfaces.

Package

`capi`

Signature

`top-level-interface-p` *pane* **`=>`** *result*

Arguments

pane⇓ A Lisp object.

Values

result A boolean.

Description

The generic function **`top-level-interface-p`** returns true if *pane* is a top level interface.

See also

`top-level-interface`
`top-level-interface-geometry`
`top-level-interface-display-state`
`interface`
`element`
3.7 Hierarchy of panes

top-level-interface-save-geometry-p *Generic Function*

Summary

Return true if the geometry of an **interface** should be saved for use by a future instance.

Package

capi

Signature

top-level-interface-save-geometry-p *interface* **=>** *result*

Arguments

interface⇓ A top level interface.

Values

result A boolean.

Description

The generic function **top-level-interface-save-geometry-p** returns true if the geometry of *interface* should be saved for use by a future instance.

The default method (on **interface**) returns **nil**.

See also

top-level-interface-geometry-key
11.6 Querying and modifying interface geometry

tracking-pinboard-layout *Class*

Summary

A pinboard with automatic highlighting.

Package

capi

Superclasses

pinboard-layout

Description

The class **`tracking-pinboard-layout`** provides a pinboard which tracks mouse movement by highlighting its objects as the mouse cursor moves over them.

This functionality is implemented via a **`:motion`** specification in the *input-model*. Therefore, you may not specify **`:motion`** in the *input-model* of a **`tracking-pinboard-layout`**. See **`output-pane`** for a description of *input-model*.

Examples

```
(example-edit-file "capi/graphics/tracking-pinboard-layout")
```

tree-view *Class*

Summary

A tree view is a pane that displays a hierarchical list of items. Each item may optionally have an image and a checkbox.

Package

`capi`

Superclasses

`choice`
`titled-object`
`simple-pane`

Initargs

`:roots` A list of the root items.

`:children-function` Returns the children of an item and hence defines the hierarchy in the tree.

`:leaf-node-p-function`
Optional function which determines whether an item is a leaf item (that is, has no children). This is useful if it can be computed faster than the *children-function*.

`:retain-expanded-nodes`
Specifies if the tree view remembers whether hidden nodes were expanded.

`:expandp-function` A designator for a function of one argument, or **`nil`**.

`:action-callback-expand-p`
A boolean. The default value is **`nil`**.

`:delete-item-callback`
A function designator for a function of two arguments.

`:right-click-extended-match`
Controls the area within which selection by the mouse right button occurs. Default **`t`**.

`:has-root-line` Controls whether the line and expanding boxes of the root items are drawn. Default **`t`**.

`:checkbox-status` Controls whether the tree has checkboxes. If non-nil, the value should be a non-negative integer less than the length of the image-list, or **`t`**. An integer specifies the default initial status, and **`t`** means the same as **`2`** (that is, by default the checkboxes are checked initially). The default is **`nil`**, meaning no checkboxes. Not implemented on Cocoa.

:checkbox-next-map Controls the change in status when the user clicks on a checkbox. Can be an array, a function or an integer. Default **#(2 2 0)**. Not implemented on Cocoa.

:checkbox-parent-function

Controls the changes in the ancestors when the status of an item is changed. Not implemented on Cocoa.

:checkbox-child-function

Controls the changes in the descendants when the status of an item is changed. Not implemented on Cocoa.

:checkbox-change-callback

A function called when the status of an item is changed interactively. Not implemented on Cocoa.

:checkbox-initial-status

Specifies the initial status of specific items. Not implemented on Cocoa.

:image-function Returns an image for an item.

:state-image-function

Returns a state image for an item.

:image-lists A plist of keywords and **image-list** objects.

:use-images Flag to specify whether items have images. Defaults to **t**.

:use-state-images Flag to specify whether items have state images. Defaults to **nil**.

:image-width Defaults to 16.

:image-height Defaults to 16.

:state-image-width Defaults to *image-width*.

:state-image-height

Defaults to *image-height*.

Accessors

tree-view-roots
tree-view-children-function
tree-view-image-function
tree-view-state-image-function
tree-view-leaf-node-p-function
tree-view-retain-expanded-nodes
tree-view-expandp-function
tree-view-action-callback-expand-p
tree-view-right-click-extended-match
tree-view-has-root-line
tree-view-checkbox-next-map
tree-view-checkbox-parent-function
tree-view-checkbox-child-function
tree-view-checkbox-change-callback
tree-view-checkbox-initial-status

Readers

tree-view-checkbox-status

Description

The class **tree-view** is a pane that displays a hierarchical list of items. Each item may optionally have an image and a checkbox.

The tree view pane allows the user to select between items displayed in a hierarchical list. Although it is a **choice**, only **:single-selection** *interaction* is supported. Use **extended-selection-tree-view** if you need other selection interaction styles.

The hierarchy of items in the tree is defined by the *children-function*, which must be a function taking a single argument (an item) and returning a list of child items. When an item is expanded, whether programmatically, automatically, or in response to a user gesture, the system calculates what children this item has by calling the *children-function* on it.

Both the *roots* and what children the *children-function* returns for an item can be any object. However, the list must not include an object which is **cl:eql** to another object in the tree. To work sensibly it also needs to be consistent over time, that is return the same objects each time it is called, unless the state of the entity that the tree represents changes. It should also be reasonably fast, as the user will be waiting to see the items.

If the tree is supposed to display items that are "the same" in different parts of the tree, you can define a "wrapper", typically **cl:defstruct** with a few slots, and return a list of these wrappers (each pointing to the actual object). This wrapping is also useful for keeping other information related to the display in the tree, for example the string or the image to display, and maybe cache the children.

If *leaf-node-p-function* is not supplied, the *children-function* is also used to decide whether unexpanded nodes are leaf items or not (and hence whether to display the expanding box). If the *children-function* is slow, this may slow significantly the display of large trees. If it is possible to check for the existence of children faster, you should supply *leaf-node-p-function* to avoid this slow down.

The default value of *children-function* is **(constantly false)**, that is no children, and hence only the roots are displayed.

expandp-function controls automatic expansion of nodes (items) in the **tree-view**. By default, initially only the items specified by the *roots* argument are displayed. This initial display can be altered by supplying a function *expandp-function* which allows further items to be displayed. If supplied, *expandp-function* should be a function which is called on the *roots* and is called recursively on the children if it returns true. When the user expands a node, *expandp-function* is called on each newly created child node, which is expanded if this call returns true, and so on recursively. The default value of *expandp-function* is **nil** so that there is no automatic expansion and only the root nodes are visible initially.

The default value of *retain-expanded-nodes* is **t**.

Any item which has children has a small expansion button next to it to indicate that it can be expanded. When the user clicks on this button, the children items (as determined by the children function) are displayed.

If *action-callback-expand-p* is true, then the activate gesture expands a collapsed node, and collapses an expanded node. This expansion and contraction of the node is additional to any supplied *action-callback*.

delete-item-callback is called when the user presses the **Delete** key. Two arguments are passed: the **tree-view** and the selected item *item*. Note that, apart from calling the callback, the system does nothing in response to the **Delete** key. In particular, if you want to remove the selected *item*, *delete-item-callback* needs to do it by changing what the *children-function* returns when called on the parent of *item*. Normally you also need to to call **tree-view-update-item** with *in-parent* = **t** to actually update the tree on the screen.

Note also that in **extended-selection-tree-view** (a subclass of **tree-view**), if the *interaction* was not explicitly changed to **:single-selection**, the second argument to *delete-item-callback* is a list of the selected items (even when only one item is selected).

The *image-function* is called on an item to return an image associated with the item. It can return one of the following:

A pathname or string — This specifies the filename of a file suitable for loading with **load-image**. Currently this must be a bitmap file.

A symbol — The symbol must have been previously registered by means of a call to **register-image-translation**. It can also one of the following symbols, which map to standard images: **:std-cut**, **:std-copy**, **:std-paste**, **:std-undo**, **:std-redo**, **:std-delete**, **:std-file-new**, **:std-file-open**, **:std-file-save**, **:std-print**, **:std-print-pre**, **:std-properties**, **:std-help**, **:std-find** and **:std-replace**.

On Microsoft Windows, the following symbols are also recognized. They map to view images: **:view-large-icons**, **:view-small-icons**, **:view-list**, **:view-details**, **:view-sort-name**, **:view-sort-size**, **:view-sort-date**, **:view-sort-type**, **:view-parent-folder**, **:view-net-connect**, **:view-net-disconnect** and **:view-new-folder**.

Also on Microsoft Windows, these symbols are recognized. They map to history images: **:hist-back**, **:hist-forward**, **:hist-favorites**, **:hist-addtofavorites** and **:hist-viewtree**.

An **image** object — For example, as returned by **load-image**.

An image locator object

This allowing a single bitmap to be created which contains several button images side by side. See **make-image-locator** for more information. On Microsoft Windows, it also allows access to bitmaps stored as resources in a DLL.

An integer — This is a zero-based index into the tree-view's image lists. This is generally only useful if the image list is created explicitly. See **image-list** for more details.

The *state-image-function* is called on an item to determine the state image: an additional optional image used to indicate the state of an item. It can return one of the objects listed above, just as for *image-function*, or **nil** to indicate that there is no state image. See also *checkbox-status*, which overrides the *state-image-function*.

If *image-lists* is specified, it should be a plist containing the following keywords as keys. The corresponding values should be **image-list** objects.

:normal — Specifies an **image-list** object that contains the item images. The *image-function* should return a numeric index into this **image-list**.

:state — Specifies an **image-list** object that contains the state images. The *state-image-function* should return a numeric index into this **image-list**.

If *right-click-extended-match* is **nil**, the mouse right button gesture within the tree view selects an item only when the cursor is on the item. Otherwise, this gesture also selects an item to the left or right of the cursor. The default for *right-click-extended-match* is **t**.

If *has-root-line* is **nil**, the vertical root line and expanding boxes of the root items are not drawn. This is useful in two cases:

- When the tree view needs to be neater. Note that the user does not have a mouse gesture to expand the root item. Normally the programmer would compensate for this by making some other gesture call **(setf tree-view-expanded-p)**.
- If a *children-function* is not supplied, this can be used to create a pane like a list view with checkboxes (see below for details of checkboxes). This pane can be handled as if it is a typical choice, except that setting the items is done by **(setf tree-view-roots)** or by passing **:roots** to **make-instance**. In a typical choice, you would do **(setf collection-items)** or pass **:items** to **make-instance**.

The default for *has-root-line* is **t**.

If the *checkbox-status* is non-nil then the tree view provides an automatic way of using the state images as checkboxes (except on Cocoa where check boxes are not supported). The *state-image* is defaulted to a set of images containing checkboxes and the *state-image-function* is ignored, but each *item* has a status that is a non-negative integer no greater than the number of images in *state-image-list*. The status specifies which image is displayed alongside *item*.

When *item* is expanded in the tree for the first time, the status of each child is set to *item*'s status. The status can be changed interactively by the user:

- Left mouse button on a checkbox changes its status.
- Space changes the status of all selected items.

The status can also be read and set programmatically (see **`tree-view-item-checkbox-status`**).

When the status of an item changes:

- The statuses of its ancestors may change if a *checkbox-parent-function* was supplied.
- The statuses of an items descendants may change if a *checkbox-child-function* was supplied.
- A callback given by *checkbox-callback-function* will be called, if this was supplied.

By default checkboxes have three statuses indicated by images: un-checked(0), gray-checked(1) and checked(2). If an item is checked or un-checked, then all its descendants have the same status. If an item is gray-checked, then its descendants have various statuses. When the status of an item changes, all the descendants of that item change to the same status, and all its ancestors change to gray-checked.

For non-default status-changing behavior, specify *checkbox-next-map*. The value can be:

- An array of statuses. When the user clicks on *item*'s checkbox, the status of *item* is used to index into *checkbox-next-map*, and the status at that index becomes the new status of *item*. For example, with the default checkbox-next-map, checked(0) changes to un-checked(2), gray-checked(1) changes to un-checked(2), and un-checked(2) changes to checked(0).
- A function of two arguments. The first argument is a list of items and the second argument is their current status (and if the items have various statuses, the most common is used). *checkbox-next-map* should return the new status to use.
- An integer: the status is increased by 1, until this integer is reached, at which point the status becomes 0 again.

When the status of an item is changed, the statuses of items above and below it in the tree may also be changed: the system recurses up and down the tree using *checkbox-parent-function* and *checkbox-child-function* respectively.

To recurse upwards, *checkbox-parent-function* is called on the parent with five arguments: the parent, the parent's status, the item, the item's status and an flag which is non-nil if all the items at the same level as the item now have the same status:

`checkbox-parent-function` *parent parent-status item item-status all-items-same-p* **`=>`** *new-parent-status*`,` *recurse-up*`,` *recurse-down*

If *new-parent-status* differs from *parent-status*, then the status of *parent* is set to *new-parent-status*. If *recurse-up* is non-nil, then the system recurses up from parent, and if *recurse-down* is non-nil, the system recurses down. The default *checkbox-parent-function* returns **`(values`** *new-item-status* **`t nil)`** where *new-item-status* is *item-status* if *all-items-same-p* is non-nil and **`1`** otherwise.

To recurse downwards, *checkbox-child-function* is called on each child with four arguments and the results are used similarly to those of *checkbox-parent-function*:

`checkbox-child-function` *child child-status item item-status* **`=>`** *new-child-status*`,` *recurse-up*`,` *recurse-down*

The default *checkbox-child-function* returns **`(values`** *parent-status* **`nil t)`**.

Note: if an item has never been expanded, then it has no children. If an item has been collapsed, then it has children even though they are not currently visible.

checkbox-parent-function and *checkbox-child-function* should not modify the tree in any way.

checkbox-change-callback takes three arguments: the tree, a list of items and their new status:

checkbox-change-callback *tree items new-status*

This is called after the new statuses of *items* and their ancestors and descendants have been resolved.

checkbox-initial-status is used the first time that each specified item, which can be anywhere in the tree, appears. The value is a list of conses of items and their initial statuses, for example **((item1. 2) (item2. 0))**. When *item* is displayed, its status is set from this list or, if item is not specified, from *checkbox-status*. Items are removed from the list when they are displayed and setting the list does not affect the checkbox status of items that have already been displayed. Note that check boxes are not supported on Cocoa.

The default value of *vertical-scroll* in a **tree-view** is **t**.

Notes

1. Since the items of a tree view are not computed until display time, the **choice** initarg **:selected-item** has no effect. See the examples in **interface-display** for a way to set the selected item in a tree view.
2. Although **tree-view** is a subclass of **collection**, it does its own items handling and you must not access its *items* and related slots directly. In particular for **tree-view** do not pass **:items**, **:items-count-function**, **:items-get-function** or **:items-map-function**, and do not use the corresponding accessors.
3. On Microsoft Windows, the system always sets the input focus to the **tree-view** after its *selection-callback* returns. If you need this callback to set the focus elsewhere, call **set-pane-focus** outside the callback, like this:

```
(mp:process-send process
  (list 'capi:set-pane-focus pane))
```

Examples

This example shows how to combine an XML parser with **tree-view** to display an RSS file.

```
(example-edit-file "capi/applications/rss-reader")
```

There are further examples in **20 Self-contained examples**.

See also

choice
extended-selection-tree-view
tree-view-ensure-visible
tree-view-expanded-p
tree-view-item-checkbox-status
tree-view-item-children-checkbox-status
tree-view-update-item
1.2.1 CAPI elements
5 Choices - panes with items
13.10 Working with images
17 Drag and Drop

tree-view-ensure-visible *Function*

Summary

Ensures that an item in a **tree-view** is visible.

Package

capi

Signature

tree-view-ensure-visible *tree-view item*

Arguments

tree-view⇓	A tree view.
item⇓	A displayed item of *tree-view*.

Description

The function **tree-view-ensure-visible** ensures that an item in a tree view is visible, scrolling the tree view if necessary.

Note that *item* must be an item that is displayed in *tree-view*.

See also

tree-view

tree-view-expanded-p *Accessor Generic Function*

Summary

Gets and sets the expanded state of an item in a **tree-view**.

Package

capi

Signature

tree-view-expanded-p *tree-view item* **=>** *value*

setf (tree-view-expanded-p *tree-view item***)** *value* **=>** *value*

Arguments

tree-view⇓	A **tree-view**.
item⇓	An item.

value⇓ A boolean.

Values

value⇓ A boolean.

Description

The accessor generic function **tree-view-expanded-p** is the predicate for whether *item* is expanded in *tree-view*. If item is not in *tree-view*, the function returns **nil**.

(setf tree-view-expanded-p) sets the expanded state of *item* in *tree-view* to *value*. If *item* is not in *tree-view*, the function does nothing.

See also

tree-view

tree-view-item-checkbox-status *Accessor*

Summary

Gets and sets the checkbox status of an item in a **tree-view**.

Package

capi

Signature

tree-view-item-checkbox-status *tree-view item* **=>** *status*

(setf tree-view-item-checkbox-status) *status tree-view item* **=>** *status*

Arguments

tree-view⇓ A tree view.

item⇓ An item.

status⇓ A non-negative integer.

Values

status⇓ A non-negative integer.

Description

The accessor **tree-view-item-checkbox-status** gets and sets the checkbox status of *item* in *tree-view*, except on Cocoa.

(setf tree-view-item-checkbox-status) sets the checkbox status of *item* in *tree-view*. *status* must be an non-negative integer smaller than the number of images in *tree-view*'s *state-image-list*.

See also

tree-view
tree-view-item-children-checkbox-status

tree-view-item-children-checkbox-status *Function*

Summary

Gets the checkbox statuses of a **tree-view** item's children.

Package

capi

Signature

tree-view-item-children-checkbox-status *tree-view item* **=>** *result*

Arguments

tree-view⇓	A **tree-view**.
item⇓	An item.

Values

result⇓	A list of conses (*child* . *status*) where each *child* is a child of *item* and *status* is *child*'s checkbox status.

Description

The function **tree-view-item-children-checkbox-status** returns *item*'s children together with their checkbox statuses, except on Cocoa.

Note that, if *item* has not been expanded in *tree-view*, then it has no children and *result* will be **nil**.

See also

tree-view
tree-view-item-checkbox-status

tree-view-update-an-item *Generic Function*

Summary

Updates an item in a **tree-view**.

Package

capi

Signature

tree-view-update-an-item *tree-view item in-parent*

Arguments

tree-view⇓ A **tree-view**.

item⇓ A Lisp object.

in-parent⇓ A boolean.

Description

The generic function **tree-view-update-an-item** is a synonym for **tree-view-update-item**, with the same meaning for *tree-view*, *item* and *in-parent*.

Notes

tree-view-update-an-item is deprecated. Please use **tree-view-update-item** instead.

See also

tree-view
tree-view-update-item

tree-view-update-item *Function*

Summary

Updates an item in a **tree-view**.

Package

capi

Signature

tree-view-update-item *tree-view item in-parent*

Arguments

tree-view⇓ A **tree-view**.

item⇓ An item.

in-parent⇓ A boolean.

Description

The function **tree-view-update-item** updates the item *item* in *tree-view*. This includes recomputing the text, images and children of *item*. This is useful when the data in *tree-view* changes, but the entire tree does not need recomputing.

When *in-parent* is non-nil, **tree-view-update-item** updates the children of the parent of *item*. This is useful when *item* is actually removed from *tree-view*, causing the children of its parent to be re-positioned.

See also

tree-view

undefine-menu *Macro*

Summary

Undefines a menu.

Package

capi

Signature

undefine-menu *function-name* **&rest** *args*

Arguments

function-name⇓	A symbol.
args⇓	Ignored extra arguments.

Description

The macro **undefine-menu** undefines a menu named *function-name* that was created with **define-menu**. *args* are ignored.

See also

define-menu
menu

unhighlight-pinboard-object *Function*

Summary

Removes the highlighting from a **pinboard-object**.

Package

capi

Signature

unhighlight-pinboard-object *pinboard object* **&key** *redisplay* **=>** *was-highlighted-p*

Arguments

pinboard⇓	A **pinboard-layout**.
object⇓	A **pinboard-object**.

redisplay⇓ A generalized boolean.

Values

was-highlighted-p⇓ A boolean.

Description

The function **`unhighlight-pinboard-object`** removes the highlighting from a pinboard object if necessary, and then if *redisplay* is non-nil it redisplays it. The default value of *redisplay* is **`t`**.

pinboard should be the **`pinboard-layout`** of *object*.

To highlight a pinboard object use **`highlight-pinboard-object`**.

The returned value *was-highlighted-p* is true if *object* was highlighted before the call.

See also

`highlight-pinboard-object`
`pinboard-object`

uninstall-postscript-printer *Function*

Summary

Uninstalls a Postscript printer definition.

Package

`capi`

Signature

`uninstall-postscript-printer` *name* **`&key`** *if-does-not-exist deletep*

Arguments

name⇓ A string.

if-does-not-exist⇓ One of **`nil`** or **`:error`**.

deletep⇓ A boolean.

Description

The function **`uninstall-postscript-printer`** uninstalls a PostScript printer definition for the given device *name*.

This applies only on GTK+ and Motif.

if-does-not-exist controls what happens if the named printer does not exist. The default value is **`:error`**.

deletep, if true, causes the printer to be removed for subsequent sessions as well as the current session, by deleting the file on the disk. The default value of *deletep* is **`nil`**.

See also

[**`install-postscript-printer`**]
[**16.7 Printing on Motif**]

unmap-typeout *Function*

Summary

Removes a **`collector-pane`** that **`map-typeout`** had made visible.

Package

`capi`

Signature

`unmap-typeout` *collector-pane*

Arguments

collector-pane⇓	A **`collector-pane`**.

Description

The function **`unmap-typeout`** switches *collector-pane* out from its switchable layout, and brings back the pane that was there before **`map-typeout`** was called.

See also

`map-typeout`
`with-random-typeout`
`collector-pane`

update-all-interface-titles *Function*

Summary

Updates interface window titles.

Package

`capi`

Signature

`update-all-interface-titles`

Description

The function **`update-all-interface-titles`** can be used to update all the **`interface`** window titles when needed.

This is useful when **interface-extend-title** may return a new, different, value.

update-all-interface-titles calls **update-screen-interface-titles** on all the screens.

See also

interface-extend-title
update-screen-interface-titles

update-drawing-with-cached-display
update-drawing-with-cached-display-from-points *Functions*

Summary

Updates the drawing using the cached display.

Package

capi

Signatures

update-drawing-with-cached-display *pane* **&optional** *x y width height*

update-drawing-with-cached-display-from-points *pane x1 y1 x2 y2* **&key** *extend extend-x extend-y*

Arguments

pane⇓ An **output-pane**.

x⇓, *y*⇓, *width*⇓, *height*⇓

Real numbers.

x1⇓, *y1*⇓, *x2*⇓, *y2*⇓, *extend*⇓, *extend-x*⇓, *extend-y*⇓

Real numbers.

Description

The functions **update-drawing-with-cached-display** and **update-drawing-with-cached-display-from-points** update the drawing using the cached display, indicating the rectangle in which the *temp-display-callback* (argument to **start-drawing-with-cached-display**) needs to draw.

These functions must be called in the scope of **start-drawing-with-cached-display** or **output-pane-free-cached-display**. Calls outside this scope have no effect.

pane is the output pane to update. The other arguments specify the rectangle to be updated. The arguments are used in three ways: first they cause an implicit call to **invalidate-rectangle** with the appropriate arguments, secondly they define a mask that is used when calling the *temp-display-callback*, and third they provide arguments that are passed to the *temp-display-callback*.

In the case of **update-drawing-with-cached-display**, the arguments specify the rectangle in the standard way (the same way that they are passed to the *display-callback*). *x* and *y* default to 0, *width* defaults to the width of *pane* minus *x*, and *height* defaults to the height of *pane* minus *y*.

In the case of **update-drawing-with-cached-display-from-points**, the arguments specify two points, (*x1*,*y1*) and (*x2*,*y2*), which are corners of a rectangle. This rectangle is then extended horizontally in both directions by *extend-x*, and extended vertically in both directions by *extend-y*. *extend-x* and *extend-y* default to *extend*, which defaults to 0. The final result is:

```
x = (- (min x1 x2) extend-x)
y = (- (min y1 y2) extend-y)
width = (+ (- (max x1 x2) x) extend-x)
height = (+ (- (max y1 y2) y) extend-y)
```

Both *extend-x* and *extend-y* default to *extent*, which itself defaults to 0.

Notes

Omitting the rectangle (that is, calling **update-drawing-with-cached-display** with only *pane*) causes all of the pane to be redisplayed each time. On slow displays, that may cause the display to be sluggish. On Windows and Cocoa with the normal settings, it is probably always fast enough, at least with modern machines. On GTK+ it depends on the speed of the connection to the X server, which in many cases is too slow for medium-size panes.

These calls also take care to redraw the area that was drawn by previous calls to the *temp-display-callback*, so you do not to do anything about erasing the results of previous calls.

Examples

This file shows how to use **update-drawing-with-cached-display-from-points** to redraw an arrowhead shape:

```
(example-edit-file "capi/output-panes/cached-display")
```

See also

start-drawing-with-cached-display
redraw-drawing-with-cached-display
12.5 Transient display on output-pane and subclasses

update-interface-title *Generic Function*

Summary

Updates the title of an interface window.

Package

capi

Signature

update-interface-title *interface*

Arguments

interface⇓	A CAPI **interface**.

Description

The generic function **update-interface-title** updates the title of interface *interface*. This is useful when **interface-extend-title** may return a new, different, value.

You can specialize **update-interface-title** if needed.

To update all the interface titles, use **update-all-interface-titles** or **update-screen-interface-titles**.

See also

interface-extend-title
update-all-interface-titles
update-screen-interface-titles

update-internal-scroll-parameters *Function*

Summary

Updates the internal scroll parameters.

Package

capi

Signature

update-internal-scroll-parameters *pane scroll-dimension scroll-operation scroll-value*

Arguments

pane⇓	A pane that supports scrolling.
scroll-dimension⇓	**:horizontal**, **:vertical** or **:pan**.
scroll-operation⇓	**:drag**, **:move**, **:step** or **:page**.
scroll-value⇓	An integer, or a list of two integers, or a keyword, or a list of two keywords.

Description

The function **update-internal-scroll-parameters** updates the internal scroll parameters of *pane* (the ones you read by **with-geometry**, or **get-horizontal-scroll-parameters** and **get-vertical-scroll-parameters**), according to its arguments. The arguments *pane*, *scroll-dimension*, *scroll-operation* and *scroll-value* are interpreted the same way as the arguments to **scroll**. **update-internal-scroll-parameters** does not affect the display and does not perform any drawing.

Notes

update-internal-scroll-parameters is needed only when *pane* is an **output-pane** created with initargs **:coordinate-origin :fixed** or **:coordinate-origin :fixed-graphics** (see **12.4 output-pane scrolling**). It normally should not be used when **:coordinate-origin** is not supplied or **:coordinate-origin :scrolled** is supplied (the default).

The other way of setting the scroll parameters is using **set-horizontal-scroll-parameters** and

set-vertical-scroll-parameters.

update-internal-scroll-parameters is intended to be used in your *scroll-callback* (see **simple-pane** and **12.4 output-pane scrolling**). It changes the internal parameters in the same way that ordinary scrolling would change them for the same arguments, so it gives a consistent behavior with the rest of the application. You will still need to draw the appropriate things in the *display-callback*.

For example, scrolling needs to update the display based on the values of the scroll parameters before and after the scrolling happened, you can define a *scroll-callback* like this:

```
(defun my-scroll-callback (self scroll-dimension
                                scroll-operation
                                scroll-value)
  (with-geometry self
    (let ((prev-scroll-x %scroll-x%)
          (prev-scroll-y %scroll-y%))

      (update-internal-scroll-parameters
       self scroll-dimension
       scroll-operation scroll-value)

      (let ((new-scroll-x %scroll-x%)
            (new-scroll-y %scroll-y%))

        (update-display self
                        prev-scroll-x prev-scroll-y
                        new-scroll-x new-scroll-y)))))
```

See also

set-horizontal-scroll-parameters
set-vertical-scroll-parameters
simple-pane
output-pane
12.4 output-pane scrolling

update-pinboard-object *Function*

Summary

Updates the size of a **pinboard-object** to match its constraints.

Package

capi

Signature

update-pinboard-object *object* **=>** *result*

Arguments

object⇓ A **pinboard-object**.

Values

result A boolean.

Description

The function **`update-pinboard-object`** checks the constraints of *object*, and adjusts its size as necessary. It then forces the parent layout to redisplay *object* at its new size. Finally, it returns **`t`** if a resize was necessary and **`nil`** otherwise.

See also

`redraw-pinboard-object`
`pinboard-object`

update-screen-interfaces-hooks *Variable*

Summary

A list of functions that are called when a CAPI interface is created or destroyed. This variable is deprecated.

Package

`capi`

Initial Value

`nil`

Description

The variable **`*update-screen-interfaces-hooks*`** contains a list of function designators. Each function the list is called when an interface *interface* is created or destroyed.

Each function takes two arguments: the screen and *interface*.

You should not remove system functions from this variable so take care if setting its value. Only add or delete your own functions.

Notes

`*update-screen-interfaces-hooks*` is deprecated. If you use it, please contact Lisp Support.

update-screen-interface-titles *Function*

Summary

Updates interface window titles.

Package

`capi`

Signature

update-screen-interface-titles *screen*

Arguments

screen⇓ A CAPI **screen**.

Description

The function **update-screen-interface-titles** can be used to update the titles of all the interface windows on the screen *screen* when needed.

This is useful when **interface-extend-title** may return a new, different, value.

update-screen-interface-titles calls **update-interface-title** on all the relevant interfaces.

See also

interface-extend-title
update-interface-title

update-toolbar *Function*

Summary

Updates a toolbar object.

Package

capi

Signature

update-toolbar *self*

Arguments

self⇓ A **toolbar-object**.

Description

The function **update-toolbar** updates the toolbar object *self*. It computes the enabled function of *self* and the enabled functions of any toolbar components or toolbar buttons contained in it. Each toolbar object is enabled if the enabled function returns **t**, and is disabled if it returns **nil**.

See also

toolbar
toolbar-button
toolbar-component

virtual-screen-geometry *Function*

Summary

Returns, as multiple values, a screen rectangle covering the full area of all the monitors associated with a screen.

Package

capi

Signature

virtual-screen-geometry *screen* **=>** *x*, *y*, *width*, *height*

Arguments

screen⇓	A CAPI screen.

Values

x⇓	An integer.
y⇓	An integer.
width⇓	A positive integer.
height⇓	A positive integer.

Description

The function **virtual-screen-geometry** returns the "virtual" geometry of the screen *screen*, which is a screen rectangle covering the full area of all the monitors that are associated with screen.

The screen rectangle is at coordinates *x* and *y* as offsets from the top-left of the primary screen, with dimensions *width* and *height*.

See also

pane-screen-internal-geometry
screen-internal-geometries
screen-monitor-geometries
4.3 Support for multiple monitors
11.6 Querying and modifying interface geometry

with-atomic-redisplay *Macro*

Summary

Delays the updating of specified panes until all state changes have been performed.

Package

capi

Signature

with-atomic-redisplay (&rest *panes***) &body** *body* **=>** *result****

Arguments

panes⇓ Panes.

body⇓ Lisp forms.

Values

*result** Multiple values.

Description

The macro **with-atomic-redisplay** delays the updating of *panes* and their descendants until the exit from the **with-atomic-redisplay** macro.

The forms in *body* are evaluated as in implicit **progn** and the value of the last form is returned.

Most CAPI pane slot writers update the visual appearance of the pane at the point that their state changes, but it is sometimes necessary to cause all updates to the pane to be left until after they are all completed. The macro **with-atomic-redisplay** defers all visible changes to the state of each pane in *panes* until the end of the scope of the macro.

Notes

1. **with-atomic-redisplay** does not cause Graphics Ports drawing operations on *panes* to be deferred.
2. **with-atomic-redisplay** can be used recursively. The actual display happens when exiting the outermost invocation.

See also

display
simple-pane

with-busy-interface *Macro*

Summary

Displays an alternate cursor during the execution of some code, on platforms other than Cocoa.

Package

capi

Signature

with-busy-interface (*pane* **&key** *cursor delay***) &body** *body* **=>** *result****

Arguments

pane⇓ A **simple-pane**.

cursor⇓ A keyword naming a cursor or a cursor object.

delay⇓ A real number.

body⇓ Lisp forms.

Values

*result** Multiple values.

Description

The macro **with-busy-interface** switches the cursor of the interface containing *pane* to be the busy cursor, evaluates the forms in *body* as an implicit **progn**, and then restores the cursor. The value of the last form is returned. This is useful when a piece of code may take significant time to run, and visual feedback should be provided.

cursor specifies the cursor to use while *body* is running. The default value is **:busy**. For other allowed values, see **simple-pane**.

delay specifies a time in seconds before the cursor is switched, so if *body* runs in less than *delay* seconds, then the cursor is not switched at all. This is usually more useful behavior than switching the cursor immediately. The default value of *delay* is 0.5.

with-busy-interface must be called in the process of the interface containing *pane*.

with-busy-interface has no effect on Cocoa.

See also

simple-pane

with-dialog-results *Macro*

Summary

Displays a dialog and executes a body when the dialog is dismissed.

Package

capi

Signature

with-dialog-results (&rest *results***)** *dialog-form* **&body** *body* **=>** *result1***,** *result2*

Arguments

results⇓ Variables.

dialog-form⇓ A function call form.

body⇓ Forms.

Values

result1 **`:continuation`**.

result2 **`nil`**.

Description

The macro **`with-dialog-results`** is designed to evaluate *dialog-form* in a special way to allow dialogs on Cocoa to use window-modal sheets. It is not needed unless you want to make code that is portable to Cocoa. *dialog-form* should be a function call form that displays a dialog.

The overall effect is that the forms in *body* are evaluated with the variables in *results* bound to the values returned by *dialog-form* when the dialog is dismissed.

The dynamic environment in which *body* is evaluated varies between platforms:

- On Microsoft Windows, GTK+ and Motif, the **`with-dialog-results`** macro waits until the dialog has been dismissed and then evaluates *body*.
- On Cocoa, *dialog-form* creates a sheet attached to the active window and the **`with-dialog-results`** macro returns immediately. *body* is evaluated when the user dismisses the sheet.

dialog-form must be a cons with one of the following two formats:

- **`(`***function-name* **`.`** *arguments***`)`**
- **`(apply`** *function-name* **`.`** *arguments***`)`**

The *function-name* is called with all the given *arguments*, plus an additional pair of arguments, **`:continuation`** and a continuation function created from *body*. In the first format, the additional arguments are placed after all the given *arguments*. In the second format, the additional arguments are placed just before the last of the given *arguments* (i.e. before the list of remaining argument to **`apply`**).

The continuation function binds the variables in *results* to its arguments and evaluates *body*. If there are more arguments than *results* variables, the extra arguments are discarded.

This macro is designed for use with *function-name*s such as **`popup-confirmer`** or **`prompt-for-string`**, which take a **`:continuation`** keyword. You can define your own such functions provided that they call one of the CAPI functions, passing the received *continuation* argument.

Examples

On Microsoft Windows, GTK+ and Motif, this displays a dialog, calls **`record-label-in-database`** when the user clicks OK and then returns. On Cocoa, this creates a sheet and returns; **`record-label-in-database`** will be called when the user clicks OK.

```
(with-dialog-results (new-label okp)
    (prompt-for-string "Enter a label")
  (when okp  ; the user clicked in the OK button
    (record-label-in-database new-label)))
```

Here is an example with skeleton code for using **`with-dialog-results`**. Note that the dialog function (**`choose-file`** below) that is called by **`with-dialog-results`** must take a *continuation* keyword argument and pass it to a CAPI prompting function. Also note that the call to the CAPI prompting function must be the last form in the dialog function. Forms after the CAPI prompting function will be executed at an indeterminate time, and their values will not be used in the body of **`with-dialog-results`**.

```
(defun choose-file (&key continuation)
```

```
    (print 'in-choose-file)
    (capi:prompt-for-file "Choose File"
                          :pathname "~/Desktop/"
                          :continuation continuation))

  (defun open-file (rep)
    (format t "~%Opening ~a~%" rep))

  (defun my-callback ()
    (print 'doing-something-before)
    (capi:with-dialog-results (res ok-p)
        (choose-file)
      (print 'after-choose-file)
      (if ok-p
          (open-file res)
        (print 'cancelled))))

  (defun prompt-for-file-working ()
    (capi:contain
     (make-instance
      'capi:push-button
      :text "Click Here"
      :callback-type :none
      :callback 'my-callback)))

  (prompt-for-file-working)
```

See also

display-dialog
popup-confirmer
10 Dialogs: Prompting for Input

with-document-pages *Macro*

Summary

Executes a body of code repeatedly with a variable bound to the number of the page to be printed each iteration.

Package

capi

Signature

with-document-pages *page-var first-page last-page* **&body** *body*

Arguments

page-var⇓	A symbol (not evaluated).
first-page⇓	A positive integer.
last-page⇓	A positive integer.
body⇓	Lisp forms.

Description

The macro **`with-document-pages`** evaluates the forms in *body* repeatedly, with *page-var* bound to the number of the page to print on each iteration. It is used to by applications providing Page on Demand printing.

first-page and *last-page* are evaluated to yield the page numbers of the first and last pages in the document.

`with-document-pages` takes care of *first-page* and *last-page* when the user sets them in **`print-dialog`**, by evaluating *body* for the pages that are in the intersection of what user chose and the other arguments.

`with-document-pages` must be called within the dynamic context of **`with-print-job`**.

Notes

The code in *body* should do the printing by calling standard GRAPHICS-PORTS drawing functions (see **13.4 Drawing functions**), typically also using **`with-page-transform`**.

Examples

```
(example-edit-file "capi/printing/fit-to-page")
```

```
(example-edit-file "capi/printing/multi-page")
```

```
(example-edit-file "capi/printing/page-on-demand")
```

See also

`print-dialog`
`with-page`
`with-print-job`
16 Printing from the CAPI—the Hardcopy API

with-external-metafile *Macro*

Summary

Creates a metafile on disk using Graphics Ports operations.

Package

`capi`

Signature

`with-external-metafile (`*var* **`&key`** *pane bounds format pathname owner drawing-mode***`) &body`** *body* **`=> nil`**

Arguments

var⇓	A variable.
pane⇓	A graphics port, or **`nil`**.
bounds⇓	A list of four integers. Can also be **`nil`** on Microsoft Windows.

format⇓	One of the keywords **`:enhanced`**, **`:enhanced-plus`**, **`:enhanced-gdi`** and **`:windows`**.
pathname⇓	A pathname or string.
owner⇓	A graphics port, or **`nil`**.
drawing-mode⇓	One of the keywords **`:compatible`** and **`:quality`**.
body⇓	Code containing Graphic Ports operations that draw to *var*.

Description

The macro **`with-external-metafile`** creates a metafile at the location given by *pathname* containing records corresponding to the Graphics Ports operations in *body* that draw to *var*.

On Microsoft Windows the metafile is a device-independent format for storing pictures. For more information about metafiles, see the Microsoft documentation.

On Cocoa and GTK+ the metafile format is PDF.

If *pane* is **`nil`**, the macro binds *var* to an object of type **`metafile-port`**. If *pane* is non-nil then it must be an instance of **`output-pane`** or a subclass. In this case *var* is bound to *pane*, and *pane* is modified within the dynamic extent of **`with-external-metafile`** so all drawing operations draw to the metafile instead of *pane*. This can be useful when reusing existing redisplay code that is written expecting an **`output-pane`**. The default value of pane is **`nil`**.

If *bounds* is **`nil`** the metafile size will be computed from the drawing done within the body. This value is not allowed on Cocoa.

If *bounds* is non-nil (required on Cocoa), it should be a list of integers specifying the coordinate rectangle (*x y width height*) that the metafile contains.

pathname specifies the filename of the metafile. If its **`pathname-type`** is **`nil`**, then the file extension **`"EMF"`** is used for an Enhanced-metafile, or **`"WMF"`** for a Windows-metafile.

owner specifies the owner of the metafile, which calls to **`port-owner`** will return. This has an effect only when *pane* is **`nil`**.

drawing-mode should be either **`:compatible`** which causes drawing to be the same as in LispWorks 6.0, or **`:quality`** which causes all the drawing to be transformed properly, and allows control over anti-aliasing on Microsoft Windows and GTK+. The default value of *drawing-mode* is **`:quality`**.

For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

On Cocoa and GTK+ the metafile format is always PDF as a single page, and *format* is ignored. *format* is used only on Microsoft Windows and it can be one of:

`:enhanced`	Generate an Enhanced-metafile file containing "dual drawing" both in GDI+ and GDI.
`:enhanced-plus`	Generate an Enhanced-metafile file containing drawing only in GDI+.
`:enhanced-gdi`	Generate an Enhanced-metafile file containing drawing only in GDI.
`:windows`	Generate a Windows-metafile.

The default value of *format* is **`:enhanced`**.

When *drawing-mode* is **`:compatible`** (rather than the default value **`:quality`**) **`:enhanced`** and **`:enhanced-plus`** behave like **`:enhanced-gdi`**.

Notes

1. GDI+ gives the best quality, so normally that is what you would want. However some programs may be able to display only GDI (and not GDI+), which is why the default is dual drawing. This however generates a larger file and is presumably slightly slower, so if you are sure that the file will be used only by programs that can draw GDI+ emf files (sometimes called EMF+), you can use *format* **`:enhanced-plus`**.
2. **`with-external-metafile`** is not implemented on X11/Motif.

See also

`draw-metafile`
`metafile-port`
`port-owner`
`with-internal-metafile`
13 Drawing - Graphics Ports

with-geometry *Macro*

Summary

Helps you to define layouts and create new **`pinboard-object`** subclasses.

Package

`capi`

Signature

`with-geometry` *pane* **`&body`** *body* **`=>`** *result**

Arguments

pane⇓ A **`simple-pane`** or a **`pinboard-object`**.

body⇓ Lisp forms.

Values

*result** Multiple values.

Description

The macro **`with-geometry`** is used for defining layouts and for creating new **`pinboard-object`** subclasses, by providing access to the geometry of a pane.

`with-geometry` evaluates the forms in *body* as an implicit **`progn`** while binding the following variables to slots in the geometry of *pane* in much the same way as the Common Lisp macro **`with-slots`**. Except the special cases which are mentioned below, these variables are read-only and should not be set.

Four variables define the geometry of the pane. If you define define your own **`calculate-layout`** method, it can set these variables:

`%x%` An integer specifying the x position of the pane in pixels relative to its parent.

%y% An integer specifying the y position of the pane in pixels relative to its parent.

%width% An integer specifying the width in pixels of the pane.

%height% An integer specifying the height in pixels of the pane.

Four variables specify constraints on the pane. If you define your own **calculate-constraints** method, it can set these variables:

%min-width% A real number specifying the minimum width of the pane.

%min-height% A real number specifying the minimum height of the pane.

%max-width% A real number specifying the maximum width of the pane.

%max-height% A real number specifying the maximum height of the pane.

The following variables are also bound but apply only to instances of **output-pane** or **layout** which have at least one scroll bar. They can be retrieved by **get-horizontal-scroll-parameters** and **get-vertical-scroll-parameters**. They can be set by **set-horizontal-scroll-parameters** and **set-vertical-scroll-parameters**. These variables should be regarded as read-only inside **with-geometry** (they are writable for backwards compatibility only).

%scroll-width% The extent of the horizontal scroll range.

%scroll-height% The extent of the vertical scroll range.

%scroll-horizontal-page-size%

The horizontal scroll page size.

%scroll-horizontal-slug-size%

The width of the scroll bar slug.

%scroll-horizontal-step-size%

The horizontal scroll step size.

%scroll-start-x% The start of the horizontal scroll range.

%scroll-start-y% The start of the vertical scroll range.

%scroll-vertical-page-size%

The vertical scroll page size.

%scroll-vertical-slug-size%

The height of the scroll bar slug.

%scroll-vertical-step-size%

The vertical scroll step size.

%scroll-x% x coordinate of the current scroll position.

%scroll-y% y coordinate of the current scroll position.

The following two variables access the object for which the representation is:

%object% The object whose geometry this is.

%child% The same as **%object%** (kept for compatibility with LispWorks 3.1).

See also

calculate-constraints
calculate-layout
convert-relative-position
element
get-horizontal-scroll-parameters
get-vertical-scroll-parameters
scroll
set-horizontal-scroll-parameters
set-vertical-scroll-parameters
3.8 Accessing pane geometry
6 Laying Out CAPI Panes
12 Creating Panes with Your Own Drawing and Input

with-internal-metafile *Macro*

Summary

Creates a metafile in memory using Graphics Ports operations.

Package

capi

Signature

with-internal-metafile (*var* **&key** *pane bounds format owner drawing-mode***) &body** *body* **=>** *metafile*

Arguments

var⇓ A variable.

pane⇓ A graphics port, or **nil**.

bounds⇓ A list of four integers. Can also be **nil** on Microsoft Windows.

format⇓ One of the keywords **:enhanced**, **:enhanced-plus** and **:enhanced-gdi**.

owner⇓ A graphics port, or **nil**.

drawing-mode⇓ One of the keywords **:compatible** and **:quality**.

body⇓ Lisp code.

Values

metafile⇓ A metafile.

Description

The macro **with-internal-metafile** creates a metafile containing records corresponding to the Graphics Ports operations in *body* that draw to *var*.

`with-internal-metafile` behaves like **`with-external-metafile`** except that an object representing the metafile is returned, and no file is created on disk.

var, *pane*, *bounds*, *format*, *owner*, *drawing-mode* and *body* are interpreted as for **`with-external-metafile`** except that *format* cannot have the value **`:windows`**.

Note: GDI+ gives the best quality, so normally that what you want. But you cannot put a GDI+ only metafile on the clipboard, which is why the default is to make a "dual" metafile containing both GDI and GDI+ drawing. If are not going to put the metafile on the clipboard (by calling **`set-clipboard`** with *format* **`:metafile`**) you can use *format* **`:enhanced-plus`** which is slightly faster and uses less memory.

metafile must be freed after use, by calling **`free-metafile`**.

Notes

1. **`with-internal-metafile`** is supported on GTK+ only where Cairo is supported (GTK+ version 2.8 and later).
2. On GTK+, the internal metafile is slow to resize, so it is probably not useful when it is frequently resized (that is, drawn with different width or height).
3. **`with-internal-metafile`** is not implemented on X11/Motif.

Examples

```
(example-edit-file "capi/graphics/metafile")
```

```
(example-edit-file "capi/graphics/metafile-rotation")
```

See also

`draw-metafile`
`free-metafile`
`port-owner`
`with-external-metafile`
13 Drawing - Graphics Ports

with-output-to-printer *Macro*

Summary

Binds a stream variable and prints its output.

Package

`capi`

Signature

`with-output-to-printer (`*stream* **`&key`** *printer tab-spacing interactive jobname***`) &body`** *body* **`=>`** *result****

Arguments

stream⇓ A variable.

printer⇓ A printer or **nil**.

tab-spacing⇓ An integer.

interactive⇓ A boolean.

jobname⇓ A string.

body⇓ Lisp forms.

Values

*result** The values returned by evaluating *body*.

Description

The macro **with-output-to-printer** binds the variable *stream* to a stream object, and prints everything is that is written to it in the code of *body*.

If *interactive* is **t** then **print-dialog** is called to select the printer to use. If *interactive* is **nil** then *printer* is used unless it is **nil** in which case the **current-printer** is used. The default value of *interactive* is **t** and the default value of *printer* is **nil**.

The values of *jobname* and *tab-spacing* are passed to **print-text**, which is used to actually do the printing. The default value of *tab-spacing* is 8 and the default value of *jobname* is "Text".

See also

current-printer
print-dialog
print-text
16 Printing from the CAPI—the Hardcopy API

with-page *Macro*

Summary

Binds a variable to either **t** or **nil**, and executes a body of code to print a page only if the variable is **t**.

Package

capi

Signature

with-page (*printp***) &body** *body*

Arguments

printp⇓ A symbol (not evaluated).

body⇓ Lisp forms.

Description

The macro **with-page** binds *printp* to **t** if a page is to be printed, or **nil** if it is to be skipped. The forms in *body* are evaluated once as in implicit **progn**, and are expected to draw the document only if *printp* is **t**.

Each call to **with-page** contributes a new page to the document.

with-page must be called within the dynamic context of **with-print-job**.

Notes

1. **with-page** does not work on Cocoa.
2. The code in *body* should do the printing by calling standard GRAPHICS-PORTS drawing functions (see **13.4 Drawing functions**), typically also using **with-page-transform**.
3. *printp* can be **nil** when only part of the document is printed, for example when the user specifies that she wants only odd pages. When *printp* is **nil**, the code in *body* needs to ensure that the next call to **with-page** prints the right page.
4. Normally **with-document-pages** is the preferred method of printing.

See also

with-document-pages
with-page-transform
with-print-job
16 Printing from the CAPI—the Hardcopy API

with-page-transform *Macro*

Summary

Defines a rectangular region within the coordinate space of an output pane or printer port.

Package

capi

Signature

with-page-transform (*x y width height***) &body** *body*

Arguments

x⇓, *y*⇓	Real numbers.
width⇓, *height*⇓	Positive real numbers.
body⇓	Lisp forms.

Description

The macro **with-page-transform** evaluates *x*, *y*, *width* and *height* to define a rectangular region within the coordinate space of an output pane or printer port. The forms of *body* are evaluated as an implicit **progn** with that region mapped onto the printable area of the page. If the specified rectangle does not have the same aspect ratio as the printable area of the page, then non-isotropic scaling will occur.

Any number of calls to **with-page-transform** can occur during the printing of a page; for example, it is sometimes convenient to use a different page transform from that used to print the main body of the page when printing headers and footers.

Examples

```
(example-edit-file "capi/graphics/metafile")
```

```
(example-edit-file "capi/printing/fit-to-page")
```

```
(example-edit-file "capi/printing/multi-page")
```

```
(example-edit-file "capi/printing/page-on-demand")
```

See also

get-printer-metrics
with-document-pages
with-page
16 Printing from the CAPI—the Hardcopy API

with-print-job *Macro*

Summary

Creates a print job that prints to the specified printer.

Package

capi

Signature

with-print-job (*var* **&key** *pane jobname printer owner drawing-mode***) &body** *body*

Arguments

var⇓	A symbol.
pane⇓	A **output-pane** or **nil**.
jobname⇓	A string or **nil**.
printer⇓	A printer or **nil**.
owner⇓	An owner window, or **nil**.
drawing-mode⇓	One of **:compatible**, **:quality** or **nil**.
body⇓	Lisp forms.

Description

The macro **`with-print-job`** creates a print job that prints to *printer*. If *printer* is not specified, the default printer is used. The macro binds *var* to a graphics port object and evaluates the forms in *body* as an implicit **`progn`**. Printing is performed by these forms using Graphics Ports operations to draw to *var*.

If *pane* is non-nil it must be an instance of **`output-pane`** or a subclass. In this case *var* is bound to *pane*, and *pane* is modified within the dynamic extent of the **`with-print-job`** so all drawing operations draw to the printer instead of *pane*. This can be useful when implementing printing by modifying existing redisplay code that is written expecting an **`output-pane`**. If pane is **`nil`**, *var* is bound to a graphics port of type **`printer-port`**, which is alive only inside the body of **`with-print-job`**, and sends any drawing into it to the printer.

jobname is the name of the print job. The default value is **`nil`**, meaning that the name "Document" is used.

The actual printing is done by using one of the macros **`with-document-pages`** or **`with-page`**, within the scope of **`with-print-job`**.

owner specifies the owner of the printer port object, which calls to **`port-owner`** will return. This has an effect only when *pane* is **`nil`**.

drawing-mode should be either **`:compatible`** which causes drawing to be the same as in LispWorks 6.0, or **`:quality`** which causes all the drawing to be transformed properly, and allows control over anti-aliasing on Microsoft Windows and GTK+. If *pane* is supplied, then *pane* determines the print job's *drawing-mode*, otherwise the default value of *drawing-mode* is **`:quality`**.

For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

Examples

```
(example-edit-file "capi/graphics/metafile")
```

```
(example-edit-file "capi/printing/fit-to-page")
```

```
(example-edit-file "capi/printing/multi-page")
```

```
(example-edit-file "capi/printing/page-on-demand")
```

See also

`port-owner`
`printer-port-handle`
`printer-port-supports-p`
`set-printer-options`
`with-document-pages`
`with-page`
`with-page-transform`
16 Printing from the CAPI—the Hardcopy API
13 Drawing - Graphics Ports

with-random-typeout *Macro*

Summary

Binds a stream variable to a collector pane.

Package

capi

Signature

with-random-typeout (*stream-variable pane***) &body** *body*

Arguments

stream-variable⇓	A symbol (not evaluated).
pane⇓	A pane.
body⇓	Lisp forms.

Description

The macro **with-random-typeout** binds the variable *stream-variable* to a collector pane stream associated with *pane* for the scope of the macro and evaluates the forms in *body* as an implicit **progn**. The collector pane is automatically mapped and unmapped around the body. If *body* exits normally, the typeout is not unmapped until the space bar is pressed or the mouse is clicked.

See also

map-typeout
unmap-typeout
collector-pane

wrap-text *Function*

Summary

Wraps text for a given character width.

Package

capi

Signature

wrap-text *text width* **&key** *start end* **=>** *strings*

Arguments

text⇓ A string.

width⇓ A positive integer.

start⇓, *end*⇓ Bounding index designators of *text*.

Values

strings⇓ A list of strings.

Description

The function **`wrap-text`** takes a string *text* and returns a list of strings, each of which is no longer than *width*. Together the strings in *strings* contain all the non-whitespace characters of *text* between *start* and *end* and are suitable for displaying this text on multiple lines of length *width*.

See also

`wrap-text-for-pane`

wrap-text-for-pane *Function*

Summary

Wraps text for a given pane.

Package

`capi`

Signature

`wrap-text-for-pane` *pane text* **`&key`** *external-width visible-width font start end* **`=>`** *strings*

Arguments

pane⇓ A displayed CAPI pane.

text⇓ A string.

external-width⇓ An integer or **`nil`**.

visible-width⇓ An integer or **`nil`**.

font⇓ A font object.

start⇓ An integer.

end⇓ An integer or **`nil`**.

Values

strings⇓ A list of strings.

Description

The function **`wrap-text-for-pane`** takes a string *text* and returns a list of strings. Together the strings in *strings* contain all the non-whitespace characters of *text* and are suitable for displaying this text on *pane*. That is, each string has a display width no greater than the width of *pane* when drawn using the font of *pane*. The arguments *start* and *end* are used as bounding index designators for *text* and characters outside these bounds are ignored.

If *visible-width* is non-nil then text is wrapped to that width. Otherwise, if *external-width* is non-nil then text is wrapped as if the pane had that external width (that is, taking account of any borders in the pane). If both *visible-width* and *external-width* are **`nil`**, then the text is wrapped to the current visible width of the pane. The default value of both *visible-width* and *external-width* is **`nil`**.

font is used to perform the wrapping calculations. If it is **`nil`** (the default), then the **`graphics-state-font`** is used for panes such as **`output-pane`** that have a **`graphics-state`** and the **`simple-pane-font`** is used for other panes.

See also

`wrap-text`

x-y-adjustable-layout *Class*

Summary

The class **`x-y-adjustable-layout`** provides functionality for positioning panes in a space larger than themselves (for example, it is used to choose whether to center them, or left justify them).

Package

`capi`

Superclasses

`layout`

Subclasses

`simple-layout`
`grid-layout`

Initargs

`:x-adjust`	The adjust value for the *x* direction.
`:y-adjust`	The adjust value for the *y* direction.

Accessors

`layout-x-adjust`
`layout-y-adjust`

Description

The values *x-adjust* and *y-adjust* of the slots are used by layouts to decide what to do when a pane is smaller than the space in which it is being laid out. Typically the values will be a keyword or a list of the form **`(`***keyword n***`)`** where *n* is an integer. These values of *adjust* are interpreted as by **`pane-adjusted-position`**.

:top is the default for *y-adjust* and **:left** is the default for *x-adjust*.

Examples

Note: **column-layout** is a subclass of **x-y-adjustable-layout**.

```
(setq column (capi:contain
               (make-instance
                'capi:column-layout
                :description (list
                              (make-instance
                               'capi:push-button
                               :text "Ok")
                              (make-instance
                               'capi:list-panel
                               :items '(1 2 3 4 5)
                              )))))

(capi:apply-in-pane-process
 column #'(setf capi:layout-x-adjust) :right column)

(capi:apply-in-pane-process
 column #'(setf capi:layout-x-adjust) :center column)
```

See also

pane-adjusted-position

22 GRAPHICS-PORTS Reference Entries

The following chapter provides reference entries for the symbols exported from the **graphics-ports** package. You can use these to draw graphics in CAPI output panes, which are a kind of graphics port. See **13 Drawing - Graphics Ports** for more information on graphics ports and their associated types.

2pi *Constant*

Summary

(* 2 pi) as a **double-float**.

Package

graphics-ports

Description

The constant **2pi** is the result of **(* 2 cl:pi)**. It is a **cl:double-float**.

See also

fpi
pi-by-2

analyze-external-image *Function*

Summary

Gets the properties of DIB data in an external image.

Package

graphics-ports

Signature

analyze-external-image *external-image* **=>** *width*, *height*, *color-table*, *number*

Arguments

external-image⇓ An **external-image**.

Values

width An integer.

height	An integer.
color-table	A color table.
number	An integer.

Description

The function **`analyze-external-image`** returns the width, height, color-table, and number of important colors for the external image *external-image*.

The image data in *external-image* must be in Device Independent Bitmap (DIB) format.

apply-rotation *Function*

Summary

Modifies a **`transform`** such that a rotation of a given number of radians is performed on any points multiplied by the transform.

Package

`graphics-ports`

Signature

`apply-rotation` *transform theta* **`=>`** *transform*

Arguments

transform⇓	A **`transform`**.
theta⇓	A real number.

Values

transform	A **`transform`**.

Description

The function **`apply-rotation`** modifies *transform* such that a rotation of *theta* radians is performed on any points multiplied by the transform. Any operations already contained in the transform occur before the new rotation.

The rotation is around the point (0,0).

If *theta* is positive, then the rotation is clockwise.

`apply-rotation` returns the transform.

Notes

See **`graphics-state`** for details of how a **`transform`** is used.

Examples

```
(example-edit-file "capi/graphics/metafile-rotation")
```

See also

apply-rotation-around-point
apply-scale
apply-translation
graphics-state
transform

apply-rotation-around-point *Function*

Summary

Modifies a **transform** such that a specified rotation around a specified point is performed on any points multiplied by the transform.

Package

graphics-ports

Signature

apply-rotation-around-point *transform theta x y* **=>** *transform*

Arguments

transform⇓	A **transform**.
theta⇓	A real number.
x⇓	A real number.
y⇓	A real number.

Values

transform	A **transform**.

Description

The function **apply-rotation-around-point** modifies *transform* such that a clockwise rotation of *theta* radians around the point (*x*,*y*) is performed on any points multiplied by the transform. Any operations already contained in the transform occur before the new rotation.

apply-rotation-around-point returns the transform.

Notes

See **graphics-state** for details of how a **transform** is used.

Examples

```
(example-edit-file "capi/graphics/rotation-around-point")
```

There are further examples in **20 Self-contained examples**.

See also

apply-rotation
graphics-state
transform

apply-scale *Function*

Summary

Modifies a **transform** such that a scaling occurs on any points multiplied by the transform.

Package

graphics-ports

Signature

apply-scale *transform sx sy* **=>** *transform*

Arguments

transform⇓	A **transform**.
sx⇓	A real number.
sy⇓	A real number.

Values

transform	A **transform**.

Description

The function **apply-scale** modifies *transform* such that a scaling of *sx* in *x* and *sy* in *y* is performed on any points multiplied by the transform. Any operations already contained in the transform occur before the new scaling.

apply-scale returns the transform.

Notes

See **graphics-state** for details of how a **transform** is used.

Examples

```
(example-edit-file "capi/graphics/metafile-rotation")
```

See also

apply-rotation
apply-rotation-around-point
apply-translation
graphics-state
transform

apply-translation *Function*

Summary

Modifies a transform such that a translation is performed on any points multiplied by the transform.

Package

graphics-ports

Signature

apply-translation *transform dx dy* **=>** *transform*

Arguments

transform⇓	A **transform**.
dx⇓	A real number.
dy⇓	A real number.

Values

transform	A **transform**.

Description

The function **apply-translation** modifies *transform* such that a translation of (*dx dy*) is performed on any points multiplied by the transform. Any operations already contained in the transform occur before the new translation.

apply-translation returns the transform.

Notes

See **graphics-state** for details of how a **transform** is used.

Examples

```
(example-edit-file "capi/graphics/metafile-rotation")
```

See also

apply-rotation
apply-rotation-around-point
apply-scale

graphics-state
transform

augment-font-description *Function*

Summary

Returns a font description combining the attributes of a given font description with a set of font attributes.

Package

graphics-ports

Signature

augment-font-description *fdesc* **&rest** *font-attributes* **=>** *return*

Arguments

fdesc⇓	A font description.
font-attributes⇓	Font attributes.

Values

return⇓	A font description.

Description

The function **augment-font-description** returns a font description that contains all the attributes of *fdesc* combined with *font-attributes*. The attribute **:stock** is handled specially: it is omitted from *return*, unless it is the only attribute specified.

If an attribute appears in both *fdesc* and *font-attributes*, the value in *font-attributes* is used. The contents of *fdesc* are not modified.

See also

make-font-description
13 Drawing - Graphics Ports

clear-external-image-conversions *Function*

Summary

Clears external image conversions for a port.

Package

graphics-ports

Signature

`clear-external-image-conversions` *external-image-or-null gp-or-null* **`&key`** *free-image all errorp*

Arguments

external-image-or-null⇓
An external image or **`nil`**.

gp-or-null⇓ A graphics port or **`nil`**.

free-image⇓ A boolean.

all⇓ A boolean.

errorp⇓ A boolean.

Description

The function **`clear-external-image-conversions`** clears the external image conversions for a port.

If *external-image-or-null* is **`nil`**, then conversions for all images are cleared. Otherwise, only conversions for *external-image-or-null* are cleared.

If *gp-or-null* is **`nil`** all conversions are cleared using the image-color-users. If *all* is non-nil all conversions for all ports are cleared using *gp-or-null*. Conversions are also freed if *free-image* is non-nil. By default, *free-image* is **`t`**, *all* is **`(null`** *gp-or-null***`)`**, and *errorp* is **`t`**.

See also

13 Drawing - Graphics Ports

clear-graphics-port *Function*

Summary

Draws a filled rectangle covering the entire port in the port's background color.

Package

`graphics-ports`

Signature

`clear-graphics-port` *port*

Arguments

port⇓ A graphics port.

Description

The function **`clear-graphics-port`** draws a filled rectangle in *port* covering the entire port in the port's *background*. All other graphics state parameters are ignored.

clear-graphics-port-state *Function*

Summary

Sets the graphics state of a port back to its default values.

Package

`graphics-ports`

Signature

`clear-graphics-port-state` *port*

Arguments

port⇓	A graphics port.

Description

The function **`clear-graphics-port-state`** sets the graphics state of *port* back to its default values, which are the ones it possessed immediately after creation.

See also

`graphics-state`

clear-rectangle *Function*

Summary

Draws a rectangle in the port's background color. This function is deprecated.

Package

`graphics-ports`

Signature

`clear-rectangle` *port x y width height*

Arguments

port⇓	A graphics port.
x⇓	A real number.
y⇓	A real number.
width⇓	A real number.
height⇓	A real number.

Description

The function **clear-rectangle** (deprecated) draws the rectangle specified by *x*, *y*, *width*, and *height* in *port* using the port's background color. All other **graphics-state** parameters are ignored.

clear-rectangle is deprecated because it ignores the graphics state args, which means it does not work properly with other drawing functions. In particular, it does not work properly in the *display-callback* of **output-pane**.

Use instead:

```
(draw-rectangle pane x y width height
                :filled t
                :foreground color
                :compositing-mode :copy
                :shape-mode :plain)
```

compositing-mode is needed only when the color has alpha.

foreground is needed only if it is different from the foreground in the graphics state.

Note that **draw-rectangle** does take into account the transformation in the **graphics-state**.

See also

draw-rectangle
13 Drawing - Graphics Ports

compress-external-image *Function*

Summary

Compresses DIB data in an external image.

Package

graphics-ports

Signature

compress-external-image *external-image* **=>** *result*

Arguments

external-image⇓ An **external-image**.

Values

result The difference in bytes between size of the original image and the size of the compressed version.

Description

The function **compress-external-image** converts the data of *external-image* into compressed DIB format.

The image data in *external-image* must be in Device Independent Bitmap (DIB) format.

compute-char-extents *Function*

Summary

Returns the *x* coordinates of the end of each of the characters in a string if the string was printed to a graphics port.

Package

`graphics-ports`

Signature

`compute-char-extents` *port string* **`&optional`** *font* **`=>`** *extents*

Arguments

port⇓	A CAPI pane.
string⇓	A string.
font⇓	A font.

Values

extents	An array of integers.

Description

The function **`compute-char-extents`** returns the extents of the characters in *string* in the font associated with *port*, or of *font* if given. The extents are an array, one element per character, which gives the ending *x* coordinate of that character if the string was drawn to *port*.

Note: To compute the extents of the entire string for a given port or font, use **`port-string-width`** or **`get-string-extent`**.

See also

`get-string-extent`
`port-string-width`

convert-external-image *Function*

Summary

Returns an image derived from an external image format.

Package

`graphics-ports`

Signature

`convert-external-image` *gp external-image* **`&key`** *cache force-new* **`=>`** *image*

Arguments

gp⇓	A CAPI pane.
external-image⇓	An **`external-image`**.
cache⇓	A boolean.
force-new⇓	A boolean.

Values

image	An **`image`**.

Description

The function **`convert-external-image`** returns an **`image`** derived from *external-image* . The image is ready for drawing to *gp*.

If *cache* is non-nil image conversions are cached in *external-image*. The default value of *cache* is **`nil`**.

If *force-new* is non-nil a new image is always created, and put in the cache. The default value of *force-new* is **`nil`**.

See also

13 Drawing - Graphics Ports

convert-to-font-description *Function*

Summary

Converts a font-spec to a font description.

Package

`graphics-ports`

Signature

`convert-to-font-description` *port font-spec* **`=>`** *fdesc*

Arguments

port⇓	A graphics port.
font-spec⇓	A font description object, font or symbol.

Values

fdesc⇓	A font-description.

Description

The function **convert-to-font-description** converts *font-spec* to a font description object *fdesc* for the graphics port *port*. If *font-spec* is a font, then its description is returned. If *font-spec* is a font description object, then it is returned. If *font-spec* is a symbol naming a font alias, then **convert-to-font-description** converts this alias to a font and returns its font description. Other platform-specific values of *font-spec* are also accepted.

See also

font-description
make-font-description

copy-area *Function*

Summary

Copies a rectangular area from one port to another.

Package

graphics-ports

Signature

copy-area *to-port from-port to-x to-y width height from-x from-y* **&rest** *args*

Arguments

to-port⇓	A graphics port.
from-port⇓	A graphics port.
to-x⇓	A real number.
to-y⇓	A real number.
width⇓	A real number.
height⇓	A real number.
from-x⇓	A real number.
from-y⇓	A real number.
args⇓	**graphics-state** parameters passed as keyword arguments.

Description

The function **copy-area** copies a rectangular area from *from-port* to *to-port*, taking account of transformations.

In *drawing-mode* **:compatible** (old drawing mode), **copy-area** is exactly the same as **copy-pixels**.

In *drawing-mode* **:quality** (the default), **copy-area** copies a rectangular area from one port to another. The *transform*, *mask*, *mask-transform*, *compositing-mode* and *shape-mode* from *to-port*'s **graphics-state** are all used, unless overridden in *args*. *to-port* and *from-port* need not have the same depth and can be the same object. The corners of the copied rectangle are (*from-x from-y*), (*from-x*+*width from-y*), (*from-x*+*width from-y*+*height*) and (*from-x from-y*+*height*), which are interpreted as pixel positions in the window coordinates of *from-port*, that is, they are not transformed by *from-port*'s transform. The top left of the rectangle is copied to (*to-x to-y*) in *to-port*'s coordinates.

Notes

The main difference between **copy-area** and **copy-pixels** in *drawing-mode* **:quality** is when copying from a displayed window.

copy-area always copies using the correct transformation of the target, but that it means that it may copy from an obscured part of the window and hence copy the wrong thing. **copy-pixels** generates an exposure event on the target port instead of copying obscured areas, but to do that it has to ignore the transformation.

Examples

```
(example-edit-file "capi/graphics/compositing-mode")
```

See also

copy-pixels
graphics-state
13 Drawing - Graphics Ports

copy-external-image *Function*

Summary

Returns a copy of an external image.

Package

graphics-ports

Signature

copy-external-image *external-image* **&key** *new-color-table* **=>** *new-external-image*

Arguments

external-image⇓	An external image.
new-color-table⇓	A color table.

Values

new-external-image	An external image.

Description

The function **copy-external-image** returns a copy of *external-image*, optionally supplying a *new-color-table*. An error is signalled if this is a different size from the existing color-table.

copy-pixels *Function*

Summary

Copies a rectangular area from one port to another.

Package

graphics-ports

Signature

copy-pixels *to-port from-port to-x to-y width height from-x from-y* **&rest** *args*

Arguments

to-port⇓	A graphics port.
from-port⇓	A graphics port.
to-x⇓	A real number.
to-y⇓	A real number.
width⇓	A real number.
height⇓	A real number.
from-x⇓	A real number.
from-y⇓	A real number.
args⇓	**graphics-state** parameters passed as keyword arguments.

Description

The function **copy-pixels** copies a rectangular area from *from-port* to *to-port*. The *transform*, *mask*, *mask-transform*, *compositing-mode* and *shape-mode* from *to-port*'s **graphics-state** are all used, unless overridden in *args*. *to-port* and *from-port* need not have the same depth and can be the same object.

The corners of the copied rectangle are (*from-x from-y*), (*from-x*+*width from-y*), (*from-x*+*width from-y*+*height*) and (*from-x from-y*+*height*), which are interpreted as pixel positions in the window coordinates of *from-port*, that is, they are not transformed by *from-port*'s transform. The top left of the rectangle is copied to (*to-x to-y*) in *to-port*'s coordinates.

When *to-port*'s *drawing-mode* is **:quality** the target is generally fully transformed, except that when it copies from a visible window it may generate expose events when copying from an obscured part, and in *drawing-mode* **:quality** it ignores the transformation in this case.

If *to-port*'s *drawing-mode* is **:compatible** then the image is not scaled or rotated. For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

Notes

copy-pixels can be used to draw to an **output-pane** inside the *display-callback* of that pane, but it cannot be used to copy from the **output-pane** inside its *display-callback* (the result of such an operation is not defined).

See also

copy-area
output-pane
13 Drawing - Graphics Ports

copy-transform *Function*

Summary

Returns a copy of a transform.

Package

graphics-ports

Signature

copy-transform *transform* **=>** *result*

Arguments

transform⇓ A **transform**.

Values

result A **transform**.

Description

The function **copy-transform** returns a copy of *transform*.

Notes

See **graphics-state** for details of how a **transform** is used.

See also

graphics-state
transform

create-pixmap-port *Function*

Summary

Creates a pixmap port and its window system representation.

Package

graphics-ports

Signature

create-pixmap-port *port width height* **&key** *background foreground collect relative clear drawing-mode* **=>** *pixmap-port*

Arguments

port⇓ A graphics port for a window.

width⇓ An integer.

height⇓ An integer.

background⇓ A color specification, or **nil**.

foreground⇓ A color specification, or **nil**.

collect⇓ A boolean.

relative⇓ A boolean.

clear⇓ A list or **t**.

drawing-mode⇓ One of the keywords **:compatible** and **:quality**.

Values

pixmap-port⇓ A pixmap graphics port.

Description

The function **create-pixmap-port** creates a pixmap port *pixmap-port* and its window system representation. *port* specifies the color-user, used for color conversions, and its representation may also be used by the library to match the pixmap port properties. *pixmap-port* will have dimensions *width*, *height* and will use the specified *drawing-mode*.

background and *foreground* are used to initialize the **graphics-state-background** and **graphics-state-foreground** of *pixmap-port*. If *background* or *foreground* are **nil** then the corresponding color from *port* is used.

If *clear* is **t**, then *pixmap-port* is cleared to its background color, otherwise the initial colors will be non-deterministic. If *clear* is a list of the form (*x y width height*), only that part of *pixmap-port* is cleared initially. The default value is **nil**.

If *relative* is non-nil, then *pixmap-port* collects pixel coordinates corresponding to the left, top, right, and bottom extremes of the drawing operations taking place within the body forms, and if these extend beyond the edges of *pixmap-port* (into negative coordinates for example) the entire drawing is offset by an amount which ensures it remains within the port. It is as if the port moves its relative origin in order to accommodate the drawing. If the drawing size is greater than the screen size, then some of it is lost. The default value is **nil**.

If *collect* is non-nil, this causes the drawing extremes to be collected but without having the pixmap shift to accommodate the drawing, as *relative* does. The extreme values can be read using the **get-bounds** function. The default value of *collect* is *relative*.

When *pixmap-port* is no longer needed, it should be destroyed by calling **destroy-pixmap-port**. Alternatively, use **with-pixmap-graphics-port** to create and destroy the port within a dynamic extent.

See also

get-bounds
destroy-pixmap-port
with-pixmap-graphics-port
13 Drawing - Graphics Ports

default-image-translation-table *Variable*

Summary

The default image translation table.

Package

`graphics-ports`

Initial Value

The global image translation table.

Description

The variable **`*default-image-translation-table*`** contains the default image translation table. It is used if no image translation table is specified in calls to image translation table functions.

See also

`load-image`

define-font-alias *Function*

Summary

Defines an alias for a font.

Package

`graphics-ports`

Signature

`define-font-alias` *keyword* *font*

Arguments

keyword⇓	A keyword.
font⇓	A font or a **font-description** object.

Description

The function **`define-font-alias`** defines *keyword* as an alias for *font*.

Notes

Once a font alias is defined, it can be used to specify the *font* for a CAPI pane (see **`simple-pane`**).

See also

13.9 Portable font descriptions

destroy-pixmap-port *Function*

Summary

Destroys a pixmap port, thereby freeing any window system resources it used.

Package

graphics-ports

Signature

destroy-pixmap-port *pixmap-port*

Arguments

pixmap-port⇓	A pixmap port.

Description

The function **destroy-pixmap-port** destroys *pixmap-port*, freeing any window system resources.

dither-color-spec *Function*

Summary

Returns **t** if the color specification for a given pixel should result in a pixel that is on in a 1 bit dithered bitmap.

Package

graphics-ports

Signature

dither-color-spec *rgb-color-spec y x* **=>** *result*

Arguments

rgb-color-spec⇓	An RGB specification.
y⇓	An integer.
x⇓	An integer.

Values

result	A boolean.

Description

The function **`dither-color-spec`** returns **`t`** if *rgb-color-spec* should result in a pixel that is on at the point (*x y*) in a 1-bit dithered bitmap. The current set of dithers is used in the decision.

Notes

`dither-color-spec` is deprecated. Dithers do not affect drawing or the anti-aliasing that occurs when drawing in Cocoa.

See also

`initialize-dithers`
`make-dither`
`with-dither`

draw-arc *Function*

Summary

Draws an arc.

Package

`graphics-ports`

Signature

`draw-arc` *port x y width height start-angle sweep-angle* **`&rest`** *args* **`&key`** *filled*

Arguments

port⇓	A graphics port.
x⇓	A real number.
y⇓	A real number.
width⇓	A real number.
height⇓	A real number.
start-angle⇓	A real number.
sweep-angle⇓	A real number.
args⇓	**`graphics-state`** parameters passed as keyword arguments.
filled⇓	A boolean.

Description

The function **`draw-arc`** draws an arc contained in the rectangle from (*x y*) to (*x*+*width y*+*height*) from *start-angle* to *start-angle*+*sweep-angle*. Both angles are specified in radians. Currently, arcs are parts of ellipses whose major and minor axes are parallel to the screen axes. When *port*'s *drawing-mode* is **`:quality`** the arc is transformed properly, but if *drawing-mode* is **`:compatible`** and *port* has rotation in its transform, the enclosing rectangle is modified to be the external enclosing orthogonal rectangle of the rotated rectangle. The start angle is rotated. The *transform*, *foreground*, *background*, *operation*, *pattern*, *thickness*, *scale-thickness*, *mask*, *shape-mode* and *compositing-mode* from *port*'s **`graphics-state`** are all used, unless overridden in *args*. Additionally on X11/Motif only, *stipple* is used. When *filled* is non-nil, a sector is drawn.

See also

draw-arcs
graphics-state
13 Drawing - Graphics Ports

draw-arcs *Function*

Summary

Draws several arcs.

Package

graphics-ports

Signature

draw-arcs *port description* **&rest** *args* **&key** *filled*

Arguments

port⇓	A graphics port.
description⇓	A description sequence.
args⇓	**graphics-state** parameters passed as keyword arguments.
filled⇓	A boolean.

Description

The function **draw-arcs** draws several arcs to *port* as specified by *description*. This is usually more efficient than making several calls to **draw-arc**. *description* is a repeating sequence of values of the form *x y width height start-angle sweep-angle*. See **draw-arc** for more information, including about how *args* and *filled* are used.

See also

draw-arc
graphics-state
13 Drawing - Graphics Ports

draw-character *Function*

Summary

Draws a character in a given graphics port.

Package

graphics-ports

Signature

draw-character *port character x y* **&rest** *args* **&key** *block*

Arguments

port⇓	A graphics port.
character⇓	A character.
x⇓	A real number.
y⇓	A real number.
args⇓	**graphics-state** parameters passed as keyword arguments.
block⇓	A boolean.

Description

The function **draw-character** draws the character *character* at (*x y*) on the port. The *transform*, *foreground*, *background*, *operation*, *stipple*, *pattern*, *mask*, *mask-transform*, *font*, *text-mode* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*.

(*x y*) specifies the leftmost point of the character's baseline.

block, if true, causes the character to be drawn in a character cell filled with the port's **graphics-state** *background*.

Notes

The **graphics-state** parameter *operation* is not supported for drawing text on Windows.

See also

graphics-state
13 Drawing - Graphics Ports

draw-circle *Function*

Summary

Draws a circle.

Package

graphics-ports

Signature

draw-circle *port x y radius* **&rest** *args* **&key** *filled*

Arguments

port⇓	A graphics port.
x⇓	A real number.

y⇓	A real number.
radius⇓	A real number.
args⇓	**graphics-state** parameters passed as keyword arguments.
filled⇓	A boolean.

Description

The function **draw-circle** draws a circle with radius *radius* centered on (*x y*). The *transform*, *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *mask*, *shape-mode* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*. When *filled* is non-nil, the circle is filled with the foreground color.

Notes

draw-circle does not work properly under a rotation transform (see **make-transform**). A workaround is to use a many-sided polygon drawn by **draw-polygon** which will be rotated correctly.

Examples

```
(gp:draw-circle port 100 100 20)
```

```
(gp:draw-circle port 100 100 50
                       :filled t
                       :foreground :green)
```

See also

graphics-state
12 Creating Panes with Your Own Drawing and Input

draw-ellipse *Function*

Summary

Draws an ellipse.

Package

graphics-ports

Signature

draw-ellipse *port x y x-radius y-radius* **&rest** *args* **&key** *filled*

Arguments

port⇓	A graphics port.
x⇓	A real number.
y⇓	A real number.

x-radius⇓	A real number.
y-radius⇓	A real number.
args⇓	**`graphics-state`** parameters passed as keyword arguments.
filled⇓	A boolean.

Description

The function **`draw-ellipse`** draws an ellipse of the given radii *x-radius* and *y-radius* centered on (*x y*). The *transform*, *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *mask*, *shape-mode* and *compositing-mode* from *port*'s **`graphics-state`** are all used, unless overridden in *args*. When *filled* is true, the ellipse is filled with the *foreground* color.

Notes

1. **`draw-ellipse`** does not work properly under a rotation transform when *port*'s *drawing-mode* is **`:compatible`**. A workaround is to use a many-sided polygon drawn by **`draw-polygon`** which will be rotated correctly.
2. **`draw-ellipse`** does work properly under any transform when *port*'s *drawing-mode* is **`:quality`**.
3. See **`make-transform`** for information about rotation transforms.
4. For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

Examples

```
(gp:draw-ellipse port 100 100 20 40)
```

```
(gp:draw-ellipse port 100 100 50 10
                       :filled t
                       :foreground :green)
```

See also

`graphics-state`
13 Drawing - Graphics Ports

draw-image *Function*

Summary

Displays an image on a graphics port at a given position.

Package

`graphics-ports`

Signature

`draw-image` *port image to-x to-y* **`&rest`** *args* **`&key`** *from-x from-y to-width to-height from-width from-height global-alpha*

Arguments

port⇓ A graphics port.

image⇓ An **image**.

to-x⇓, *to-y*⇓ Real numbers.

args⇓ **graphics-state** parameters passed as keyword arguments.

from-x⇓, *from-y*⇓ Real numbers.

to-width⇓, *to-height*⇓ Real numbers.

from-width⇓, *from-height*⇓

Real numbers.

global-alpha⇓ A real number in the inclusive range [0,1], or **nil**.

Description

The function **draw-image** displays *image* on the port at *to-x to-y*. The *transform, operation, mask* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*.

The default values of *from-x* and *from-y* are 0. *from-width* and *from-height* default to the size of *image*. In addition, *to-width* defaults to *from-width* and *to-height* defaults to *from-height*.

When *port*'s *drawing-mode* is **:compatible**, graphics state translation is guaranteed to be supported but support for scaling and rotation are library dependent. Specifically, scaling is supported in the Windows, Cocoa and GTK+ implementations, but not on X11/Motif.

When *port*'s *drawing-mode* is **:quality**, the target coordinates are fully transformed according to the transformation in the **graphics-state**.

For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

global-alpha, if non-nil, is a blending factor that applies to the whole image, in the Windows and Cocoa implementations, but not on X11/Motif or GTK+. The value 0 means use only the target (that is, do not draw anything) and the value 1 means use only the source (that is, normal drawing). Intermediate real values mean use proportions of both the target and source. The value **nil** also means normal drawing, and this is the default value.

Notes

On Microsoft Windows, if the image was loaded from a .ico file then **draw-image** ignores *from-x*, *from-y*, *from-width*, *from-height* and the **graphics-state** *operation* when drawing the image, and also *global-alpha* is ignored.

Compatibility note

In LispWorks 6.1 and earlier versions, *to-width* and *to-height* defaulted to the size of the image and *from-width* defaulted to *to-width* and *from-height* defaulted to *to-height*.

Examples

This example scales an image with various values of *from-width*, *to-width*, *from-height* and *to-height*. It illustrates the effect of the default of these value which has changed since LispWorks 6.1:

```
(example-edit-file "capi/graphics/image-scaling")
```

Further examples:

Draw the whole image at (10 20) without scaling:

```
(gp:draw-image port image 10 20)
```

Draw the whole image at (10 20) scaling it to 100x200:

```
(gp:draw-image port image 10 20
               :to-width 100
               :to-height 200)
```

Draw a 16x32 pixel rectangle from (60 80) in the image at (10 20) without scaling:

```
(gp:draw-image port image 10 20
               :from-x 60
               :from-y 80
               :from-width 16
               :from-height 32)
```

Draw a 16x32 pixel rectangle from (60 80) in the image at (10 20) scaling it to 100x200:

```
(gp:draw-image port image 10 20
               :from-x 60
               :from-y 80
               :from-width 16
               :from-height 32
               :to-width 100
               :to-height 200)
```

See also

image
13 Drawing - Graphics Ports

draw-line *Function*

Summary

Draws a line between two given points.

Package

graphics-ports

Signature

draw-line *port from-x from-y to-x to-y* **&rest** *args*

Arguments

port⇓	A graphics port.
from-x⇓	A real number.
from-y⇓	A real number.
to-x⇓	A real number.

to-y⇓ A real number.

args⇓ **graphics-state** parameters passed as keyword arguments.

Description

The function **draw-line** draws a line from (*from-x from-y*) to (*to-x to-y*).

The *transform*, *foreground*, *background*, *operation*, *pattern*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-end-style*, *mask*, *shape-mode* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*. Additionally on X11/Motif only, *stipple* is used.

See also

draw-lines
graphics-state
13 Drawing - Graphics Ports

draw-lines *Function*

Summary

Draws several lines between pairs of two given points.

Package

graphics-ports

Signature

draw-lines *port description* **&rest** *args*

Arguments

port⇓ A graphics port.

description⇓ A description sequence.

args⇓ **graphics-state** parameters passed as keyword arguments.

Description

The function **draw-lines** draws several lines to *port* as specified by *description*. This is usually more efficient than making several calls to **draw-line**. *description* is a repeating sequence of values of the form *x1 y1 x2 y2*. See **draw-line** for more information, including about how *args* is used.

See also

draw-line
graphics-state
13 Drawing - Graphics Ports

draw-path *Function*

Summary

Draws a path at a given point, optionally closing it or filling it.

Package

`graphics-ports`

Signature

`draw-path` *port path x y* **`&rest`** *args* **`&key`** *closed filled fill-rule*

Arguments

port⇓	A graphics port.
path⇓	A path specification.
x⇓	A real number.
y⇓	A real number.
args⇓	**`graphics-state`** parameters passed as keyword arguments.
closed⇓	A boolean.
filled⇓	A boolean.
fill-rule⇓	One of the keywords **`:even-odd`** and **`:winding`**.

Description

The function **`draw-path`** draws the path *path* at (*x y*) in *port*.

When *closed* is non-nil, a line is drawn from the last point in the path to the start of the last figure in the path. When *filled* is non-nil, the path is filled, otherwise its outline is drawn; *closed* is ignored if *filled* is non-nil. The *transform*, *foreground*, *background*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-end-style, line-joint-style* and *mask* from *port*'s **`graphics-state`** are all used, unless overridden in *args*. *fill-rule* specifies how overlapping regions are filled. Possible values for *fill-rule* are **`:even-odd`** and **`:winding`**.

path is a path specification, which consists of path elements that describe a number of disconnected figures. The origin of the path is (*x y*), so all other coordinates within the path are translated relative to that point.

The following formats of path specification are supported:

- A sequence of lists, each of which is a path element as described below.
- A function designator to generate the path elements. Graphics ports calls the function when it wants to obtain the path elements. The function takes a single argument, which is a function that should be called with each path elements as its arguments.

The following path elements can be used:

`:close`	Closes the current figure by adding a straight line from the current point to the start point.

:move *nx ny* Closes the current figure and starts a new one at (*nx ny*).

:line *nx ny* Adds a straight line to the current figure, from the current point to (*nx ny*) and makes (*nx ny*) be the current point.

:arc *ax ay width height start-angle sweep* **&optional** *movep*

Adds an elliptical arc to the current figure, contained in the rectangle from (*ax ay*) to (*ax*+*width* *ay*+*width*) from *start-angle* to *start-angle*+*sweep-angle*. Both angles are specified in radians and positive values mean anticlockwise. If *movep* is **nil** (the default), then a straight line is also added from the current point to the start of the arc, otherwise a new figure is started from the start of the arc. The end of the arc becomes the new current point.

:bezier *cx1 cy1 cx2 cy2 nx ny*

Adds a cubic Bézier curve to the current figure, from the current point to (*nx ny*) using control points (*cx1 cy1*) and (*cx2 cy2*).

:rectangle *rx ry width height*

Adds a self contained figure, a rectangle from (*rx ry*) to (*rx*+*width ry*+*width*).

:ellipse *ex ey x-radius y-radius*

Adds a self contained figure, an ellipse of the given radii centered on (*ex ey*).

:scale *sx sy elements*

Adds the path elements *elements*, scaling them by *sx* and *sy*.

:rotate *theta elements*

Adds the path elements *elements*, rotating them *theta* radians about the origin. If *theta* is positive, then the rotation is clockwise.

:translate *dx dy elements*

Adds the path elements *elements*, translating them by *dx* and *dy*.

:transform *transform elements*

Adds the path elements *elements*, transformed by *transform*.

Examples

Draws two lines from (40 30) to (140 30) and from (140 30) to (140 130):

```
(draw-path port '((:line 100 0) (:line 100 100)) 40 30)
```

Draws an outline triangle with vertices (40 30), (140 30) and (140 130):

```
(draw-path port '((:line 100 0) (:line 100 100))
           40 30 :closed t)
```

Draws a filled triangle with vertices (40 30), (140 30) and (140 130):

```
(draw-path port '((:line 100 0) (:line 100 100))
```

```
              40 30 :filled t)
```

Draws a filled triangle exactly as in the previous example but using a function to generate the path elements:

```
(flet ((generate (fn)
          (funcall fn :line 100 0)
          (funcall fn :line 100 100)))
  (draw-path port #'generate 40 30 :filled t))
```

Draws 6 copies of a shape consisting of two lines and an arc:

```
(labels ((generate-1 (fn)
           (funcall fn :line 50 0)
           (funcall fn :line 50 50)
           (funcall fn :arc 0 -50 100 100
                    (/ pi -2) (/ pi -2)))
         (generate-6 (fn)
           (dotimes (x 6)
             (funcall fn :rotate (* 2 pi (/ x 6))
                      #'generate-1))))
  (draw-path port #'generate-6 80 80))
```

There are more examples in:

```
(example-edit-file "capi/graphics/paths")
```

There are further examples in **20 Self-contained examples**.

See also

draw-polygon
draw-line
draw-arc
draw-ellipse
graphics-state
13 Drawing - Graphics Ports

draw-point *Function*

Summary

Draws a pixel or unit square at a given point.

Package

graphics-ports

Signature

draw-point *port x y* **&rest** *args*

Arguments

port⇓ A graphics port.

x⇓	A real number.
y⇓	A real number.
args⇓	**graphics-state** parameters passed as keyword arguments.

Description

The function **draw-point** draws a single-pixel point at (*x y*). The *transform*, *foreground*, *background*, *operation*, *mask*, *pattern*, *shape-mode* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*. Additionally on X11/Motif only, *stipple* is used.

When *drawing-mode* is **:compatible** the output is a single pixel. Note that its position is transformed in the normal way.

When *drawing-mode* is **:quality** this draws a unit square as if by **draw-rectangle**, transformed in the normal way.

See also

draw-points
graphics-state

draw-points *Function*

Summary

Draws pixels or unit squares at given points.

Package

graphics-ports

Signature

draw-points *port description* **&rest** *args*

Arguments

port⇓	A graphics port.
description⇓	A description sequence.
args⇓	**graphics-state** parameters passed as keyword arguments.

Description

The function **draw-points** draws several points in *port* (as if by **draw-point**) as specified by *description*, which is a sequence of *x y* pairs. It is usually faster than several calls to **draw-point**. See **draw-point** for more information, including about how *args* are used.

See also

draw-point

draw-polygon *Function*

Summary

Draws a polygon.

Package

`graphics-ports`

Signature

`draw-polygon` *port points* **`&rest`** *args* **`&key`** *filled closed fill-rule*

Arguments

port⇓ A graphics port.

points⇓ A description sequence.

args⇓ **`graphics-state`** parameters passed as keyword arguments.

filled⇓ A boolean.

closed⇓ A boolean.

fill-rule⇓ A keyword.

Description

The function **`draw-polygon`** draws a polygon using alternating *x* and *y* values in *points* as the vertices. When *closed* is true the edge from the last vertex to the first to be drawn. When *filled* is true a filled, closed polygon is drawn; *closed* is ignored if *filled* is true.

The *transform*, *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-end-style*, *line-joint-style*, *mask*, *pattern*, *shape-mode* and *compositing-mode* from *port*'s **`graphics-state`** are all used, unless overridden in *args*. Additionally on X11/Motif only, *stipple* is used.

fill-rule specifies how overlapping regions are filled. Possible values are **`:even-odd`** and **`:winding`**.

See also

`draw-polygons`
`graphics-state`
13 Drawing - Graphics Ports

draw-polygons *Function*

Summary

Draws several polygons.

Package

graphics-ports

Signature

draw-polygons *port description* **&rest** *args* **&key** *filled closed fill-rule*

Arguments

port⇓	A graphics port.
description⇓	A sequence of sequences of real numbers.
args⇓	**graphics-state** parameters passed as keyword arguments.
filled⇓	A boolean.
closed⇓	A boolean.
fill-rule⇓	A keyword.

Description

The function **draw-polygons** draws several polygons in *port*. *description* should be a sequence containing sequences with alternating *x* and *y* values representing the vertices. *description* consists of groups of *points* as in **draw-polygon**.

When *closed* is true the edge from the last vertex to the first to be drawn.

When *filled* is true a filled, closed polygons are drawn; *closed* is ignored if *filled* is true.

The *transform*, *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-end-style*, *line-joint-style*, *mask*, *pattern*, *shape-mode* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*. Additionally on X11/Motif only, *stipple* is used.

fill-rule specifies how overlapping regions are filled. Possible values are **:even-odd** and **:winding**.

Examples

This draws two hexagons, one inside the other:

```
(gp:draw-polygons oo
           '((150 100 200 100 235 150 200
              200 150 200  115 150)
             (140 90  210 90  250 150
              210 210  140 210  100 150))
           :closed t)
```

See also

draw-polygon
graphics-state
13 Drawing - Graphics Ports

draw-rectangle *Function*

Summary

Draws a rectangle.

Package

`graphics-ports`

Signature

`draw-rectangle` *port x y width height* **`&rest`** *args* **`&key`** *filled*

Arguments

port⇓	A graphics port.
x⇓	A real number.
y⇓	A real number.
width⇓	A real number.
height⇓	A real number.
args⇓	**`graphics-state`** parameters passed as keyword arguments.
filled⇓	A boolean.

Description

The function **`draw-rectangle`** draws a rectangle whose corners are (*x y*), (*x*+*width y*), (*x*+*width y*+*height*) and (*x y*+*height*).

filled, if non-nil, causes a filled rectangle to be drawn. While the exact results are host-specific, it is intended that a filled rectangle does not include the lines where the x coordinate is *x*+*width* or the y coordinate is *y*+*height* while a non-filled rectangle does. This function works correctly if *port*'s transform includes rotation.

The *transform*, *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-joint-style*, *mask*, *pattern*, *shape-mode* and *compositing-mode* from *port*'s **`graphics-state`** are all used, unless overridden in *args*. Additionally on X11/Motif only, *stipple* is used.

See also

`draw-rectangles`
`graphics-state`
13 Drawing - Graphics Ports

draw-rectangles *Function*

Summary

Draws several rectangles.

Package

graphics-ports

Signature

draw-rectangles *port description* **&rest** *args* **&key** *filled*

Arguments

port⇓ A graphics port.

description⇓ A description sequence.

args⇓ **graphics-state** parameters passed as keyword arguments.

filled⇓ A boolean.

Description

The function **draw-rectangles** draws several rectangles as specified by *description*. This is usually more efficient than making several calls to **draw-rectangle**. *description* is a repeating sequence of values of the form *x y width height*.

filled, if true, causes filled rectangles to be drawn. While the exact results are host-specific, it is intended that a filled rectangle does not include the lines where the x coordinate is *x*+*width* or the y coordinate is *y*+*height* while a non-filled rectangle does. This function works correctly if *port*'s transform includes rotation.

The *transform*, *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-joint-style*, *mask*, *pattern*, *shape-mode* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*. Additionally on X11/Motif only, *stipple* is used.

See also

draw-rectangle
graphics-state
13 Drawing - Graphics Ports

draw-string *Function*

Summary

Draws a string with the baseline positioned at a given point.

Package

graphics-ports

Signature

draw-string *port string x y* **&rest** *args* **&key** *start end block*

Arguments

port⇓ A graphics port.

string⇓ A string.

x⇓ A real number.

y⇓ A real number.

args⇓ **graphics-state** parameters passed as keyword arguments.

start⇓ A real number.

end⇓ A real number.

block⇓ A boolean.

Description

The function **draw-string** draws the string *string* with the baseline starting at (*x y*). The *transform*, *foreground*, *background*, *operation*, *stipple*, *pattern*, *mask*, *mask-transform*, *font*, *text-mode* and *compositing-mode* from *port*'s **graphics-state** are all used, unless overridden in *args*.

start and *end* specify which elements of *string* to draw. The default value of *start* is 0.

block, if true, causes each character to be drawn in a character cell filled with the *background* of *port*'s **graphics-state**.

You can draw with the system highlight by setting **graphics-state** parameter *foreground* **:color_highlighttext** and *background* **:color_highlight**.

Notes

The **graphics-state** parameter *operation* is not supported for drawing text on Microsoft Windows.

Examples

```
(let ((op (capi:contain
           (make-instance 'capi:output-pane
                          :background :red))))
  (gp:draw-string op "highlighted"
                  10 10
                  :graphics-args
                  (list :foreground
                        :color_highlighttext)))
```

See also

graphics-state
13 Drawing - Graphics Ports

ensure-gdiplus *Function*

Summary

Ensures GDI+ is present and running, or shuts it down. Needed only when writing FLI graphics code on Windows.

Package

`graphics-ports`

Signature

`ensure-gdiplus &key` *event-func force shutdown* **`=>`** *result*

Arguments

event-func⇓ A function, or **`nil`**.

force⇓ A boolean.

shutdown⇓ A boolean.

Values

result A boolean.

Description

The function **`ensure-gdiplus`** checks that the GDI+ module **`gdiplus.dll`** is loaded and that **`GdiplusStartup`** has been called, or shuts down GDI+.

Most users will not need to call **`ensure-gdiplus`**. This is because when LispWorks itself uses GDI+, for instance via **`read-external-image`**, it calls **`ensure-gdiplus`** automatically, and never shuts GDI+ down.

However, if your code uses GDI+ directly (by calling it through the Foreign Language Interface), then you should call **`ensure-gdiplus`** instead of using **`GdiplusStartup`** directly. Then, LispWorks will know that GDI+ has already started. This is the only circumstance in which you need to call **`ensure-gdiplus`**.

Note: **`ensure-gdiplus`** is implemented only in LispWorks for Windows.

If *shutdown* is **`nil`**, **`ensure-gdiplus`** ensures GDI+ is started, by the following steps:

1. Load the GDI+ module **`gdiplus.dll`**, if it is not already loaded.
2. If GDI+ was already started by a previous call to **`ensure-gdiplus`**, *force* is **`nil`**, and *event-func* was either not passed or is **`cl:eq`** to the value that was passed in the previous call to **`ensure-gdiplus`** then **`ensure-gdiplus`** simply returns **`nil`**.
3. If GDI+ was already started, shut it down.
4. Start GDI+, and return the result of **`GdiplusStartup`**. This is 0 for success. For he meaning of other values, see the documentation of **`gpStatus`** in the MSDN.

If *shutdown* is true, then if GDI+ was started **`ensure-gdiplus`** shuts it down, and returns **`t`**, otherwise **`ensure-gdiplus`** returns **`nil`**. The default value of *shutdown* is **`nil`**.

The default value of both *event-func* and *force* is **`nil`**.

See also

`read-external-image`

external-image *System Class*

Summary

A class representing a color image.

Package

`graphics-ports`

Superclasses

`t`

Description

The system class **`external-image`** provides a representation of a color image that is subject to **`write-external-image`**, **`read-external-image`** and **`convert-external-image`** operations.

See also

`convert-external-image`
`read-external-image`
`write-external-image`
13 Drawing - Graphics Ports

external-image-color-table *Accessor*

Summary

Returns a vector containing RGB color specifications of an external image.

Package

`graphics-ports`

Signature

`external-image-color-table` *external-image* **`=>`** *color-table*

`setf (external-image-color-table` *external-image***`)`** *color-table* **`=>`** *color-table*

Arguments

external-image⇓ An **`external-image`**.

color-table	A color table.

Values

color-table	A color table.

Description

The accessor **external-image-color-table** gets and sets a vector containing RGB color specifications representing the color table as specified in *external-image*.

external-image must be a plain **external-image**. See **13.10 Working with images** for details.

If the result is **nil**, the external image is a 24-bit DIB, with the colors defined in each pixel instead of through a table.

When setting the color-table of an external image, the new color-table must be the same length as the external image's original color table.

externalize-and-write-image *Function*

Summary

Externalizes and writes an image to file.

Package

graphics-ports

Signature

externalize-and-write-image *gp image destination* **&key** *type if-exists errorp x-hot y-hot quality* **&allow-other-keys =>** *result*

Arguments

gp⇓	A CAPI pane.
image⇓	An **image** object.
destination⇓	A file namestring, a pathname or an open output stream with element type compatible with **(unsigned-byte 8)**, i.e. **base-char**, **(signed-byte 8)** or **(unsigned-byte 8)**.
type⇓	One of the keywords **:bmp**, **:jpg**, **:jpeg**, **:png** and **:tiff**. Other keywords may be supported, depending on the platform.
if-exists⇓	One of the keywords **:error**, **:new-version**, **:rename**, **:rename-and-delete**, **:overwrite**, **:append** and **:supersede**, or **nil**.
errorp⇓	A boolean.
x-hot⇓	A non-negative integer.
y-hot⇓	A non-negative integer.
quality⇓	An integer in the range [0,100].

Values

result⇓ A filename or **`nil`**.

Description

The function **`externalize-and-write-image`** externalizes and writes an **`image`** object to a file or stream.

image should be an image that can be drawn to *gp*. The bytes of *image* are written to *destination* as if by **`write-sequence`**.

The output image type can be specified by *type*. If *type* is not supplied then the output image type is determined by the file type of *destination*.

If *type* is supplied, it must be a keyword which specifies a known type, as returned by **`list-known-image-formats`** with *for-writing-too* **`t`**. The types **`:bmp`**, **`:jpg`**, **`:png`** and **`:tiff`** are known on all platforms (except Motif). Additionally, **`:jpeg`** is an as alias for **`:jpg`**.

If *type* is not supplied, then the file extension of *destination* is used to "guess" the type. In general it is the extension uppercased and interned in the keyword package. It also recognizes some special cases:

Image type from file extension: special cases

File extension	Image type
"TIF"	**`:tiff`**
"DIB"	**`:bmp`**
"JPE"	**`:jpg`**
"JPEG"	**`:jpg`**
"JFIF"	**`:jpg`**
"JP2"	**`:jpg2000`**

Note: Image type **`:jpg2000`** is implemented on Cocoa only.

errorp controls what happens if **`externalize-and-write-image`** does not recognize the type. If *errorp* is non-nil, it calls **`error`**, otherwise it returns **`nil`**. The default value of *errorp* is **`t`**.

if-exists controls what to do if *destination* already exists, in the same way as the *if-exists* argument to **`open`**. However, unlike **`open`**, the default value of *if-exists* is **`:supersede`**.

x-hot and *y-hot* are used only when generating a CUR file, which is currently implemented on GTK+ only. They specify the hotspot coordinates when the image is used as a cursor (in a LispWorks application by **`load-cursor`** and **`(setf capi:simple-pane-cursor)`**, or in other applications). Their values must be integers within the width/height of the image. The default value of both *x-hot* and *y-hot* is 0.

quality is used for writing a JPG image on GTK+. It must be an integer in the inclusive range [0,100]. High values generate better images and larger files.

result is *destination* on success, or **`nil`** for an unknown type when *errorp* is **`nil`**. It signals an error in other cases (for example, failure to open the file because of permissions).

Examples

There is a simple example in:

```
(example-edit-file "capi/graphics/images-with-alpha")
```

See also

list-known-image-formats
13 Drawing - Graphics Ports

externalize-image *Function*

Summary

Returns an external image containing color information from an image.

Package

graphics-ports

Signature

externalize-image *gp image* **&key** *maximum-colors important-colors type quality* **&allow-other-keys =>** *external-image*

Arguments

gp⇓	A CAPI pane.
image⇓	An image.
maximum-colors⇓	An integer or **nil**. The default is **nil**.
important-colors⇓	An integer or **nil**.
type⇓	One of the keywords **:bmp**, **:jpg**, **:jpeg**, **:png** and **:tiff**. Other keywords may be supported, depending on the platform.
quality⇓	An integer in the range [0,100].

Values

external-image⇓	An external image.

Description

The function **externalize-image** returns an **external-image** containing color information from *image*, which should be an image that can be drawn to *gp*.

If *maximum-colors* is **nil** or if the screen has no palette, an **external-image** using all the colors in *image* is created.

If *maximum-colors* is an integer, the **external-image** containing image will be created using no more than that number of colors. If the image contains more than *maximum-colors* colors, then *maximum-colors* most frequently used colors will be accurately stored; the remainder will be approximated by nearest colors out of the accurate ones, using internal Color System parameters as the weighting factors for the color distance.

important-colors is recorded in *external-image* for later use, and specifies the number of colors required to draw a good likeness of the image. The default value is the number of colors in the image.

If *type* is supplied, it must be a keyword which specifies a known type, as returned by **list-known-image-formats** with *for-writing-too* **t**. The types **:bmp**, **:jpg**, **:png** and **:tiff** are known on all platforms (except Motif). Additionally, **:jpeg** is an as alias for **:jpg**.

quality is used for writing a JPG image on GTK+. It must be an integer in the inclusive range [0,100]. High values generate better images and larger files.

See also

make-image-from-port
write-external-image
13 Drawing - Graphics Ports

f2pi *Constant*

Summary

(* 2 pi) as a **single-float**.

Package

graphics-ports

Description

The constant **f2pi** is the result of **(float (* 2.0 cl:pi) 1.0)**. It is a **cl:single-float**.

See also

fpi
fpi-by-2

find-best-font *Function*

Summary

Returns the best font for a CAPI pane.

Package

graphics-ports

Signature

find-best-font *pane fdesc* **=>** *font*

Arguments

pane⇓ A graphic port.

fdesc⇓ A font description.

Values

font A font.

Description

The function **find-best-font** returns the best font for *pane* which matches *fdesc*. When there alternative fonts available the choice of best font is operating system dependent.

When *fdesc* contains the attribute **:stock** with value **:system-font** or **:system-fixed-font**, the lookup will always find a stock font.

By default **find-best-font** looks only for Truetype fonts in LispWorks 6.1 and later.

Notes

With the default *drawing-mode* **:quality** only Truetype fonts are supported. Non-Truetype fonts are supported only when using *drawing-mode* **:compatible**.

Compatibility note

To get the LispWorks 6.0 behavior where non-Truetype fonts are also found, pass **:type :wild** to **make-font-description**.

Examples

```
(example-edit-file "capi/graphics/catherine-wheel")
```

See also

find-matching-fonts
make-font-description
prompt-for-font
13 Drawing - Graphics Ports

find-matching-fonts *Function*

Summary

Returns a list of the font objects available for a pane.

Package

graphics-ports

Signature

find-matching-fonts *pane fdesc* **=>** *fonts*

Arguments

pane⇓ A CAPI pane.

fdesc⇓ A font description.

Values

fonts A list of fonts.

Description

The function **`find-matching-fonts`** returns a list of the font objects available for *pane* which match the attributes in *fdesc*. **`nil`** is returned if none match.

When *fdesc* contains the attribute **`:stock`** with value **`:system-font`** or **`:system-fixed-font`**, the lookup will always find a stock font.

`find-matching-fonts` behaves as if the **`:family`**, **`:weight`**, **`:slant`** and **`:size`** attributes have value **`:wild`** if they are missing from *fdesc*.

See also

`find-best-font`
`list-all-font-names`
`make-font-description`
13 Drawing - Graphics Ports

font *Type*

Summary

An object corresponding to a font in the native system.

Package

`graphics-ports`

Signature

`font`

Description

The type **`font`** is the type of objects are returned by **`find-best-font`** and **`find-matching-fonts`**.

`font` objects are used to specify fonts for drawing, either in the **`graphics-state`** of the port or in the drawing functions themselves. **`font`** objects can also be used for querying the actual attributes of the font (ascent, descent and so on) and the dimensions of character and strings.

Notes

`font` objects are not externalizable objects.

See also

`font-description`
`find-best-font`

find-matching-fonts
graphics-state
get-font-ascent
get-font-descent
get-font-width
get-font-height
get-font-average-width
get-char-width
get-char-ascent
get-char-descent
get-character-extent
get-string-extent
compute-char-extents
font-single-width-p
font-fixed-width-p
font-dual-width-p

font-description *Function*

Summary

Returns a font description object for a given font.

Package

graphics-ports

Signature

font-description *font* **=>** *fdesc*

Arguments

font⇓	A font.

Values

fdesc	A font description.

Description

The function **font-description** returns a font description object for *font*. Using this font description in a later call to **find-matching-fonts** or **find-best-font** on the original pane is expected to return a similar font.

See also

convert-to-font-description
make-font-description
font-description

font-description *Type*

Summary

An object used in CAPI to describe a font.

Package

graphics-ports

Signature

font-description

Description

The type **font-description** is used for objects that contain a description of a font. The description can be partial, with only some attributes given values. **font-description** objects are the normal way of specifying fonts in CAPI.

font-description objects are created or returned by **make-font-description**, **convert-to-font-description**, **font-description**, **merge-font-descriptions** and **augment-font-description**.

font-description objects are used as the font specification for CAPI panes (see **simple-pane**). They can also be used directly in calls to **find-best-font** and **find-matching-fonts**.

Notes

1. **font-description** objects do not contain native system dependent values, and are externalizable objects.
2. A **font-description** cannot be used directly as an argument to **draw-string** or **draw-character**, or as the value of the graphics state parameter *font* in a **graphics-state**. These require the result of **find-best-font** or **find-matching-fonts**.

See also

make-font-description
convert-to-font-description
merge-font-descriptions
augment-font-description
font-description-attributes
find-best-font
find-matching-fonts
3 General Properties of CAPI Panes

font-description-attributes *Function*

Summary

Returns the attributes of a given font description.

Package

`graphics-ports`

Signature

`font-description-attributes` *fdesc* **`=>`** *font-attributes*

Arguments

fdesc⇓	A font description.

Values

font-attributes	A list of font attributes.

Description

The function **`font-description-attributes`** returns the attributes of *fdesc*. The list should not be destructively modified.

See also

`font-description-attribute-value`

font-description-attribute-value *Function*

Summary

Returns the values of a given font attribute in a font description.

Package

`graphics-ports`

Signature

`font-description-attribute-value` *fdesc* *font-attribute* **`=>`** *value*

Arguments

fdesc⇓	A font description.
font-attribute⇓	A font attribute.

Values

value	A font attribute value.

Description

The function **`font-description-attribute-value`** returns the value of *font-attribute* in *fdesc*, or **`:wild`** if *font-attribute* is not specified in *fdesc*.

See also

font-description-attributes

font-dual-width-p *Function*

Summary

The predicate for dual-width fonts. This function is deprecated.

Package

graphics-ports

Signature

font-dual-width-p *port* **&optional** *font* **=>** *result*

Arguments

port⇓	A graphics port.
font⇓	A **font** object.

Values

result	A boolean.

Description

The function **font-dual-width-p** returns **t** if *font* is fixed-width and contains double width characters. Such a font is dual-width. *font* defaults to the font associated with *port*.

See also

font-fixed-width-p

font-fixed-width-p *Function*

Summary

The predicate for fixed-width fonts.

Package

graphics-ports

Signature

font-fixed-width-p *port* **&optional** *font* **=>** *result*

Arguments

port⇓ A graphics port.

font⇓ A **font** object.

Values

result A boolean.

Description

The function **font-fixed-width-p** returns **t** if *font* is fixed-width. *font* defaults to the font associated with *port*.

Fixed-width is not exactly the same as single-width. A fixed-width font with double width characters is dual-width; other fixed-width fonts are single-width.

Notes

editor-pane supports variable width fonts on Microsoft Windows, GTK+ and Motif.

See also

font-dual-width-p

font-single-width-p *Function*

Summary

The predicate for single-width fonts. This function is deprecated.

Package

graphics-ports

Signature

font-single-width-p *port* **&optional** *font* **=>** *result*

Arguments

port⇓ A graphics port.

font⇓ A **font** object.

Values

result A boolean.

Description

The function **font-single-width-p** returns **t** when all characters in the font specified by *font* are of the same width. *font* defaults to the font associated with *port*.

A single-width font is fixed-width.

See also

font-fixed-width-p
font-dual-width-p

fpi *Constant*

Summary

pi as a **single-float**.

Package

graphics-ports

Description

The constant **fpi** is the result of **(float cl:pi 1.0)**. It is a **cl:single-float**.

See also

2pi
f2pi
fpi-by-2

fpi-by-2 *Constant*

Summary

(/ pi 2) as a **single-float**.

Package

graphics-ports

Description

The constant **fpi-by-2** is the result of **(float (* 0.5 cl:pi) 1.0)**. It is a **cl:single-float**.

See also

fpi
f2pi

free-image *Function*

Summary

Frees the library resources allocated with an image.

Package

graphics-ports

Signature

free-image *port image*

Arguments

port⇓	A CAPI pane.
image⇓	An image.

Description

The function **free-image** frees the library resources associated with *image*. This should be done when an image is no longer needed.

port should be the pane used when the image was created, for example by **load-image**.

See also

13 Drawing - Graphics Ports
17 Drag and Drop

free-image-access *Function*

Summary

Frees an Image Access object.

Package

graphics-ports

Signature

free-image-access *image-access*

Arguments

image-access⇓	An Image Access object.

Description

The function **`free-image-access`** discards *image-access*, which should be an Image Access object returned by **`make-image-access`**.

See also

`image-access-transfer-from-image`
`image-access-transfer-to-image`
`image-access-pixel`
`make-image-access`
13.10.8 Image access

get-bounds *Function*

Summary

Returns the four values of the currently collected drawing extremes.

Package

`graphics-ports`

Signature

`get-bounds` *pixmap-port* **`=>`** *left*, *top*, *right*, *bottom*

Arguments

pixmap-port⇓ A graphics port.

Values

left⇓ An integer.

top⇓ An integer.

right⇓ An integer.

bottom⇓ An integer.

Description

The function **`get-bounds`** returns the four values *left*, *top*, *right*, *bottom* of the currently collected drawing extremes in *pixmap-port*. The values can be used to get an image from the port.

Drawing extremes are collected by passing non-nil for the *collect* or *relative* arguments to **`create-pixmap-port`** or **`with-pixmap-graphics-port`**.

Examples

```
(with-pixmap-graphics-port (p1 pane width height
                            :relative t)
  (with-graphics-rotation (p1 0.123)
  (draw-rectangle p1 100 100 200 120 :filled t
```

```
                    :foreground :red)
     (get-bounds p1)))
```

produces the following output:

```
72
112
285
255
```

See also

create-pixmap-port
make-image-from-port
with-pixmap-graphics-port

get-character-extent *Function*

Summary

Returns the extent of a character in pixels.

Package

graphics-ports

Signature

get-character-extent *port character* **&optional** *font* **=>** *left*, *top*, *right*, *bottom*

Arguments

port⇓	A CAPI pane.
character⇓	A character.
font⇓	A font.

Values

left	An integer.
top	An integer.
right	An integer.
bottom	An integer.

Description

The function **get-character-extent** returns the extent in pixels of *character* in *font*.

font defaults to the font associated with *port*.

get-char-ascent *Function*

Summary

Returns the ascent of a character in pixels.

Package

graphics-ports

Signature

get-char-ascent *port character font* **=>** *ascent*

Arguments

port⇓	A CAPI pane.
character⇓	A character.
font⇓	A font.

Values

ascent	An integer.

Description

The function **get-char-ascent** returns the ascent in pixels of *character* in in *font*.

font defaults to the font associated with *port*.

get-char-descent *Function*

Summary

Returns the descent of a character in pixels.

Package

graphics-ports

Signature

get-char-descent *port character font* **=>** *descent*

Arguments

port⇓	A CAPI pane.
character⇓	A character.

font⇓ A font.

Values

descent An integer.

Description

The function **`get-char-descent`** returns the descent in pixels of *character* in *font*.

font defaults to the font associated with *port*.

get-char-width *Function*

Summary

Returns the width of a character in pixels.

Package

`graphics-ports`

Signature

`get-char-width` *port character font* **`=>`** *width*

Arguments

port⇓ A CAPI pane.

character⇓ A character.

font⇓ A font.

Values

width An integer.

Description

The function **`get-char-width`** returns the width in pixels of *character* in *font*.

font defaults to the font associated with *port*.

get-enclosing-rectangle *Function*

Summary

Returns the smallest rectangle enclosing the given points.

Package

`graphics-ports`

Signature

`get-enclosing-rectangle &rest` *points* **`=>`** *left*, *top*, *right*, *bottom*

Arguments

points⇓	Real numbers.

Values

left	A real number.
top	A real number.
right	A real number.
bottom	A real number.

Description

The function **`get-enclosing-rectangle`** returns four values, describing the rectangle which exactly encloses the input points. *points* must be a (possibly empty) list of alternating *x* and *y* values. If no *points* are given the function returns the null (unspecified) rectangle, which is four **`nil`**s.

get-font-ascent *Function*

Summary

Returns the ascent of a font.

Package

`graphics-ports`

Signature

`get-font-ascent` *port* **`&optional`** *font* **`=>`** *ascent*

Arguments

port⇓	A CAPI pane.
font⇓	A font.

Values

ascent	An integer.

Description

The function **`get-font-ascent`** returns the ascent in pixels of *font*.

font defaults to the font associated with *port*.

get-font-average-width *Function*

Summary

Returns the average width of a font in pixels.

Package

graphics-ports

Signature

get-font-average-width *port* **&optional** *font* **=>** *average-width*

Arguments

port⇓	A CAPI pane.
font⇓	A font.

Values

average-width	An integer.

Description

The function **get-font-average-width** returns average width in pixels of *font*.

font defaults to the font associated with *port*.

See also

13 Drawing - Graphics Ports

get-font-descent *Function*

Summary

Returns the descent in pixels of a font.

Package

graphics-ports

Signature

get-font-descent *port* **&optional** *font* **=>** *descent*

Arguments

port⇓ A CAPI pane.

font⇓ A font.

Values

descent An integer.

Description

The function **get-font-descent** returns the descent in pixels of *font*.

font defaults to the font associated with *port*.

get-font-height *Function*

Summary

Returns the height of a font.

Package

graphics-ports

Signature

get-font-height *port* **&optional** *font* **=>** *height*

Arguments

port⇓ A CAPI pane.

font⇓ A font.

Values

height An integer.

Description

The function **get-font-height** returns the height in pixels of *font*.

font defaults to the font associated with *port*.

See also

13 Drawing - Graphics Ports

get-font-width *Function*

Summary

Returns the width of a font.

Package

graphics-ports

Signature

get-font-width *port* **&optional** *font* **=>** *width*

Arguments

port⇓	A graphics port.
font⇓	A font.

Values

width	An integer.

Description

The function **get-font-width** returns the width in pixels of *font*.

font defaults to the font associated with *port*.

See also

13 Drawing - Graphics Ports

get-graphics-state *Function*

Summary

Returns the **graphics-state** object for a graphics port. Deprecated, use **port-graphics-state** instead.

Package

graphics-ports

Signature

get-graphics-state *port* **=>** *state*

Arguments

port⇓	A graphics port.

Values

state	A **graphics-state** object.

Description

The function **get-graphics-state** returns the **graphics-state** object of *port*. **get-graphics-state** is deprecated. Use **port-graphics-state** instead.

See also

port-graphics-state

get-origin *Function*

Summary

Returns the coordinate origin of a pixmap graphics port.

Package

graphics-ports

Signature

get-origin *pixmap-port* **=>** *x*, *y*

Arguments

pixmap-port⇓	A graphics port.

Values

x	An integer.
y	An integer.

Description

The function **get-origin** returns the coordinate origin or *pixmap-port*. Normally this is (0 0) but after a series of drawing function calls with **:relative t**, the drawing may have been shifted. The values returned by **get-origin** tell you by how much. The values are *not* needed when making images from the port's drawing.

Examples

```
(with-pixmap-graphics-port (p1 pane width height :relative t)
    (with-graphics-rotation (p1 0.123)
      (draw-rectangle p1 0 0 200 120 :filled t
                                     :foreground :red)
      (get-origin p1)))
```

produces:

```
-15
0
```

get-string-extent *Function*

Summary

Returns the extent in pixels of a string.

Package

graphics-ports

Signature

get-string-extent *port string* **&optional** *font* **=>** *left*, *top*, *right*, *bottom*

Arguments

port⇓	A CAPI pane.
string⇓	A string.
font⇓	A font.

Values

left	An integer.
top	An integer.
right	An integer.
bottom	An integer.

Description

The function **get-string-extent** returns the extent in pixels of *string* in *font*.

font defaults to the font associated with *port*.

Note: To compute the horizontal extents of each successive character in a string for a given port or font, use **compute-char-extents**.

See also

compute-char-extents

get-transform-scale *Function*

Summary

Returns the overall scaling factor of a transform.

Package

graphics-ports

Signature

get-transform-scale *transform* **=>** *result*

Arguments

transform⇓ A **transform** object.

Values

result A real number.

Description

The function **get-transform-scale** returns a single number representing the overall scaling factor present in *transform*.

Notes

See **graphics-state** for details of how a **transform** is used.

See also

graphics-state
transform

graphics-port-background
graphics-port-font
graphics-port-foreground
graphics-port-transform

Accessors

Summary

Accesses the *background*, *font*, *foreground* or *transform* in the graphics state of a graphics port.

Package

graphics-ports

Signatures

graphics-port-background *port* **=>** *background*

(setf graphics-port-background) *background port* **=>** *background*

graphics-port-font *port* **=>** *font*

(setf graphics-port-font) *font port* **=>** *font*

graphics-port-foreground *port* **=>** *foreground*

(setf graphics-port-foreground) *foreground port* **=>** *foreground*

graphics-port-transform *port* **=>** *transform*

(setf graphics-port-transform) *transform port* **=>** *transform*

Arguments

port⇓ A graphics port.

background⇓ A color specification, or **nil**.

font⇓ A **font** object, or **nil**.

foreground⇓ A color specification, or **nil**.

transform⇓ A **transform** object.

Values

background⇓ A color specification, or **nil**.

font⇓ A **font** object, or **nil**.

foreground⇓ A color specification, or **nil**.

transform⇓ A **transform** object.

Description

The accessors **graphics-port-background**, **graphics-port-font**, **graphics-port-foreground** and **graphics-port-transform** access the current *background*, *font*, *foreground* or *transform* in the **graphics-state** associated with *port*. This can be used to set the value by **setf**.

See the **graphics-state** entry for the types and acceptable values of the various slots, and information about how they are used.

See also

graphics-state
port-graphics-state
set-graphics-state
transform
with-graphics-state

graphics-port-mixin *Class*

Summary

An abstract class supporting Graphics Ports operations.

Package

graphics-ports

Superclasses

standard-object

Subclasses

output-pane
pixmap-port
printer-port
metafile-port

Description

The class **graphics-port-mixin** is an abstract class for supporting graphics ports operations. All the classes that support drawing (generally referred to as "graphics ports") inherit from it.

See also

13 Drawing - Graphics Ports

graphics-state *System Class*

Summary

The graphics state object, holding default parameters for drawing operations on an associated *port*.

Package

graphics-ports

Superclasses

t

Accessors

graphics-state-transform
graphics-state-foreground
graphics-state-background
graphics-state-operation
graphics-state-stipple
graphics-state-pattern
graphics-state-thickness
graphics-state-scale-thickness
graphics-state-dashed
graphics-state-dash
graphics-state-fill-style
graphics-state-line-end-style
graphics-state-line-joint-style
graphics-state-mask
graphics-state-mask-x
graphics-state-mask-y
graphics-state-mask-transform
graphics-state-font
graphics-state-text-mode

graphics-state-shape-mode
graphics-state-compositing-mode

Description

The system class **graphics-state** contains the default values of graphics parameters for drawing operations. Each graphics port has a **graphics-state** object associated with it. The drawing operations such as **draw-ellipse**, **draw-rectangle** and **draw-string** can override specific parameters by passing them as keyword arguments.

graphics-state objects are used in the **with-graphics-state** macro and modified using the accessor functions listed above. See **13.3.1 Setting the graphics state** for examples.

graphics-state contains the following properties:

transform — A **transform** object which determines the coordinate transformation applying to the graphics port. The default value is the unit transform which leaves the port coordinates unchanged from those used by the host window system — origin at top left, X increasing to the right and Y increasing down the screen. Allowed values are anything returned by the transform functions, described in **13.6 Graphics state transforms**.

foreground — Determines the foreground color used in drawing functions. The value can be a converted color (result of **convert-color**), a color name symbol, a color name string or a color spec object. Using converted colors results in better performance, because it saves the system from doing the conversion each time it uses it. The default value is **:black**. The value **:color_highlighttext** is useful for drawing text with the system highlighting.

background — Determines the background color used in functions which draw text such as **draw-string** when *block* is true.

On X11/Motif, *background* also determines the background color used in drawing functions which use a stipple.

Valid values are the same as for *foreground*. The default value is **:white**. The value **:color_highlight** is useful for drawing text with the system highlighting.

operation — Determines the color combination used in the drawing primitives when the *port*'s *drawing-mode* is **:compatible**. Valid values are 0 to 15, being the same logical values as the *op* arg to the Common Lisp function **boole**. The default value is **boole-1**. **13.7.1 Combining pixels with :compatible drawing** shows how to use *operation*.

stipple — On X11/Motif *stipple* is a 1-bit pixmap ("bitmap") or **nil** (which is the default value). The bitmap is used in conjunction with the *fill-style* when drawing. Here, **nil** means that all pixels are drawn in the *foreground* color. A stipple is not transformed by the *transform* parameter. Its origin is assumed to coincide with the origin of the port. The *stipple* is tiled across the drawing. *stipple* is ignored if a *pattern* is given. If no *fill-style* is given, or it is specified as **:solid**, when a *stipple* is given, then *fill-style* defaults to **:opaque-stippled**.

fill-style — Determines how the drawing is done. The value should be one of **:solid**, **:tiled, :opaque-stippled** or **:stippled**. The default value **:solid** means that the *foreground* is used everywhere. **:tiled** means that the *pattern* is repeated over across the drawing.

Additionally on X11/Motif **:opaque-stippled** means that the *stipple* bitmap is used with stipple 1s giving the *foreground* and 0s the *background*. **:stippled** means that the *stipple* bitmap is used with *foreground* where there are 1s and where the are 0s, no drawing is done. If you specify a stipple but no *fill-style,* or a *fill-style* of **:solid**, it defaults to **:opaque-stipple**.

pattern An image the same depth as the *port*, or **`nil`**. If non-nil, *pattern* is used as the source of color for drawing instead of the *foreground* and *background* parameters. A pattern is not transformed by the *transform* parameter. The *pattern* is tiled across the drawing. When *pattern* is specified, the *stipple* value is ignored.The default value of *pattern* is **`nil`**.

See **13.10 Working with images** for information on creating an image.

thickness A number (defaulting to 1) specifying the thickness of lines drawn. If *scale-thickness* is non-nil, the value *thickness* is in *port* (transformed) coordinates, otherwise *thickness* is in pixels.

scale-thickness A boolean, defaulting to **`t`** which means interpret the *thickness* parameter in transformed port coordinates. If *scale-thickness* is **`nil`**, *thickness* is interpreted in pixels.

dashed A boolean, defaulting to **`nil`**. If *dashed* is **`t`** then lines are drawn as a dashed line using *dash* as the mark-space specifier.

dash A list of two or more integer, or **`nil`**. A list of integers specifies the alternate mark and space sizes for dashed lines. These mark and space values are interpreted in pixels only. The default value of *dash* is **`(4 4)`**.

line-end-style The value should be one of **`:butt`**, **`:round`** or **`:projecting`** and specifies how to draw the ends of lines. The default value is **`:butt`**.

line-joint-style The value should be one of **`:bevel`**, **`:miter`** or **`:round`** and specifies how to draw the areas where the edges of polygons meet. The default value is **`:miter`**.

mask **`nil`**, or a list specifying a shape. The mask clips the drawing, so that drawing occurs only inside it.

mask should be **`nil`** (the default), a list of the form (*x y width height*), defining a rectangle inside which the drawing is done or a list of the form (**`:path`** *path* **`:fill-rule`** *fill-rule*) specifying a path inside which the drawing is done. The mask is not tiled.

In the latter case *path* should be a path specification (see **`draw-path`**). The *fill-rule* specifies how overlapping regions are filled. Possible values are **`:even-odd`** and **`:winding`**. The *mask* will be transformed by the *mask-transform* parameter.

There some examples of path masks in:

```
(example-edit-file "capi/graphics/paths")
```

mask-x An integer specifying in window coordinates where in the port the X coordinate of the mask origin is to be considered to be. The default value is 0.

The *mask-x* parameter works only when the *drawing-mode* is **`:compatible`** and the platform is GTK+ or X11/Motif.

mask-x is deprecated.

mask-y An integer specifying in window coordinates where in the port the Y coordinate of the mask origin is to be considered to be. The default value is 0.

The *mask-y* parameter works only when the *drawing-mode* is **`:compatible`** and the platform is GTK+ or X11/Motif.

mask-y is deprecated.

mask-transform	A **`transform`** object which determines the coordinate transformation use for the mask in *drawing-mode* **`:quality`**. *mask-transform* is used only in *drawing-mode* **`:quality`**. It is ignored in *drawing-mode* **`:compatible`**. The default value is the unit transform, which can also be specified as **`nil`**. Other allowed values include anything returned by the transform functions, described in **13.6 Graphics state transforms**. The other allowed value of *mask-transform* is the keyword **`:dynamic`** which is replaced by the current value of the *transform* graphics state parameter when the drawing operation uses the mask.
font	Either **`nil`** or a **`font`** object to be used by the **`draw-character`** and **`draw-string`** functions. The default value is **`nil`**. Note that *font* cannot be a **font-description**. Use **`find-best-font`** to convert a **font-description** to a **`font`**.
text-mode	A keyword controlling the mode of rendering text, most importantly anti-aliasing (see **below**).
shape-mode	A keyword controlling the mode of drawing shapes, that is, anything except text (see **below**).
compositing-mode	A keyword controlling the combining of new drawing with existing drawing (see **below**).

Each of *text-mode* and *shape-mode* can be one of:

`:plain`	No anti-aliasing.
`:antialias`	With anti-aliasing.
`:fastest`	Fastest rendering. The same as **`:plain`** except on Windows.
`:best`	Best display.
`:default`	The system default (which is **`:antialias`**).

Additionally *text-mode* can be **`:compatible`**, which causes text to be drawn the way it would be drawn if *drawing-mode* was **`:compatible`**. This makes a difference only on Microsoft Windows, because on other platforms the default *text-mode* draws like the **`:compatible`** one.

The default of both *text-mode* and *shape-mode* is **`:default`**.

compositing-mode is a keyword or an integer controlling the compositing mode, that is the way that a new drawing is combined with the existing value in the target of the drawing to generate the result.

Two values of *compositing-mode* are supported on all platforms other than Motif:

`:over`	Draw over the existing values. If the source is a solid color, then the result is simply the source. If the source has alpha value *alpha*, then it is blended with the destination, with the destination multiplied by the remainder of the alpha, that is (- 1 *alpha*).
`:copy`	The source is written to the destination ignoring the existing values. If the source has alpha and the target does not, that has the effect of converting semi-transparent source to solid.

The default value of *compositing-mode* is **`:over`**.

The value **`:copy`** of *compositing-mode* is especially useful for creating a transparent or semi-transparent **`pixmap-port`**, which can be displayed directly or converted to an image by **`make-image-from-port`**.

On Cocoa 10.5 and later and GTK+ 2.8 or later, these additional keyword values of *compositing-mode* are supported: **`:clear`**, **`:over`**, **`:in`**, **`:out`**, **`:atop`**, **`:dest-over`**, **`:dest-in`**, **`:dest-out`**, **`:dest-atop`**, **`:xor`** and **`:add`**. These correspond to the **CAIRO_OPERATOR_*** operators in Cairo, which are documented in **cairographics.org/operators** and the **CGBlendMode** values which are documented in the CGContext Reference at **developer.apple.com**.

Note: on GTK+, the "unbounded" operators (**:in**, **:out**, **:dest-in** and **:dest-atop**) do not work properly for shape drawings. They can only be used for image drawing and copying operations.

Both Cocoa and GTK+ also allow *compositing-mode* to be an integer, which is simply passed through to the underlying system. This allows using modes that are not available via keywords, but it is not portable. For Cocoa, it is a **CGBlendMode** as documented in the CGContext Reference. For GTK+ it is **cairo_operator_t**, as documented in the entry for **cairo_t** in the Gnome documentation for Cairo.

Note: For drawing images on Cocoa, only values that corresponding to available keywords work properly.

Notes

1. *operation* is not supported for drawing text on Microsoft Windows.
2. *stipple* is supported only on X11/Motif.
3. *mask-x* and *mask-y* are supported only on GTK+ and X11/Motif, and only when the *drawing-mode* is **:compatible**.
4. *pattern* is supported only on Microsoft Windows, GTK+ and X11/Motif.
5. *operation* is not supported by Cocoa/Core Graphics so this slot or argument is ignored on Cocoa.
6. *operation* is ignored when the port's *drawing-mode* is **:quality**.
7. *text-mode* and *shape-mode* are supported only on Cocoa, Cairo and GDI+, which are used on Macintosh, GTK and Windows respectively when the *drawing-mode* is **:quality**. For more information about *drawing-mode*, see **13.2.1 The drawing mode and anti-aliasing**.

Examples

```
(example-edit-file "capi/graphics/compositing-mode-simple")
```

```
(example-edit-file "capi/graphics/compositing-mode")
```

See also

make-graphics-state
set-graphics-state
with-graphics-state
13 Drawing - Graphics Ports

image *System Class*

Summary

An abstract image object.

Package

graphics-ports

Superclasses

t

Accessors

image-height
image-width

Description

The system class **image** is the abstract image object class. An image can be drawn using **draw-image**.

image-height and **image-width** return the image size in pixels.

Notes

On Cocoa and GTK+ you can drag and drop images. See **set-drop-object-supported-formats** for more information.

See also

convert-external-image
draw-image
load-image
make-image-from-port
make-sub-image
make-scaled-sub-image
read-and-convert-external-image
9 Adding Toolbars
13 Drawing - Graphics Ports
17 Drag and Drop

image-access-height
image-access-width *Functions*

Summary

Return the dimensions of the underlying image in an Image Access object.

Package

graphics-ports

Signatures

image-access-height *image-access* **=>** *height*

image-access-width *image-access* **=>** *width*

Arguments

image-access⇓ An Image Access object.

Values

height	An integer.
width	An integer.

Description

The functions **`image-access-height`** and **`image-access-width`** return the height and width of the underlying image in *image-access*.

image-access must be an Image Access object returned by **`make-image-access`**.

Notes

It is an error to call **`image-access-height`** or **`image-access-width`** on an Image Access object that has been freed by **`free-image-access`**.

Examples

```
(example-edit-file "capi/graphics/image-access")
```

```
(example-edit-file "capi/graphics/image-access-alpha")
```

See also

`free-image-access`
`make-image-access`

image-access-pixel *Accessor*

Summary

Gets and sets the pixels in an Image Access object.

Package

`graphics-ports`

Signature

`image-access-pixel` *image-access x y* **`=>`** *color-rep*

`(setf image-access-pixel)` *color-rep image-access x y* **`=>`** *color-rep*

Arguments

image-access⇓	An Image Access object.
x⇓	An integer.
y⇓	An integer.
color-rep⇓	A color reference.

Values

color-rep⇓ A color reference.

Description

The accessor **image-access-pixel** accesses the converted color at position *x*, *y* in the Image Access object *image-access*.

The converted color *color-rep* is a color representation like that returned by **convert-color**. If needed, *color-rep* can be converted to an RGB value using **unconvert-color**. *color-rep* can contain an alpha value, for images with an alpha channel, and in that case the values in *color-rep* are assumed to be premultiplied.

The function **(setf image-access-pixel)** sets the value of the pixel at position *x*, *y* in the Image Access object *image-access*.

The color rep has to be a converted color, and if the image has alpha it is assumed to be premultiplied.

image-access must be an Image Access object returned by **make-image-access**.

Notes

If the result of **image-access-pixel** on an image with alpha is used elsewhere (for example drawing a string with the same color), to get the same color you need to un-premultiply it first using **color-from-premultiplied**. When setting the color that came from elsewhere in an image with alpha, you will need to premultiply it using **color-to-premultiplied**. For images without alpha, premultiplication has no effect.

Examples

```
(example-edit-file "capi/graphics/image-access")
```

```
(example-edit-file "capi/graphics/image-access-alpha")
```

See also

color-from-premultiplied
color-to-premultiplied
image-access-pixels-from-bgra
image-access-pixels-to-bgra
image-access-transfer-to-image
image-access-transfer-from-image
free-image-access
make-image-access
13.10.8 Image access

image-access-pixels-from-bgra *Function*

Summary

Copies a vector of pixel values into an Image Access object.

Package

graphics-ports

Signature

image-access-pixels-from-bgra *image-access vector*

Arguments

image-access⇓ An Image Access object.

vector⇓ A vector.

Description

The function **image-access-pixels-from-bgra** copies all the pixels to the Image Access object *image-access* from the vector *vector*. *vector* should contain a sequence of integer values in the range 0-255 for blue, green, red and alpha of each pixel. This function is optimized for the case where *vector* has element type **(unsigned-byte 8)**. If the image has alpha, the values in *vector* are premultiplied.

An error is signalled if *vector* is not of the correct length for the Image Access object, that is **(* 4** *width height***)** where *width* and *height* represent the size of *image-access*.

image-access must be an Image Access object returned by **make-image-access**.

Notes

1. If you want to use the values in the vector that was filled from an image with alpha in other places, to get the sample color you will need to un-premultiply them, either by hand (divide the color values by the alpha), or by making a RGB color and using **color-from-premultiplied**.
2. **image-access-transfer-to-image** must be called after this function (similarly to **(setf image-access-pixel)**).

Examples

```
(example-edit-file "capi/graphics/image-access-bgra")
```

See also

color-from-premultiplied
image-access-pixel
image-access-pixels-to-bgra

image-access-pixels-to-bgra *Function*

Summary

Copies pixel values from an Image Access object into a vector.

Package

graphics-ports

Signature

image-access-pixels-to-bgra *image-access vector*

Arguments

image-access⇓ An Image Access object.

vector⇓ A vector.

Description

The function **`image-access-pixels-to-bgra`** copies all the pixels in the Image Access object *image-access* into the vector *vector* as a sequence of integer values in the range 0-255 for the blue, green, red and alpha components of each pixel. This function is optimized for the case where *vector* has element type **`(unsigned-byte 8)`**. If the image has alpha, the values in *vector* are assumed to be premultiplied.

An error is signalled if *vector* is not of the correct length for the Image Access object, that is **`(* 4`** *width height***`)`** where *width* and *height* represent the size of *image-access*.

image-access must be an Image Access object returned by **`make-image-access`**.

Notes

1. When setting values in a vector that is going to be used by **`image-access-pixels-to-bgra`** to modify an image with alpha using colors that came from elsewhere, you need to premultiply them either by hand (multiply the color values by the alpha), or using **`color-to-premultiplied`**.
2. **`image-access-transfer-from-image`** must be called before this function (similarly to **`image-access-pixel`**).

Examples

```
(example-edit-file "capi/graphics/image-access-bgra")
```

See also

`color-to-premultiplied`
`image-access-pixel`
`image-access-pixels-from-bgra`

image-access-transfer-from-image *Function*

Summary

Gets the pixel values from an **`image`**.

Package

`graphics-ports`

Signature

`image-access-transfer-from-image` *image-access*

Arguments

image-access⇓ An Image Access object.

Description

The function **`image-access-transfer-from-image`** gets the pixel values from an **`image`** object, making them accessible via a corresponding Image Access object *image-access*.

image-access must be an Image Access object returned by **`make-image-access`**.

Notionally **`image-access-transfer-from-image`** transfers the pixel data from the window system into *image-access*, though it might do nothing on platforms where the window system allows direct access to the pixel data.

You can read the pixel data with **`image-access-pixel`** and **`image-access-pixels-to-bgra`**.

You can write the pixel data with **`(setf image-access-pixel)`** and **`image-access-pixels-from-bgra`**.

Examples

```
(example-edit-file "capi/graphics/image-access")
```

See also

`image-access-transfer-to-image`
`image-access-pixel`
`image-access-pixels-from-bgra`
`image-access-pixels-to-bgra`
`free-image-access`
`make-image-access`
13.10.8 Image access

image-access-transfer-to-image *Function*

Summary

Sets the pixel values in an **`image`**.

Package

`graphics-ports`

Signature

`image-access-transfer-to-image` *image-access*

Arguments

image-access⇓ An Image Access object.

Description

The function **`image-access-transfer-to-image`** sets the pixel values in an **`image`** object from the values in a corresponding Image Access object *image-access*.

image-access must be an Image Access object returned by **`make-image-access`**.

Notionally **`image-access-transfer-to-image`** transfers the pixel data from *image-access* to the window system, though it might do nothing on platforms where the window system allows direct access to the pixel data.

Examples

```
(example-edit-file "capi/graphics/image-access")
```

See also

free-image-access
image-access-transfer-from-image
image-access-pixel
make-image-access
13.10.8 Image access

image-freed-p *Function*

Summary

Determines whether an image has been freed.

Package

graphics-ports

Signature

image-freed-p *image* **=>** *bool*

Arguments

image⇓ An image object.

Values

bool A boolean.

Description

The function **image-freed-p** returns non-nil if *image* has been freed, and **nil** otherwise.

image-loader *Function*

Summary

Returns the image load function.

Package

graphics-ports

Signature

`image-loader` *image-id* **`&key`** *image-translation-table* **`=>`** *loader*

Arguments

image-id⇓ An image identifier.

image-translation-table⇓

An image translation table.

Values

loader An image load function.

Description

The function **`image-loader`** returns the image load function that would be called to load the image associated with *image-id* in *image-translation-table*. If *image-id* is not registered with a load function, the default image load function is returned. The default value of *image-translation-table* is **`*default-image-translation-table*`**.

See also

`register-image-load-function`
`register-image-translation`

image-translation *Function*

Summary

Returns the translation for an image registered in its image translation table.

Package

`graphics-ports`

Signature

`image-translation` *image-id* **`&key`** *image-translation-table* **`=>`** *translation*

Arguments

image-id⇓ An image identifier.

image-translation-table⇓

An image translation table.

Values

translation A translation.

Description

The function **image-translation** returns the translation for *image-id* registered in *image-translation-table*. The default value of *image-translation-table* is ***default-image-translation-table***.

See also

register-image-load-function
register-image-translation

initialize-dithers *Function*

Summary

Initialize dither objects up to a given order.

Package

graphics-ports

Signature

initialize-dithers &optional *order*

Arguments

order⇓	An integer.

Description

The function **initialize-dithers** initializes dither objects up to the given *order* (size = 2 ^ *order*).

The default value of *order* is 3.

Notes

initialize-dither is deprecated. Dithers do not affect drawing or anti-aliasing.

See also

dither-color-spec
make-dither
with-dither

inset-rectangle *Function*

Summary

Moves the corners of a rectangle inwards by a given amount.

Package

`graphics-ports`

Signature

`inset-rectangle` *rectangle dx dy* **`&optional`** *dx-right dy-bottom*

Arguments

rectangle⇓	A list of integers.
dx⇓	An integer.
dy⇓	An integer.
dx-right⇓	An integer.
dy-bottom⇓	An integer.

Description

The function **`inset-rectangle`** moves the *left*, *top*, *right* and *bottom* elements of *rectangle* inwards towards the center by the distances *dx*, *dy*, *dx-right* and *dy-bottom* respectively.

By default, *dx-right* is *dx*, and *dy-bottom* is *dy*.

inside-rectangle *Function*

Summary

Determines if a point lies inside a rectangle.

Package

`graphics-ports`

Signature

`inside-rectangle` *rectangle x y* **`=>`** *result*

Arguments

rectangle⇓	A list of integers.
x⇓	An integer.
y⇓	An integer.

Values

result	A boolean.

Description

The function **`inside-rectangle`** returns **`t`** if the point (*x y*) is inside *rectangle*.

rectangle is expected to be ordered; if *rectangle* is specified by (*left top right bottom*), then *left* must be less than *right*, and *top* must be less than *bottom*. The lines **`y = bottom`** and **`x = right`** are not considered to be inside the rectangle.

invalidate-rectangle *Generic Function*

Summary

Invalidates the rectangle associated with the object, which causes it to be redisplayed.

Package

`graphics-ports`

Signature

`invalidate-rectangle` *object* **`&optional`** *x y width height* **`=>`** *result*

Arguments

object⇓	An instance of a subclass of **`graphics-port-mixin`** or a subclass of **`pinboard-object`**.
x⇓	A real number.
y⇓	A real number.
width⇓	A real number.
height⇓	A real number.

Values

result	A boolean.

Description

The generic function **`invalidate-rectangle`** invalidates the rectangle associated with *object*, which causes it to be redisplayed.

By default, **`invalidate-rectangle`** invalidates the whole rectangle, but this can be limited by supplying *x*, *y*, *width* and *height*.

The effect of invalidating an area is to cause the area to be redrawn. It has no effect on **`pixmap-port`**. When the pane has a supplied *display-callback*, this callback is called with an area containing the area specified by the argument to **`invalidate-rectangle`**. However, the call to *display-callback* is asynchronous, and the system coalesces areas from calls to **`invalidate-rectangle`** and actual expose events, so there is not a one-to-one relation between calls to **`invalidate-rectangle`** and invocations of *display-callback*.

In general, **`invalidate-rectangle`** should not be called inside the *display-callback*. If it is called, it must be conditional, otherwise this will cause repeated redisplay.

Notes

With *drawing-mode* **`:quality`**, drawings are done with anti-aliasing, which means that they affect pixels which are not obviously part of the drawing. For example, drawing a rectangle with $x = 10$ may affect the pixel at $x = 9$. This needs to be taken into account when computing the arguments to **`invalidate-rectangle`**.

For pinboard objects the recommended way of forcing redraw is **`redraw-pinboard-object`**, which takes anti-aliasing into account.

Examples

```
(example-edit-file "capi/graphics/plot-offline")
```

See also

`invalidate-rectangle-from-points`
`validate-rectangle`
13 Drawing - Graphics Ports

invalidate-rectangle-from-points *Function*

Summary

Invalidates a rectangle specified by two points, causing it to be redisplayed.

Package

`graphics-ports`

Signature

`invalidate-rectangle-from-points` *port x1 y1 x2 y2* **`&key`** *extend extend-x extend-y*

Arguments

port⇓ A graphics port.

x1⇓, *y1*⇓, *x2*⇓, *y2*⇓ Real numbers.

extend⇓, *extend-x*⇓, *extend-y*⇓
Real numbers.

Description

The function **`invalidate-rectangle-from-points`** invalidates a rectangle in *port* (by calling **`invalidate-rectangle`**) specified by two points. The coordinates of one point are (*x1*, *y1*) and the other (*x2*, *y2*) The points do not have to be ordered.

The keyword arguments specify extending the rectangle: *extend-x* extends the rectangle in the x dimension in both directions, and *extend-y* extends the rectangle in the y dimension in both directions. Both *extend-x* and *extend-y* default to *extend*, which itself defaults to 0 (that is, no extension).

`invalidate-rectangle-from-points` does not return a useful value.

See also

`invalidate-rectangle`

invert-transform *Function*

Summary

Constructs the inverse of a transform.

Package

graphics-ports

Signature

invert-transform *transform* **&optional** *into* **=>** *inverse*

Arguments

transform⇓	A **transform** object.
into⇓	A **transform** object or **nil**.

Values

inverse	A **transform** object.

Description

The function **invert-transform** constructs the inverse of *transform*. If *T* is *transform* and *T'* is its inverse, then *TT'* = I. If *into* is non-nil it is modified to contain *T'* and returned, otherwise a new transform is constructed and returned.

Notes

See **graphics-state** for details of how a **transform** is used.

See also

graphics-state
transform

list-all-font-names *Function*

Summary

Finds the names of the available fonts.

Package

graphics-ports

Signature

list-all-font-names *pane* **=>** *fdescs*

Arguments

pane⇓ A graphics port.

Values

fdescs A list of font description objects.

Description

The function **list-all-font-names** returns a list of partially-specified font description objects which contain the "name" attributes for each known font that is available for *pane*.

On Microsoft Windows and Cocoa the "name" attributes are just the **:family** attribute.

On X11 the "name" attributes are **:foundry** and **:family**.

See also

font-description-attributes
find-matching-fonts
13 Drawing - Graphics Ports

list-known-image-formats *Function*

Summary

Returns the known image formats.

Package

graphics-ports

Signature

list-known-image-formats *screen-spec* **&optional** *for-writing-too* **=>** *formats*

Arguments

screen-spec⇓ A CAPI object, a plist, or **nil**.

for-writing-too⇓ A generalized boolean.

Values

formats⇓ A list of keywords.

Description

The function **list-known-image-formats** returns a list of keywords which specify known image formats.

screen-spec is an object that **convert-to-screen** can recognize, typically a pane or simply **nil**.

If *for-writing-too* is not supplied or is **nil**, then *formats* is a list of formats that can be loaded. All the formats in the list can be loaded, but on Cocoa and Windows the list is not exhaustive, and it may be possible to load formats that are not listed.

If *for-writing-too* is supplied as non-nil, then formats is a list of types that **`externalize-and-write-image`** can write. In this case the list is exhaustive on all platforms, and **`externalize-and-write-image`** can write a format if and only if it appears in the list.

All platforms (except Motif) can read and write **`:bmp`**, **`:jpg`**, **`:png`** and **`:tiff`** images, and also recognize **`:jpeg`** as an alias for **`:jpg`**, so the list will always include all of these keywords.

See also

`convert-to-screen`
`externalize-and-write-image`
13 Drawing - Graphics Ports

load-icon-image *Function*

Summary

Loads a Windows icon image, and returns the image object.

Package

`graphics-ports`

Signature

`load-icon-image` *port id* **`&key`** *width height* **`=>`** *image*

Arguments

port⇓	A graphics port or CAPI object.
id⇓	A keyword, string or pathname.
width⇓	The desired width in pixels, or **`nil`**.
height⇓	The desired height in pixels, or **`nil`**.

Values

image	An **`image`** object.

Description

The function **`load-icon-image`** loads an icon specified by *id* which should be either a keyword describing a standard icon, or a string or a pathname naming a Windows format icon (**`.ico`**) file.

The following keyword values of *id* are recognized:

`:sample`	A rectangle.
`:hand`	A cross in a circle.
`:ques`	A question mark in a bubble.

:bang	An exclamation mark in a triangle.
:note	An 'I' in a bubble.
:winlogo	The Windows logo.
:warning	Same as **:bang**.
:error	Same as **:hand**.
:information	Same as **:note**.

load-icon-image returns an **image** object which can be drawn to *port* using **draw-image** and which must be freed using **free-image** when no longer needed.

When *id* specifies a file and *width* and *height* are specified, then the most appropriate image is chosen from the icon file and is scaled accordingly. If *width* and *height* are **nil** the first image in the file is used at its natural size. *width* defaults to **nil** and *height* defaults to *width*.

Note: load-icon-image is defined only in LispWorks for Windows.

See also

draw-image
free-image
load-image
13 Drawing - Graphics Ports

load-image *Function*

Summary

Loads an image and returns the image object.

Package

graphics-ports

Signature

load-image *gp id* **&key** *cache type editable image-translation-table* **=>** *image*

Arguments

gp⇓	A graphics port.
id⇓	An image identifier, a file, an **external-image**, or an **image**.
cache⇓	A boolean.
type⇓	A keyword, or **nil**.
editable⇓	One of the keywords **:with-alpha** and **:without-alpha**, or a boolean.
image-translation-table⇓	An image translation table.

Values

image⇓ An image object.

Description

The function **`load-image`** loads an image identified by *id* via *image-translation-table* using the image load function registered with it. It returns an **`image`** object with the representation slot initialized. *gp* specifies a graphics port used to identify the library. It also specifies the resource in which colors are defined and if necessary allocated for the image. If *id* is in the table but the translation is not an external image, and the image loader returns an external image as the second value, that external image replaces the translation in the table. The default value of *image-translation-table* is **`*default-image-translation-table*`**.

id can be an **`image`**, which is just associated with the port *gp* and returned if it is a Plain Image or if *editable* is **`nil`**. Otherwise a new Plain Image object is returned, as described below.

id can also be a string or pathname denoting a file, and in this case the image is loaded according to *type*, as described below.

cache controls whether the image translation is cached. See the **`convert-external-image`** function for more details.

type tells **`load-image`** that the image is in a particular graphics format. Currently the only recognized value is **`:bmp`**, which means the image is a Bitmap. Other values of *type* cause **`load-image`** to load the image according to the file type of *id*, if *id* denotes a file, as described for **`read-external-image`**. See **13 Drawing - Graphics Ports** for a discussion of image handling. The default value of *type* is **`nil`**.

editable controls whether the image *image* is a Plain Image suitable for use with the Image Access API. The values of *editable* have the following effects:

`nil`	The image is not editable.
`:without-alpha`	The image is editable, but does not have an alpha channel.
`t`	The image is editable, but does not have an alpha channel if the source of the image has an alpha channel (for example, a TIFF file with alpha channel).
`:with-alpha`	The image is editable and has an alpha channel. It will be fully opaque when loading files without an alpha channel.

Given an **`image`** *my-image*, call:

```
(load-image port my-image :editable t)
```

to create an **`image`** guaranteed to work with **`make-image-access`**. The default value of *editable* is **`nil`**.

Normally the image is freed automatically, when *gp* is destroyed. However there are circumstances where you need to explicitly free an image, for example when you want it to go away before the port. If the image is not freed, a memory leak occurs.

Note: *gp* must already be created at the time **`load-image`** is called. If you need to delay loading the image, for example if you are computing the image dynamically, then you can call **`load-image`** in the *create-callback* of the port or even in its first *display-callback*.

Compatibility note

In LispWorks 4.4 there is a keyword argument **`:force-plain`** with the same effect as **`:editable`**. **`:force-plain`** is still accepted in LispWorks 8.0 for backwards compatibility, but you should now use **`:editable`** instead.

See also

convert-external-image
default-image-translation-table
load-icon-image
make-image
make-image-access
13 Drawing - Graphics Ports

make-dither *Function*

Summary

Makes a dither matrix of a given size.

Package

graphics-ports

Signature

make-dither *size* **=>** *matrix*

Arguments

size⇓ An integer.

Values

matrix A dither matrix.

Description

The function **make-dither** makes a dither matrix of the given *size*.

Notes

make-dither is deprecated. Dithers do not affect drawing or anti-aliasing.

See also

dither-color-spec
initialize-dithers
with-dither

make-font-description *Function*

Summary

Returns a new font description object containing given font attributes.

Package

graphics-ports

Signature

make-font-description &rest *font-attribute** **=>** *fdesc*

Arguments

*font-attribute**⇓	Keywords and values to initialize a font description.

Values

fdesc⇓	A font description object.

Description

The function **make-font-description** returns a new font description object containing the given *font-attribute** keywords and values. There is no error checking of the attributes at this point.

The attribute **:stock** is handled specially: it is omitted from *fdesc*, unless it is the only attribute specified.

See also

augment-font-description
convert-to-font-description
find-best-font
find-matching-fonts
font-description
merge-font-descriptions

make-graphics-state *Function*

Summary

Creates a **graphics-state** object.

Package

graphics-ports

Signature

make-graphics-state &key *transform foreground background operation thickness scale-thickness dashed dash line-end-style line-joint-style mask fill-style stipple pattern mask-x mask-y font text-mode shape-mode compositing-mode mask-transform* **=>** *state*

Arguments

transform⇓, *foreground*⇓, *background*⇓, *operation*⇓, *thickness*⇓, *scale-thickness*⇓, *dashed*⇓, *dash*⇓, *line-end-style*⇓, *line-joint-style*⇓, *mask*⇓, *fill-style*⇓, *stipple*⇓, *pattern*⇓, *mask-x*⇓, *mask-y*⇓, *font*⇓, *text-mode*⇓, *shape-mode*⇓, *compositing-mode*⇓, *mask-transform*⇓

See **graphics-state** for interpretation of the arguments.

Values

state A **graphics-state** object.

Description

The function **make-graphics-state** creates a **graphics-state** object using *transform*, *foreground*, *background*, *operation*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-end-style*, *line-joint-style*, *mask*, *fill-style*, *stipple*, *pattern*, *mask-x*, *mask-y*, *font*, *text-mode*, *shape-mode*, *compositing-mode* and *mask-transform*. Each graphics port has a graphics state associated with it, but you may want to create your own individual graphics states for use in specialized drawing operations. Graphics state objects do not consume local resources beyond dynamic memory for the structure (so you can be relaxed about creating them in some number if you really need to).

See also

graphics-state
set-graphics-state

make-image *Function*

Summary

Makes a new, empty, **image** object.

Package

graphics-ports

Signature

make-image *port width height* **&key** *alpha* **=>** *image*

Arguments

port⇓ A graphics port.

width⇓ A positive integer.

height⇓ A positive integer.

alpha⇓ A generalized boolean.

Values

image⇓ An **image** object.

Description

The function **make-image** makes a new blank, editable **image** object associated with *port* and of the given *width* and *height*.

On Windows and Cocoa, if *alpha* is true, then the image will have an alpha channel.

The initial pixels in *image* are undefined. *image* is editable, that is, it is suitable for use with the Image Access API. To set the pixels, see **make-image-access**.

See also

load-image
make-image-access

make-image-access *Function*

Summary

Creates an Image Access object.

Package

graphics-ports

Signature

make-image-access *port image* **=>** *image-access*

Arguments

port⇓	A graphics port.
image⇓	An **image** object.

Values

image-access⇓	An Image Access object.

Description

The function **make-image-access** returns an Image Access object for the given **image** image on *port*.

image can be any **image** object returned by **make-image-from-port**. An **image** object returned by **load-image** is also suitable, but only if it is a Plain Image (see below).

image-access is used when reading and writing the pixel values of the image. For an overview of using Image Access objects, see **13.10.8 Image access**.

Notes

1. On some platforms (currently Windows) not every **image** object is a Plain Image. If needed, forcibly create a Plain Image suitable for passing to **make-image-access** as described in **load-image**.
2. Ensure that you eventually discard *image-access*, using **free-image-access**.

Examples

```
(example-edit-file "capi/graphics/image-access")
```

See also

free-image-access
image-access-transfer-from-image
image-access-transfer-to-image
image-access-height
image-access-pixel
load-image
make-image
13.10.8 Image access

make-image-from-port *Function*

Summary

Makes an image out of a specified rectangle of a graphics port's contents.

Package

graphics-ports

Signature

make-image-from-port *port* **&optional** *x y width height* **=>** *image*

Arguments

port⇓	A graphics port.
x⇓	An integer.
y⇓	An integer.
width⇓	An integer.
height⇓	An integer.

Values

image	An image.

Description

The function **make-image-from-port** makes an **image** out of the specified rectangle of the port's contents. The default is the whole port, but a region can be specified by supplying *x*, *y*, *width*, and *height*. The default values of *x* and *y* is 0.

Normally the image is freed automatically, when *port* is destroyed. However there are circumstances where you need to explicitly free an image, for example when you want it to go away before the port. If the image is not freed, a memory leak occurs.

See also

externalize-image
13 Drawing - Graphics Ports

make-scaled-sub-image *Function*

Summary

Makes a new image from a scaled part of an image.

Package

graphics-ports

Signature

make-scaled-sub-image *port image to-width to-height* **&key** *from-x from-y from-width from-height* **=>** *sub-image*

Arguments

port⇓	A graphics port.
image⇓	An **image**.
to-width⇓	An integer.
to-height⇓	An integer.
from-x⇓	An integer.
from-y⇓	An integer.
from-width⇓	An integer.
from-height⇓	An integer.

Values

sub-image⇓	An image.

Description

The function **make-scaled-sub-image** makes a new **image** from the scaled rectangular region of *image* specified by *from-x*, *from-y*, *from-width* and *from-height*. The returned *sub-image* is associated with *port* and has size specified by *to-width* and *to-height*.

The default values of *from-x* and *from-y* are 0.

The default value of *from-width* is the width of *image*.

The default value of *from-height* is the height of *image*.

When *from-width* equals *to-width* and *from-height* equals *to-height*, then this function is equivalent to **make-sub-image**.

See also

image
make-sub-image
13 Drawing - Graphics Ports
17 Drag and Drop

make-sub-image *Function*

Summary

Makes a new image from part of an image.

Package

`graphics-ports`

Signature

`make-sub-image` *port image* **`&optional`** *x y width height* **`=>`** *sub-image*

Arguments

port⇓	A graphics port.
image⇓	An **`image`**.
x⇓	An integer.
y⇓	An integer.
width⇓	An integer.
height⇓	An integer.

Values

sub-image	An **`image`**.

Description

The function **`make-sub-image`** makes a new **`image`** object from the rectangular region of the supplied *image* specified by *x*, *y*, *width* and *height*, for use with *port*.

The default values of *x* and *y* are 0.

The default value of *width* is the *width* of *image*.

The default value of *height* is the *height* of *image*.

See also

`image`
`make-scaled-sub-image`
13 Drawing - Graphics Ports
17 Drag and Drop

make-transform *Function*

Summary

Returns a new **transform** object initialized according to a set of optional arguments.

Package

graphics-ports

Signature

make-transform &optional *a b c d e f* **=>** *transform*

Arguments

a⇓, *b*⇓, *c*⇓, *d*⇓, *e*⇓, *f*⇓

Real numbers.

Values

transform A **transform** object.

Description

The function **make-transform** returns a new transform object initialized according to the optional args. The default args make the unit transform.

Default values are as follows: *a* and *d* are 1; *b*, *c*, *e*, and *f* are 0. The transform matrix is:

```
a  b  0
c  d  0
e  f  1
```

for generalized two dimensional points of the form **(x y 1)**.

Notes

See **graphics-state** for details of how a **transform** is used.

Examples

This transform will cause rotation by pi/4 radians:

```
(let ((s (sin (/ pi 4)))
      (c (cos (/ pi 4))))
  (gp:make-transform  c  s (- s) c 0 0))
```

See also

graphics-state
transform

merge-font-descriptions *Function*

Summary

Returns a font description containing the attributes of two specified font descriptions.

Package

`graphics-ports`

Signature

`merge-font-descriptions` *fdesc1 fdesc2* **`=>`** *fdesc*

Arguments

fdesc1⇓ A font description.

fdesc2⇓ A font description.

Values

fdesc⇓ A font description.

Description

The function **`merge-font-descriptions`** returns a font description containing all the attributes of *fdesc1* and *fdesc2*. If an attribute appears in both *fdesc1* and *fdesc2*, the value in *fdesc1* is used. The attribute **`:stock`** is handled specially: it is omitted from *fdesc*, unless it is the only attribute in *fdesc1* and *fdesc2*.

The contents of *fdesc1* and *fdesc2* are not modified.

See also

`make-font-description`
13 Drawing - Graphics Ports

offset-rectangle *Function*

Summary

Offsets a rectangle by a given distance.

Package

`graphics-ports`

Signature

`offset-rectangle` *rectangle dx dy*

Arguments

rectangle⇓	A list of integers.
dx⇓	A real number.
dy⇓	A real number.

Description

The function **`offset-rectangle`** offsets *rectangle* by the distance (*dx dy*).

rectangle is a list (*left top right bottom*).

ordered-rectangle-union *Function*

Summary

Returns the union of two rectangles.

Package

`graphics-ports`

Signature

`ordered-rectangle-union` *left-1 top-1 right-1 bottom-1 left-2 top-2 right-2 bottom-2* **`=>`** *left*, *top*, *right*, *bottom*

Arguments

left-1⇓, *top-1*⇓, *right-1*⇓, *bottom-1*⇓

Real numbers.

left-2⇓, *top-2*⇓, *right-2*⇓, *bottom-2*⇓

Real numbers.

Values

left, *top*, *right*, *bottom*

Real numbers.

Description

The function **`ordered-rectangle-union`** returns four values: the *left*, *top*, *right* and *bottom* of the union of the two rectangles specified by (*left-1 top-1 right-1 bottom-1*) and (*left-2 top-2 right-2 bottom-2*). The caller guarantees that each input rectangle is ordered, that is, the left values must be smaller or equal to the right values, and the top values must be greater than or equal to the bottom ones.

See also

`rectangle-union`

pi-by-2 *Constant*

Summary

(/ pi 2) as a **double-float**.

Package

graphics-ports

Description

The constant **pi-by-2** is the result of **(/ cl:pi 2)**. It is a **cl:double-float**.

See also

2pi
fpi

pixblt *Function*

Summary

Copies one area of a graphics port to another area of a different graphics port (deprecated).

Package

graphics-ports

Signature

pixblt *to-port operation from-port to-x to-y width height from-x from-y*

Arguments

to-port⇓	A graphics port.
operation⇓	A graphics state operation.
from-port⇓	A graphics port.
to-x⇓	A real number.
to-y⇓	A real number.
width⇓	A real number.
height⇓	A real number.
from-x⇓	A real number.
from-y⇓	A real number.

Description

The function **pixblt** copies one area of *from-port* to another area of *to-port* using the specified *operation* and *mask*. Both ports should be the same depth.

The corners of the copied rectangle are (*from-x from-y*), (*from-x+width from-y*), (*from-x+width from-y+height*) and (*from-x from-y+height*), which are interpreted as pixel positions in the window coordinates of *from-port*. The top left of the rectangle is copied to (*to-x to-y*) in *to-port*'s coordinates. The graphics port transforms are not used.

operation is ignored when the *drawing-mode* is **:quality** (the default). See **13.7.1 Combining pixels with :compatible drawing** for valid values for *operation*.

pixblt is deprecated, because the **:quality** *drawing-mode* does not support *operation*, and because it ignores the transformations, which means it does not always work as expected. In particular, it can draw at the wrong place inside the *display-callback* of **output-pane**.

pixblt is deprecated -- use **copy-area** instead, which does take account of the transform. See also **graphics-state** parameter *compositing-mode* for a way to control how **copy-area** blends the source and the target.

See also

copy-area
graphics-state
13 Drawing - Graphics Ports

pixmap-port *Class*

Summary

The class of pixmap graphics port objects.

Package

graphics-ports

Superclasses

graphics-port-mixin

Description

The class **pixmap-port** is the class of pixmap graphics port objects which can be used for drawing operations.

See also

create-pixmap-port
destroy-pixmap-port
with-pixmap-graphics-port

port-drawing-mode-quality-p *Generic Function*

Summary

Tests whether a port does quality drawing.

Package

`graphics-ports`

Signature

`port-drawing-mode-quality-p` *port* **`=>`** *result*

Arguments

port⇓	A graphics port.

Values

result	A boolean.

Description

The generic function **`port-drawing-mode-quality-p`** returns true if the graphics port *port* does quality drawing.

A port does quality drawing if both:

1. It was not made with *drawing-mode* **`:compatible`**.
2. The underlying library supports quality drawing.

Microsoft Windows and Cocoa always support quality drawing, GTK+ supports it from version 2.8 and greater, but Motif never supports it.

Examples

```
(example-edit-file "capi/graphics/images-with-alpha")
```

See also

13.2.1 The drawing mode and anti-aliasing.

port-graphics-state *Function*

Summary

Returns the **`graphics-state`** object for a graphics port.

Package

`graphics-ports`

Signature

`port-graphics-state` *port* **`=>`** *state*

Arguments

port⇓	A graphics port.

Values

state	A **`graphics-state`** object.

Description

The function **`port-graphics-state`** returns the **`graphics-state`** object for *port*. The individual slots can be accessed using the accessor functions documented for **`graphics-state`**.

See also

`graphics-state`

port-height *Function*

Summary

Returns the pixel height of a port.

Package

`graphics-ports`

Signature

`port-height` *port* **`=>`** *result*

Arguments

port⇓	A graphics port.

Values

result	An integer.

Description

The function **`port-height`** returns the pixel height of *port*.

port-owner *Function*

Summary

Returns the port owner of a graphics port.

Package

graphics-ports

Signature

port-owner *graphics-port* **=>** *owner*

Arguments

graphics-port⇓ A graphics port.

Values

owner A graphics port.

Description

The function **port-owner** returns the port owner of the graphics port *graphics-port*.

For **output-pane** the owner is always the pane itself.

For **pixmap-port** it is the owner of the port that was used when it was made.

For **metafile-port** the owner can be specified by the keyword argument **:owner** in the macros **with-internal-metafile** and **with-external-metafile**, otherwise it is the port itself.

For **printer-port** the owner can be specified by the keyword argument **:owner** in **with-print-job**, otherwise it is the port itself.

port-string-height *Function*

Summary

Returns the height of a string drawn to a given port in pixels.

Package

graphics-ports

Signature

port-string-height *port string* **=>** *height*

Arguments

port⇓	A graphics port.
string⇓	A string.

Values

height	An integer.

Description

The function **`port-string-height`** returns the height in pixels of *string* when drawn to *port*. The font used is the *font* currently in the port's **`graphics-state`**.

port-string-width *Function*

Summary

Returns the width of a string drawn to a given port in pixels.

Package

`graphics-ports`

Signature

`port-string-width` *port string* **`=>`** *width*

Arguments

port⇓	A graphics port.
string⇓	A string.

Values

width	An integer.

Description

The function **`port-string-width`** returns the width in pixels of *string* when drawn to *port*. The font used is the *font* currently in the port's **`graphics-state`**.

Notes

To compute the horizontal extents of each successive character in a string for a given port or font, use **`compute-char-extents`**.

See also

`compute-char-extents`

port-width *Function*

Summary

Returns the pixel width of a port.

Package

graphics-ports

Signature

port-width *port* **=>** *width*

Arguments

port⇓	A graphics port.

Values

width	An integer.

Description

The function **port-width** returns the pixel width of *port*.

postmultiply-transforms *Function*

Summary

Postmultiplies two transforms.

Package

graphics-ports

Signature

postmultiply-transforms *transform1* *transform2*

Arguments

transform1⇓	A **transform** object.
transform2⇓	A **transform** object.

Description

The function **postmultiply-transforms** postmultiplies the partial 3 x 3 matrix represented by *transform1* by the partial 3 x 3 matrix represented by *transform2*, storing the result in *transform1*. In the result, the translation, scaling and rotation operations contained in *transform2* are effectively performed *after* those in *transform1*.

```
transform1 = transform1 . transform2
```

premultiply-transforms *Function*

Summary

Premultiplies two transforms.

Package

graphics-ports

Signature

premultiply-transforms *transform1 transform2*

Arguments

transform1⇓	A **transform** object.
transform2⇓	A **transform** object.

Description

The function **premultiply-transforms** premultiplies the partial 3 x 3 matrix represented by *transform1* by the partial 3 x 3 matrix represented by *transform2*, storing the result in *transform1*. In the result, the translation, scaling and rotation operations contained in *transform2* are effectively performed *before* those in *transform1*.

```
transform1 = transform2 . transform1
```

read-and-convert-external-image *Function*

Summary

Returns an image converted from an external image read from a file.

Package

graphics-ports

Signature

read-and-convert-external-image *gp file* **&key** *transparent-color-index* **=>** *image***,** *external-image*

Arguments

gp⇓	A CAPI pane.
file⇓	A pathname designator.

transparent-color-index⇓
An integer or **`nil`**.

Values

image An **image**.

external-image An **external-image**.

Description

The function **`read-and-convert-external-image`** returns an **image** converted for use with *gp* from an external image read from *file*. The external image is returned as a second value.

transparent-color-index is interpreted as described for **`read-external-image`**.

See also

`convert-external-image`
`external-image`
`read-external-image`
13 Drawing - Graphics Ports

read-external-image *Function*

Summary

Returns an external image read from a file.

Package

`graphics-ports`

Signature

`read-external-image` *file* **`&key`** *transparent-color-index type* **`=>`** *image*

Arguments

file⇓ A pathname designator.

transparent-color-index⇓
An integer, a cons or **`nil`**.

type⇓ A keyword, or **`nil`**.

Values

image An external image.

Description

The function **`read-external-image`** returns an external image read from *file*.

If *transparent-color-index* is an integer it specifies the index of the transparent color in the color map.

transparent-color-index can also be a cons **(***index* **.** *new-color***)** where *new-color* is a color specification that is converted to the color to use instead of the color at index *index* in the color map. *new-color* can also be the keyword **:transparent**. On most platforms this makes it truly transparent. On Motif it uses the background of the pane that it is associated with by **load-image**.

transparent-color-index works only for images with a color map, that is, those with 256 colors or less. The default value is **nil**, meaning that there is no transparent color.

type tells **read-external-image** that the image is in a particular graphics format. Currently the only recognized value is **:bmp**, which means the image is read as a Bitmap. Other values of *type* cause **read-external-image** to read the image according to the file type of *file*. **"bmp"** or **"dib"** mean that the image is read as a Bitmap. Other file types are handled in Operating System-specific ways. See **13.10 Working with images** for details. The default value of *type* is **nil**.

Examples

To see the effect of *transparent-color-index,* do:

1. **(example-edit-file "capi/graphics/images")**
2. Specify a non-white **:background** for the *viewer* pane. Use an image editing tool to find the transparent color index (183 in this image) and change the call to **read-external-image** like this:

```
(gp:read-external-image file
   :transparent-color-index 183)
```

3. Then compile and run the example, click the **Change...** button and select the **Setup.bmp** file.

See also

external-image

rectangle-bind *Macro*

Summary

Binds four variables to the corners of a rectangle across a body of code.

Package

graphics-ports

Signature

rectangle-bind ((*a b c d***)** *rectangle***) &body** *body* **=>** *result*

Arguments

a⇓	A variable.
b⇓	A variable.
c⇓	A variable.
d⇓	A variable.

rectangle⇓	A rectangle.
body⇓	A body of code.

Values

result	The return value of the last form in *body*.

Description

The macro **`rectangle-bind`** binds the variables *a b c d* to *left top right bottom* of *rectangle* and evaluates the forms in *body* as an implicit **`progn`**.

rectangle-bottom *Macro*

Summary

Get and sets the *bottom* element of a rectangle.

Package

`graphics-ports`

Signature

`rectangle-bottom` *rectangle* **`=>`** *bottom*

Arguments

rectangle⇓	A rectangle.

Values

bottom⇓	A real number.

Description

The macro **`rectangle-bottom`** returns the *bottom* element of *rectangle*. **`rectangle-bottom`** can also be used with **`setf`** to set the *bottom* element of *rectangle*.

rectangle is a list of numbers (*left top right bottom*).

rectangle-height *Macro*

Summary

Returns the height of a rectangle.

Package

`graphics-ports`

Signature

`rectangle-height` *rectangle* **`=>`** *height*

Arguments

rectangle⇓ A rectangle.

Values

height A real number.

Description

The macro **`rectangle-height`** returns the difference between the *bottom* and *top* elements of *rectangle*.

rectangle is a list of numbers (*left top right bottom*).

rectangle-left *Macro*

Summary

Gets and set the *left* element of a rectangle.

Package

`graphics-ports`

Signature

`rectangle-left` *rectangle* **`=>`** *left*

Arguments

rectangle⇓ A rectangle.

Values

left⇓ A real number.

Description

The macro **`rectangle-left`** returns and via **`setf`** sets the *left* element of *rectangle*. **`rectangle-left`** can also be used with **`setf`** to set the *left* element of *rectangle*.

rectangle is a list of numbers (*left top right bottom*).

rectangle-right *Macro*

Summary

Gets and sets the *right* element of a rectangle.

Package

`graphics-ports`

Signature

`rectangle-right` *rectangle* **`=>`** *right*

Arguments

rectangle⇓	A rectangle.

Values

right⇓	A real number.

Description

The macro **`rectangle-right`** returns and via **`setf`** sets the *right* element of *rectangle*. **`rectangle-right`** can also be used with **`setf`** to set the *right* element of *rectangle*.

rectangle is a list of numbers (*left top right bottom*).

rectangle-top *Macro*

Summary

Gets and sets the *top* element of a rectangle.

Package

`graphics-ports`

Signature

`rectangle-top` *rectangle* **`=>`** *top*

Arguments

rectangle⇓	A rectangle.

Values

top⇓	A real number.

Description

The macro **rectangle-top** returns and via **setf** sets the *top* element of *rectangle*. **rectangle-top** can also be used with **setf** to set the *top* element of *rectangle*.

rectangle is a list of numbers (*left top right bottom*).

rectangle-union *Function*

Summary

Returns the four values representing a union of two rectangles.

Package

graphics-ports

Signature

rectangle-union *left-1 top-1 right-1 bottom-1 left-2 top-2 right-2 bottom-2* **=>** *left*, *top*, *right*, *bottom*

Arguments

left-1⇓	A real number.
top-1⇓	A real number.
right-1⇓	A real number.
bottom-1⇓	A real number.
left-2⇓	A real number.
top-2⇓	A real number.
right-2⇓	A real number.
bottom-2⇓	A real number.

Values

left	A real number.
top	A real number.
right	A real number.
bottom	A real number.

Description

The function **rectangle-union** returns four values: the *left*, *top*, *right* and *bottom* of the union of the two rectangles specified by (*left-1 top-1 right-1 bottom-1*) and (*left-2 top-2 right-2 bottom-2*). The values input for the two rectangles are ordered by this function before it uses them.

See also

ordered-rectangle-union

rectangle-width *Macro*

Summary

Returns the difference between the *left* and *right* elements of a rectangle.

Package

`graphics-ports`

Signature

`rectangle-width` *rectangle* **`=>`** *width*

Arguments

rectangle⇓	A rectangle.

Values

width	A real number.

Description

The macro **`rectangle-width`** returns the difference between *right* and *left* elements of *rectangle*.

rectangle is a list of numbers (*left top right bottom*).

rect-bind *Macro*

Summary

Binds four variables to the elements of a rectangle across a body of code.

Package

`graphics-ports`

Signature

`rect-bind ((`*x y width height***`)`** *rectangle***`) &body`** *body* **`=>`** *result*

Arguments

x⇓	A variable.
y⇓	A variable.
width⇓	A variable.
height⇓	A variable.

rectangle⇓ A rectangle.

body⇓ A body of Lisp code.

Values

result The return value of the last form in *body*.

Description

The macro **`rect-bind`** binds *x*, *y*, *width* and *height* to the appropriate values from *rectangle* and evaluates the forms in *body* as an implicit **`progn`**. *rectangle* is a list of the form (*left top right bottom*).

register-image-load-function *Function*

Summary

Registers one or more image identifiers with an image loading function.

Package

`graphics-ports`

Signature

`register-image-load-function` *image-id image-load-function* **`&key`** *image-translation-table*

Arguments

image-id⇓ An image identifier or a list of image identifiers.

image-load-function⇓ A function.

image-translation-table⇓

An image translation table.

Description

The function **`register-image-load-function`** registers one or more *image-ids* with an *image-load-function* in *image-translation-table*. If *image-load-function* is **`nil`** it causes the default loader to be used in subsequent calls to **`load-image`**. *image-id* can be a list of identifiers or a single identifier. The default value of *image-translation-table* is **`*default-image-translation-table*`**.

See also

`*default-image-translation-table*`
`load-image`

register-image-translation *Function*

Summary

Registers an image identifier and image loading function with a translation in an image translation table.

Package

`graphics-ports`

Signature

`register-image-translation` *image-id translation* **`&key`** *image-translation-table image-load-fn* **`=>`** *image-id*, *image-load-fn*

Arguments

image-id⇓ An image identifier.

translation⇓ An image translation.

image-translation-table⇓ An image translation table.

image-load-fn⇓ An image loading function.

Values

image-id An image identifier.

image-load-fn An image loading function.

Description

The function **`register-image-translation`** registers *image-id* and *image-load-fn* with *translation* in *image-translation-table*. When **`load-image`** is called with second argument *image-id*, then *image-load-fn* is called with *translation* as its second argument.

If *image-load-fn* is **`nil`**, the default image loader in *image-translation-table* is used; this converts an external image object or file to an image.

If *translation* is **`nil`** then *image-id* is deregistered.

The default value of *image-translation-table* is **`*default-image-translation-table*`**.

See also

`*default-image-translation-table*`
`load-image`
`reset-image-translation-table`
13 Drawing - Graphics Ports

reset-image-translation-table *Function*

Summary

Clears the image translation table hash tables.

Package

graphics-ports

Signature

reset-image-translation-table &key *image-translation-table*

Arguments

image-translation-table⇓
An image translation table.

Description

The function **reset-image-translation-table** clears the image translation table hash tables and set the default *image-load-fn* to **read-and-convert-external-image**. The default value of *image-translation-table* is ***default-image-translation-table***.

See also

default-image-translation-table
read-and-convert-external-image
register-image-translation

separation *Function*

Summary

Returns the distance between two points.

Package

graphics-ports

Signature

separation *x1 y1 x2 y2* **=>** *dist*

Arguments

x1⇓ An integer.

y1⇓ An integer.

x2⇓ An integer.

y2⇓ An integer.

Values

dist A real number.

Description

The function **`separation`** returns the distance between points (*x1 y1*) and (*x2 y2*).

set-default-image-load-function *Function*

Summary

Sets the default image load function of an image translation table.

Package

`graphics-ports`

Signature

`set-default-image-load-function` *image-load-function* **`&key`** *image-translation-table*

Arguments

image-load-function⇓ An image load function.

image-translation-table⇓

An image translation function.

Description

The function **`set-default-image-load-function`** sets the default image load function of *image-translation-table* to *image-load-function*. The initial default image load function is **`read-and-convert-external-image`**. The default value of *image-translation-table* is **`*default-image-translation-table*`**.

See also

`*default-image-translation-table*`
`read-and-convert-external-image`

set-graphics-port-coordinates *Function*

Summary

Modifies the *transform* of a port such that the edges of the port correspond to the arguments given.

Package

graphics-ports

Signature

set-graphics-port-coordinates *port* **&key** *left top right bottom*

Arguments

port⇓	A graphics port.
left⇓	A real number.
top⇓	A real number.
right⇓	A real number.
bottom⇓	A real number.

Description

The function **set-graphics-port-coordinates** modifies the *transform* of the graphics port *port* permanently such that the edges of *port* correspond to the rectangle (*left top right bottom*).

Notes

The *transform* is part of the port's graphics state. See **graphics-state** for details of how it is used.

Examples

The following code:

```
(set-graphics-port-coordinates port :left -1.0
                                    :top 1.0
                                    :right 1.0
                                    :bottom -1.0)
```

changes the coordinates of the port so that the point (0 0) is in the exact center of the port and the edges are a unit distance away, with a right-handed coordinate system.

By default, *left* and *top* are 1.

See also

graphics-state

set-graphics-state *Function*

Summary

Directly alters the **graphics-state** of a graphics port according to the keyword arguments supplied.

Package

graphics-ports

Signature

set-graphics-state *port* **&rest** *args* **&key** *transform foreground background operation stipple pattern fill-style thickness scale-thickness dashed dash line-end-style line-joint-style mask mask-x mask-y font shape-mode text-mode compositing-mode mask-transform*

Arguments

port⇓ A graphics port.

args⇓ Keywords and values to initialize a **graphics-state**.

transform⇓, *foreground*⇓, *background*⇓, *operation*⇓, *stipple*⇓, *pattern*⇓, *fill-style*⇓, *thickness*⇓, *scale-thickness*⇓, *dashed*⇓, *dash*⇓, *line-end-style*⇓, *line-joint-style*⇓, *mask*⇓, *mask-x*⇓, *mask-y*⇓, *font*⇓, *shape-mode*⇓, *text-mode*⇓, *compositing-mode*⇓, *mask-transform*⇓

See **graphics-state** for interpretation of the arguments.

Description

The function **set-graphics-state** directly alters the graphics state of *port* according to the values of the keyword arguments *transform*, *foreground*, *background*, *operation*, *stipple*, *pattern*, *fill-style*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-end-style*, *line-joint-style*, *mask*, *mask-x*, *mask-y*, *font*, *shape-mode*, *text-mode*, *compositing-mode* and *mask-transform*. Unspecified keywords leave the associated slots unchanged. The keyword arguments *args* correspond to the slots in the graphics state, as described in **graphics-state**.

See also

graphics-state
with-graphics-state
13 Drawing - Graphics Ports

transform *Type*

Summary

The transform type, defined for transform objects.

Package

graphics-ports

Signature

transform

Description

The type **transform** is the type defined for transform objects, which are six-element lists of numbers.

Notes

For information about how transforms are used, see **graphics-state**.

See also

graphics-port-transform
6 Laying Out CAPI Panes
13 Drawing - Graphics Ports

transform-area *Function*

Summary

Transforms a set of points and returns the resulting rectangle.

Package

graphics-ports

Signature

transform-area *transform x y width height* **=>** *rectangle*

Arguments

transform⇓	A **transform**.
x⇓	A real number.
y⇓	A real number.
width⇓	A real number.
height⇓	A real number.

Values

rectangle	A rectangle.

Description

The function **transform-area** transforms the points (*x y*) and (*x*+*width y*+*height*) using *transform* and returns the transformed rectangle as (*x y width height*) values.

See also

transform

transform-distance *Function*

Summary

Transforms a distance vector by the rotation and scale of a transform.

Package

graphics-ports

Signature

transform-distance *transform dx dy* **=>** *dx2*, *dy2*

Arguments

transform⇓	A **transform**.
dx⇓	A real number.
dy⇓	A real number.

Values

dx2	A real number.
dy2	A real number.

Description

The function **transform-distance** transforms the distance (*dx dy*) by the rotation and scale in *transform*. The translation in *transform* is ignored. The transformed distance is returned as two values.

See also

transform

transform-distances *Function*

Summary

Transforms a list of alternating distance vectors by a given transform.

Package

graphics-ports

Signature

transform-distances *transform distances* **=>** *result*

Arguments

transform⇓	A **transform**.
distances⇓	A list of pairs of real numbers.

Values

result	A list of pairs of real numbers.

Description

The function **`transform-distances`** transforms a list of alternating (*dx dy*) pairs in *distances* by *transform*. The transformed distances are returned as a new list.

See also

`transform`

transform-is-rotated *Function*

Summary

Returns **`t`** if a given transform contains a rotation.

Package

`graphics-ports`

Signature

`transform-is-rotated` *transform* **`=>`** *bool*

Arguments

transform⇓ A **`transform`**.

Values

bool A boolean.

Description

The function **`transform-is-rotated`** returns **`t`** if *transform* contains any rotation.

See also

`transform`

transform-point *Function*

Summary

Transforms a point by multiplying it by a transform.

Package

`graphics-ports`

Signature

`transform-point` *transform x y* **`=>`** *xnew ynew*

Arguments

transform⇓	A **`transform`**.
x⇓	A real number.
y⇓	A real number.

Values

xnew	A real number.
ynew	A real number.

Description

The function **`transform-point`** transforms the point (*x y*) by multiplying it by *transform*. The transformed point is returned as two values.

See also

`transform`

transform-points *Function*

Summary

Transforms a list of points by a transform.

Package

`graphics-ports`

Signature

`transform-points` *transform points* **`&optional`** *into* **`=>`** *result*

Arguments

transform⇓	A **`transform`**.
points⇓	A list of pairs of real numbers.
into⇓	A list.

Values

result	A list of pairs of real numbers.

Description

The function **`transform-points`** transforms a list of alternating (*x y*) pairs in *points* by multiplying them by *transform*. If *into* is supplied it is modified to contain the result and must be a list the same length as *points*. If *into* is not supplied, a new list is returned.

See also

`transform`

transform-rect *Function*

Summary

Returns the transform of two points representing the top-left and bottom-right of a rectangle.

Package

`graphics-ports`

Signature

`transform-rect` *transform left top right bottom* **`=>`** *left2*, *top2*, *right2*, *bottom2*

Arguments

transform⇓	A **`transform`**.
left⇓	A real number.
top⇓	A real number.
right⇓	A real number.
bottom⇓	A real number.

Values

left2	A real number.
top2	A real number.
right2	A real number.
bottom2	A real number.

Description

The function **`transform-rect`** transforms the rectangle represented by the two points (*left top*) and (*right bottom*) by *transform*.

See also

`transform`

undefine-font-alias *Function*

Summary

Removes a font alias.

Package

`graphics-ports`

Signature

`undefine-font-alias` *keyword*

Arguments

keyword⇓	A keyword.

Description

The function **`undefine-font-alias`** removes the font alias named by *keyword*.

union-rectangle *Macro*

Summary

Modifies a rectangle to be a union of itself and another rectangle.

Package

`graphics-ports`

Signature

`union-rectangle` *rectangle left top right bottom* **`=>`** *rectangle*

Arguments

rectangle⇓	A rectangle.
left⇓	A real number.
top⇓	A real number.
right⇓	A real number.
bottom⇓	A real number.

Values

rectangle	A rectangle.

Description

The macro **union-rectangle** modifies *rectangle* to be the union of *rectangle* and the rectangle specified by (*left top right bottom*).

unit-transform *Variable*

Summary

The list **(1 0 0 1 0 0)**.

Package

graphics-ports

Initial Value

(1 0 0 1 0 0)

Description

The variable ***unit-transform*** holds the list **(1 0 0 1 0 0)** which is the unit transform I, such that X = XI, where X is a 3-vector. Graphics ports are initialized with the unit transform in their **graphics-state**. This means that port coordinate axes are initially the same as the window axes.

See also

graphics-state

unit-transform-p *Function*

Summary

Returns **t** if a given transform is a unit transform.

Package

graphics-ports

Signature

unit-transform-p *transform* **=>** *bool*

Arguments

transform⇓	A **transform**.

Values

bool	A boolean.

Description

The function **`unit-transform-p`** returns **`t`** if *transform* is the unit transform.

Notes

See **`graphics-state`** for details of how a **`transform`** is used.

See also

`graphics-state`

unless-empty-rect-bind *Macro*

Summary

Binds the elements of a rectangle to four variables, and if the rectangle has a non-zero area, executes a body of code.

Package

`graphics-ports`

Signature

`unless-empty-rect-bind ((`*x y width height***`)`** *rectangle***`) &body`** *body* **`=>`** *result*

Arguments

x⇓	A variable.
y⇓	A variable.
width⇓	A variable.
height⇓	A variable.
rectangle⇓	A rectangle.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **`unless-empty-rect-bind`** binds *x*, *y*, *width*, and *height* to the appropriate values from *rectangle* and if *width* and *height* are both positive, evaluates the forms in *body* as an implicit **`progn`**.

untransform-distance *Function*

Summary

Transforms a distance by the rotation and scale of the inverse of a given transform.

Package

`graphics-ports`

Signature

`untransform-distance` *transform dx dy* **`=>`** *x*, *y*

Arguments

transform⇓	A **`transform`**.
dx⇓	A real number.
dy⇓	A real number.

Values

x	A real number.
y	A real number.

Description

The function **`untransform-distance`** transforms the distance (*dx dy*) by the rotation and scale of the effective inverse of *transform*. The translation in the inverse transform is ignored. The transformed distance is returned as two values.

Notes

See **`graphics-state`** for details of how a **`transform`** is used.

See also

`graphics-state`
`transform`

untransform-distances *Function*

Summary

Transforms a list of integer pairs representing distances by the inverse of a **`transform`**.

Package

`graphics-ports`

Signature

untransform-distances *transform distances* **=>** *result*

Arguments

transform⇓	A **transform**.
distances⇓	A list of pairs of real numbers.

Values

result	A list of pairs of real numbers.

Description

The function **untransform-distances** transforms a list of alternating (*dx dy*) pairs in *distances* by the effective inverse of *transform*. Transformed values are returned as a new list.

Notes

See **graphics-state** for details of how a **transform** is used.

See also

graphics-state
transform

untransform-point *Function*

Summary

Transforms a point by multiplying it by the inverse of a given transform.

Package

graphics-ports

Signature

untransform-point *transform x y* **=>** *x2***,** *y2*

Arguments

transform⇓	A **transform**.
x⇓	A real number.
y⇓	A real number.

Values

x2	A real number.
y2	A real number.

Description

The function **`untransform-point`** transforms the point (*x y*) by effectively multiplying it by the inverse of *transform*. The transformed point is returned as two values.

untransform-points *Function*

Summary

Transforms a list of points by the inverse of a given transform.

Package

`graphics-ports`

Signature

`untransform-points` *transform points* **`&optional`** *into* **`=>`** *result*

Arguments

transform⇓ A **`transform`**.

points⇓ A list of pairs of real numbers.

into⇓ A list.

Values

result A list of pairs of real numbers.

Description

The function **`untransform-points`** transforms a list of alternating (*x y*) pairs in *points* by the effective inverse of *transform*. If *into* is supplied it must be a list the same length as *points*. If *into* is not supplied, a new list is returned.

validate-rectangle *Generic Function*

Summary

Validates the rectangle associated with the object, marks it as already drawn.

Package

`graphics-ports`

Signature

`validate-rectangle` *object* **`&optional`** *x y width height* **`=>`** *result*

Arguments

object⇓ A instance of a subclass of **graphics-port-mixin** or a subclass of **pinboard-object**.

x⇓ A real number.

y⇓ A real number.

width⇓ A real number.

height⇓ A real number.

Values

result⇓ A boolean.

Description

The generic function **validate-rectangle** validates the rectangle associated with *object* and marks it as already drawn.

The given area of *object* is marked as not needing to be displayed. This can be useful if you want to draw that area immediately and avoid it being drawn again by the window system. By default **validate-rectangle** validates the whole rectangle, but this can be limited by passing the optional arguments.

result is non-nil if the function succeeds and **nil** if it fails (doing nothing).

Notes

validate-rectangle is not fully implemented on all platforms.

On Windows, it succeeds for all valid values of *x*, *y*, *width* and *height*.

On Cocoa, it fails if *x*, *y*, *width* and *height* are passed.

On Motif, it fails in all cases.

See also

invalidate-rectangle

with-dither *Macro*

Summary

Specifies a dither for use within a specified body of code.

Package

graphics-ports

Signature

with-dither (*dither-or-size***) &body** *body* **=>** *result*

Arguments

dither-or-size⇓	See Description.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **`with-dither`** specifies a dither for use within *body*. *dither-or-size* can be a dither mask object from **`make-dither`** or a size, in which case a dither of that size is created.

Notes

`with-dither` is deprecated. Dithers do not affect drawing or anti-aliasing.

See also

`dither-color-spec`
`make-dither`
`initialize-dithers`

with-graphics-mask *Macro*

Summary

Binds the *mask* slot of a port's graphics state across the execution of a body of code.

Package

`graphics-ports`

Signature

`with-graphics-mask (`*port mask* **`&key`** *mask-x mask-y mask-transform***`) &body`** *body* **`=>`** *result*

Arguments

port⇓	A graphics port.
mask⇓	**`nil`** or a list specifying a shape.
mask-x⇓, *mask-y*⇓	Integers. These arguments are deprecated.
mask-transform⇓	**`nil`**, **`t`**, the keyword **`:dynamic`**, or a transform.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **with-graphics-mask** binds the *mask* slot of *port*'s **graphics-state** while evaluating the forms in *body* as an implicit **progn**. The mask can be a rectangular area specified by a list of the form (*x y width height*) or a path specified by a list of the form (**:path** *path* **:fill-rule** *fill-rule*).

mask-x and *mask-y* are deprecated. They work only when the *drawing-mode* is **:compatible** and the platform is GTK+ or X11/Motif. By default, *mask-x* and *mask-y* are both 0.

MASK-TRANSFORM is used to set the *mask-transform* graphics state parameter. If *mask-transform* is **nil**, then *mask* will not not transformed. If *mask-transform* is **t**, then *mask* will be transformed by the current graphics state transform at the time that **with-graphics-mask** is used. If *mask-transform* is **:dynamic**, then *mask* will be transformed by the graphics state transform that is in effect when the drawing operation uses the mask. Otherwise *mask-transform* should be a transform object. The default value of *mask-transform* is **nil**.

Notes

See **graphics-state** for more details about *mask* and *mask-transform*.

Examples

This example file demonstrates the use of *mask-transform*:

```
(example-edit-file "capi/graphics/paths")
```

See also

graphics-state
13.3 Graphics state

with-graphics-post-translation *Macro*

Summary

Like **with-graphics-translation** except that the translation is done after applying all existing transforms.

Package

graphics-ports

Signature

with-graphics-post-translation (*port dx dy***) &body** *body* **=>** *result*

Arguments

port⇓	A graphics port.
dx⇓	A real number.
dy⇓	A real number.
body⇓	Lisp forms.

Values

result The value returned by the last form of *body*.

Description

The macro **with-graphics-post-translation** is the same as **with-graphics-translation**, but the translation of (*dx*, *dy*) is done after applying all existing transforms. That means that the translation is "absolute", not transformed. In contrast, when using **with-graphics-translation** the translation is transformed by any existing transform(s).

The forms in *body* are evaluated as an implicit **progn** with the new transform bound to *port*.

Examples

This form draws a 40x40 rectangle at (100,100), because the scale is applied to the coordinates of the rectangle, but not to the translation.

```
(gp:with-graphics-scale (port 2 2)
  (gp:with-graphics-post-translation (port 100 100)
    (gp:draw-rectangle port 0 0 20 20)))
```

Compare with this form, using **with-graphics-translation** instead, which draws a 40x40 rectangle at (200,200), because the scale applies to the translation too:

```
(gp:with-graphics-scale (port 2 2)
  (gp:with-graphics-translation (port 100 100)
    (gp:draw-rectangle port 0 0 20 20)))
```

See also

with-graphics-transform-reset
with-graphics-translation
13.3.1 Setting the graphics state

with-graphics-rotation
with-graphics-scale
with-graphics-translation *Macros*

Summary

Combines a transformation (rotation, scaling or translation) with the transform of a port for the duration of the macro.

Package

graphics-ports

Signatures

with-graphics-rotation (*port angle***) &body** *body* **=>** *result*

with-graphics-scale (*port sx sy***) &body** *body* **=>** *result*

with-graphics-translation (*port dx dy***) &body** *body* **=>** *result*

Arguments

port⇓	A graphics port.
angle⇓	A real number.
body⇓	A body of Lisp code.
sx⇓, *sy*⇓	Real numbers.
dx⇓, *dy*⇓	Real numbers.

Values

result	The return value(s) of the last form executed in *body*.

Description

The macros **`with-graphics-rotation`**, **`with-graphics-scale`** and **`with-graphics-translation`** combine the transform associated with *port* with an additional transform while evaluated the forms in *body* as an implicit **`progn`**. *port* is given a new transform obtained by pre-multiplying its current transform with the transform that the macro creates.

`with-graphics-rotation` creates a transformation that rotates by *angle* radians. If *angle* is positive, then the rotation is clockwise.

`with-graphics-scale` creates a transformation that scales by *sx* and *sy* in the X and Y dimensions.

`with-graphics-translation` creates a transformation that translates by *dx* and *dy* in the X and Y dimensions.

Notes

1. These macros do the same as **`with-graphics-transform`** does with an appropriate transform.
2. The transform associated with a graphics port is part of the port's graphics state. See **`graphics-state`** for details.

Examples

```
(example-edit-file "capi/graphics/catherine-wheel")
```

See also

`graphics-state`
`with-graphics-post-translation`
`with-graphics-transform`
13.6 Graphics state transforms
13.3.1 Setting the graphics state

with-graphics-state *Macro*

Summary

Binds the graphics state values of a port to a list of arguments and executes a body of code.

Package

graphics-ports

Signature

with-graphics-state (*port* **&rest** *args* **&key** *state transform foreground background operation stipple pattern fill-style thickness scale-thickness dashed dash line-end-style line-joint-style mask mask-x mask-y font shape-mode text-mode compositing-mode mask-transform***)** *body* **=>** *result*

Arguments

port⇓ A graphics port.

args⇓ Keywords and values to initialize a **graphics-state**.

state⇓ A **graphics-state** or **nil**.

transform⇓, *foreground*⇓, *background*⇓, *operation*⇓, *stipple*⇓, *pattern*⇓, *fill-style*⇓, *thickness*⇓, *scale-thickness*⇓, *dashed*⇓, *dash*⇓, *line-end-style*⇓, *line-joint-style*⇓, *mask*⇓, *mask-x*⇓, *mask-y*⇓, *font*⇓, *shape-mode*⇓, *text-mode*⇓, *compositing-mode*⇓, *mask-transform*⇓

See **graphics-state** for interpretation of the arguments.

body⇓ A body of Lisp code.

Values

result The return value of the last form executed in *body*.

Description

The macro **with-graphics-state** binds the graphics state values for *port* according to the values of the keyword arguments *transform*, *foreground*, *background*, *operation*, *stipple*, *pattern*, *fill-style*, *thickness*, *scale-thickness*, *dashed*, *dash*, *line-end-style*, *line-joint-style*, *mask*, *mask-x*, *mask-y*, *font*, *shape-mode*, *text-mode*, *compositing-mode* and *mask-transform*. Unspecified keywords leave the associated slots unchanged. The keyword arguments *args* correspond to the slots in the graphics state, as described in **graphics-state**.

If *state* is non-nil then the **graphics-state** of *port* is bound to it before the other keywords are processed.

The forms in *body* are evaluated as an implicit **progn** with the new graphics stare bound to *port*.

For example:

```
(with-graphics-state (port :thickness 12 :foreground my-color) ...)
```

Arguments that are not supplied default to the current state of that slot in the **graphics-state**. *stipple* is used only on X11/Motif.

mask-x and *mask-y* are deprecated. They work only when the *drawing-mode* is **:compatible** and the platform is GTK+ or X11/Motif.

Examples

```
(setf gstate (make-graphics-state))

(setf (graphics-state-foreground gstate) my-color)
```

```
(with-graphics-state (port :state gstate)
  (draw-rectangle port image-1 100 100))
```

See also

graphics-state
set-graphics-state
with-graphics-translation
with-graphics-post-translation
with-graphics-scale
with-graphics-rotation
with-graphics-transform
with-graphics-transform-reset
with-graphics-mask
13 Drawing - Graphics Ports

with-graphics-transform *Macro*

Summary

Combines a given transform with the transform of a port for the duration of the macro.

Package

graphics-ports

Signature

with-graphics-transform (*port transform***) &body** *body* **=>** *result*

Arguments

port⇓	A graphics port.
transform⇓	A **transform**.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **with-graphics-transform** combines the transform associated with *port* with *transform* while evaluating the forms of *body* as an implicit **progn**. *port* is given a new transform obtained by pre-multiplying its current transform with *transform*. This has the effect of *preceding* any translation, scaling and rotation operations specified in the body of the macro by those operations embodied in *transform*.

Notes

See **graphics-state** for details of how a **transform** is used.

Examples

```
(example-edit-file "capi/graphics/metafile-rotation")
```

See also

graphics-state
transform

with-graphics-transform-reset *Macro*

Summary

Like **with-graphics-transform** except that it ignores existing transforms.

Package

graphics-ports

Signature

with-graphics-transform-reset (*port* **&optional** *transform***) &body** *body* **=>** *result*

Arguments

port⇓ A graphics port.

transform⇓ A **transform**.

body⇓ Lisp forms.

Values

result The value returned by the last form of *body*.

Description

The macro **with-graphics-transform-reset** works the same as **with-graphics-transform** except that it ignores existing transforms.

If *transform* is **nil**, then *body* is evaluated without any transform in *port* (that is, with the unit transform).

Examples

This form ignores the translation, and applies only the explicit transform (which is really just scale), so that the overall effect is to draw a 30x20 rectangle at (0,0).

```
(gp:with-graphics-translation (port 100 100)
  (gp:with-graphics-transform-reset (port (gp:make-transform 3 0 0 2 0 0 ))
    (gp:draw-rectangle port 0 0 10 10)))
```

Compare with using **with-graphics-transform**, which applies both the translation and the explicit transform, so that the overall effect is to draw a rectangle 30x20 at (100,100).

```
(gp:with-graphics-translation (port 100 100)
  (gp:with-graphics-transform (port (gp:make-transform 3 0 0 2 0 0 ))
    (gp:draw-rectangle port 0 0 10 10)))
```

See also

with-graphics-post-translation
with-graphics-transform

with-inverse-graphics *Macro*

Summary

Executes all drawing function calls to a given port within the body of the macro with *foreground* and *background* colors swapped.

Package

graphics-ports

Signature

with-inverse-graphics (*port***) &body** *body* **=>** *result*

Arguments

port⇓	A graphics port.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **with-inverse-graphics** evaluates the forms in *body* as an implicit **progn** with the *foreground* and *background* slots of the **graphics-state** of *port* swapped.

without-relative-drawing *Macro*

Summary

Evaluates a body of Lisp code with the *relative* and *collect* internal variables of the port set to **nil**.

Package

graphics-ports

Signature

without-relative-drawing (*port***) &body** *body* **=>** *result*

Arguments

port⇓	A graphic port.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **without-relative-drawing** evaluates the forms in *body* as an implicit **progn** with the *relative* and *collect* internal variables of the pixmap graphics port *port* set to **nil** to turn off the port's collecting of drawing bounds and automatic shifting of its origins. Use this macro only within a **with-pixmap-graphics-port** macro.

with-pixmap-graphics-port *Macro*

Summary

Binds a port to a new pixmap graphics port for the duration of the macro's code body.

Package

graphics-ports

Signature

with-pixmap-graphics-port (*port pane width height* **&key** *background foreground collect relative clear drawing-mode***) &body** *body* **=>** *result*

Arguments

port⇓	A graphics port.
pane⇓	An output pane.
width⇓	An integer.
height⇓	An integer.
background⇓	A color specification, or **nil**.
foreground⇓	A color specification, or **nil**.
collect⇓	A boolean.
relative⇓	A boolean.
clear⇓	A list or **t**.
drawing-mode⇓	One of the keywords **:compatible** and **:quality**.
body⇓	A body of Lisp code.

Values

result The return value of the last form executed in *body*.

Description

The macro **`with-pixmap-graphics-port`** binds *port* to a new pixmap graphics-port.

pane, *width*, *height*, *background*, *foreground*, *collect*, *relative*, *clear* AND *drawing-mode* are used as specified by **`create-pixmap-port`**. The forms in *body* are then evaluated as an implicit **`progn`**. *port* is destroyed when *body* returns.

Examples

In the code below the background in *p2* inherits from *p1*, so it draws two green rectangles.

```
(let ((op (capi:contain
           (make-instance 'capi:output-pane
                          :background :red))))
  (sleep 0.1)
  (gp:with-pixmap-graphics-port (p1 op 20 30
                                    :background :green
                                    :clear t)
    (gp:with-pixmap-graphics-port (p2 p1 20 30 :clear t)
      (gp:copy-pixels op p1 10 10 20 30 0 0)
      (gp:copy-pixels op p2 10 60 20 30 0 0))))
```

See also

`create-pixmap-port`
13 Drawing - Graphics Ports

with-transformed-area *Macro*

Summary

Transforms a rectangle using a port's transform, and binds the resulting values to a variable across the evaluation of the macro's body.

Package

`graphics-ports`

Signature

`with-transformed-area (`*points port left top right bottom***`) &body`** *body* **`=>`** *result*

Arguments

points⇓ A variable.

port⇓ A graphics port.

left⇓ A real number.

top⇓ A real number.

right⇓	A real number.
bottom⇓	A real number.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **`with-transformed-area`** uses *port*'s *transform* to transform a rectangle specified by *left*, *top*, *right* AND *bottom*. Then *points* is bound to the resulting list of eight values (alternating *x* and *y* values for four corner points) while the forms of *body* are evaluated as an implicit **`progn`**.

with-transformed-point *Macro*

Summary

Binds a point transformed by a given ports transform to two variables across the body of the macro.

Package

`graphics-ports`

Signature

`with-transformed-point (`*new-x new-y port x y***`) &body`** *body* **`=>`** *result*

Arguments

new-x⇓	A variable.
new-y⇓	A variable.
port⇓	A graphics port.
x⇓	A real number.
y⇓	A real number.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **`with-transformed-point`** transforms the point given by (*x y*) using *port*'s transform and *new-x* and *new-y* are bound to the transformed point. The forms in *body* are then evaluated as an implicit **`progn`** with this binding.

with-transformed-points *Macro*

Summary

Binds a list of transformed points in a port to a list across the execution of the macro's body.

Package

graphics-ports

Signature

with-transformed-points (*points port***) &body** *body* **=>** *result*

Arguments

points⇓	A list of real numbers.
port⇓	A graphics port.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **with-transformed-points** binds *points* to a new list of *x* and *y* values obtained by post-multiplying them by the current transform of *port*, and then evaluates the forms in *body* as am implicit **progn**. *points* must be bound to a list of alternating *x* and *y* values representing coordinate points in *port*.

with-transformed-rect *Macro*

Summary

Transforms the coordinates of a rectangle and binds them to variables while executing a body of code.

Package

graphics-ports

Signature

with-transformed-rect (*nx1 ny1 nx2 ny2 port x1 y1 x2 y2***) &body** *body* **=>** *result*

Arguments

nx1⇓	A variable.
ny1⇓	A variable.

nx2⇓	A variable.
ny2⇓	A variable.
port⇓	A graphics port.
x1⇓	A real number.
y1⇓	A real number.
x2⇓	A real number.
y2⇓	A real number.
body⇓	A body of Lisp code.

Values

result	The return value of the last form executed in *body*.

Description

The macro **`with-transformed-rect`** transforms the coordinates of a rectangle and binds them to four variables for the duration of the macro's body.

During the evaluation of the forms in *body*, the two points (*x1*, *y1*) and (*x2*, *y2*) are transformed by the current transform of *port* and the resulting values are bound to the variables *nx1*, *ny1*, *nx2* and *ny2*.

write-external-image *Function*

Summary

Writes external image data to a file.

Package

`graphics-ports`

Signature

`write-external-image` *external-image destination* **`&key`** *if-exists*

Arguments

external-image⇓	An **`external-image`**.
destination⇓	A pathname designator.
if-exists⇓	A keyword.

Description

The function **`write-external-image`** writes *external-image* to *destination*. If *destination* is a stream, it must be an output stream with element type compatible with **`(unsigned-byte 8)`**, that is one of **`cl:base-char`**, **`(signed-byte 8)`** and **`(unsigned-byte 8)`**. If *destination* is a pathname or namestring the file is opened for output with the correct element type, and **`write-external-image`** writes the bytes to the resulting stream as if by **`cl:write-sequence`**.

if-exists is passed to **open** when opening *file*. The default value of *if-exists* is **:error**.

See also

externalize-image
13.10.3 External images

23 LW-GT Reference Entries

This chapter provides reference entries for the symbols exported from the **lw-gt** package. This package is for the Graphic Tools, which are interfaces which use Graphics Ports and CAPI. These contain the drawing objects, which add a mechanism to creates a hierarchy of drawing, when a "drawing" is (typically) a simple Graphics Ports drawing operation. The hierarchy specifies the geometry of each node in the hierarchy, so the whole group drawings can be manipulated as a single object.

To use Graphic Tools, you first need to load the module "graphic-tools", like this:

```
(require "graphic-tools")
```

See **14 Graphic Tools drawing objects** for an overview of Graphic Tools.

See **1 Introduction to the CAPI** for an overview of CAPI, and **13 Drawing - Graphics Ports** for more information on Graphics Ports.

apply-drawing-object *Class*

Summary

A **drawing-object** that applies a supplied function to supplied arguments.

Package

lw-gt

Superclasses

drawing-object

Description

The class **apply-drawing-object** is a **drawing-object** that applies a supplied function to a list of supplied arguments, normally preceded by the **objects-displayer**. Its main usage is for doing the actual drawing.

apply-drawing-objects can be used repeatedly and concurrently in the same or different panes. The ones that are created by the **make-draw-*** functions (**make-draw-arc** and so on) are fixed, but for objects created by **make-a-drawing-call**, the supplied function may depend on values that change, and hence needs to be redisplayed when these values change. Use **force-objects-redraw** on the root of the hierarchy (an **objects-displayer** or a **pinboard-objects-displayer**) to do that.

See **drawing-object** for description of the drawing operation.

See also

objects-displayer
pinboard-objects-displayer
position-object
fit-object

position-and-fit-object

basic-graph-spec *System Class*

Summary

Provides a mechanism to simplify generating a graph of a mathematical function which maps x to y.

Package

lw-gt

Superclasses

t

Accessors

basic-graph-spec-function
basic-graph-spec-start-x
basic-graph-spec-step-x
basic-graph-spec-range
basic-graph-spec-color
basic-graph-spec-thickness
basic-graph-spec-name
basic-graph-spec-x-scale
basic-graph-spec-y-scale
basic-graph-spec-x-offset
basic-graph-spec-y-offset
basic-graph-spec-var1
basic-graph-spec-var2
basic-graph-spec-var3
basic-graph-spec-var4
basic-graph-spec-var5
basic-graph-spec-var6

Description

The system class **basic-graph-spec** provides a mechanism to simplify generating a graph of a mathematical function which maps x to y. Create it with **make-basic-graph-spec**.

Notes

1. The **basic-graph-spec** mechanism is intended to make it simpler to repeatedly compute graphs for a function with values that may change. It is a thin layer, and you can implement you own version using **generate-graph-from-pairs**.
2. **basic-graph-spec** is a structure type, and can be included in structures your define to extend the functionality.

See also

make-basic-graph-spec
14.2 Higher level - drawing graphs and bar charts

compound-drawing-object *Class*

Summary

A **drawing-object** that draws the "child" **drawing-object** in its *sub-object* slot.

Package

lw-gt

Superclasses

drawing-object

Subclasses

geometry-drawing-object

Accessors

compound-drawing-object-sub-object
compound-drawing-object-data

Description

The class **compound-drawing-object** is a **drawing-object** that has a "child" **drawing-object** in its *sub-object* slot. The **compound-drawing-object** draws the "child".

The main usage of **compound-drawing-object** is through its subclass **geometry-drawing-object**, which manipulates the geometry around drawing the objects. See **geometry-drawing-object**.

It is possible to set the *sub-object* slot in a **compound-drawing-object** using **(setf compound-drawing-object-sub-object)**. This can be done on any thread. This setting does not cause automatic redisplay of the object. The redisplay happens next the time the hierarchy is redisplayed. You can force the redisplay by calling **force-objects-redraw**.

compound-drawing-object should not be made by **cl:make-instance**. See **geometry-drawing-object** for how to make it.

The accessor **compound-drawing-object-data** can be used to read and set the *data* slot in the **compound-drawing-object**. You can use the *data* slot to store related information, and it is used by **compute-drawing-object-from-data**.

See also

objects-displayer
pinboard-objects-displayer
14.1 Lower level - drawing objects and objects displayers

compute-drawing-object-from-data
recurse-compute-drawing-object

Functions

Summary

Use the *function* and/or *data* in **compound-drawing-object**s.

Package

lw-gt

Signatures

compute-drawing-object-from-data *object* **=>** *result*

recurse-compute-drawing-object *object-or-displayer*

Arguments

object⇓	A Lisp object.
object-or-displayer⇓	An **objects-displayer**, **pinboard-objects-displayer**, a list, or a **compound-drawing-object**.

Values

result	A boolean.

Description

The function **compute-drawing-object-from-data** computes the drawing for an object.

If *object* is not a **compound-drawing-object**, then **compute-drawing-object-from-data** just returns **nil**.

If *object* is a **compound-drawing-object**, then **compute-drawing-object-from-data** checks if *object* has a non-nil value for either *function* or *data*. For *object* to have a non-nil *function*, this must have been supplied when *object* was created (for example when creating **geometry-drawing-object**). *data* can be passed during creation or set later by using **setf** with **compound-drawing-object-data**.

If *object* has a non-nil *function*, then **compute-drawing-object-from-data** calls *function* with *data* as a single argument, and uses the result. Otherwise, if *object* has a non-nil *data*, **compute-drawing-object-from-data** calls the generic function **get-drawing-object** with *data* as a single argument, and uses the result. If this result is **:no-change**, **compute-drawing-object-from-data** just returns **nil**. **get-drawing-object** has a default method that returns **:no-change**.

Otherwise, the result must be a "drawing-object-spec", which means either an instance of (a subclass of) **drawing-object** or a list of "drawing-object-specs". **compute-drawing-object-from-data** then sets the *sub-object* of the object to the result, and returns **t**.

For **recurse-compute-drawing-object**, *object-or-displayer* should be an **objects-displayer**, a **pinboard-objects-displayer**, a list, or a **compound-drawing-object**. For other objects **recurse-compute-drawing-object** just returns **nil**.

recurse-compute-drawing-object recurses the hierarchy starting at *object-or-displayer*, and for each

compound-drawing-object that it finds calls **compute-drawing-object-from-data**.

When it finds an **objects-displayer** or a **pinboard-objects-displayer**, **recurse-compute-drawing-object** also calls **force-objects-redraw** when it finishes.

These functions can be called on any thread.

Notes

1. The purpose of these functions is to allow creating a tree of **drawing-object**s that can update itself, by passing the *function* argument when making it or defining **get-drawing-object** and passing the appropriate *data*. Then the tree can be told to recompute itself by calling **recurse-compute-drawing-object**.
2. These functions do not cause redraw, except when **recurse-compute-drawing-object** is applied to **objects-displayer** or **pinboard-objects-displayer**. You will have to do it yourself by using **force-objects-redraw** on the root of the hierarchy or hierarchies which need redrawing.
3. **recurse-compute-drawing-object** does not check against duplication, so if the same object appears in the hierarchy more than once, it will be updated repeatedly.

See also

geometry-drawing-object
compound-drawing-object
14.1 Lower level - drawing objects and objects displayers

drawing-object *Class*

Summary

The root class for drawing objects.

Package

lw-gt

Superclasses

t

Subclasses

compound-drawing-object
apply-drawing-object
string-drawing-object

Description

The class **drawing-object** is the root class for drawing objects, which are used to create hierarchies of drawings. The hierarchy is made of **compound-drawing-object** objects, which group other drawing objects and affect their geometry, lists of **drawing-object**s, and leaf drawing objects (currently **apply-drawing-object** and **string-drawing-object**), which actually do the drawing.

A **drawing-object** is part of the hierarchy when it is in the *drawing-object* slot of an **objects-displayer** or a **pinboard-objects-displayer**, or it is inside a list which is in a hierarchy, or it is in the *sub-object* slot of a

compound-drawing-object. The root of the hierarchy is always an **objects-displayer** or a **pinboard-objects-displayer**. A node in the hierarchy (except the root) is either a **drawing-object** or a list, which is collectively called "drawing-object-spec". In a list all the elements must be "drawing-object-specs".

drawing-object can concurrently appear multiple times in the same or different hierarchies, in the same or different panes and same or different interfaces.

Drawing **drawing-object**s is always done top-down: the root object draws its *drawing-object*. Typically this is either a **compound-drawing-object** or a list, which will draw their *sub-object* or elements respectively. Each object which is a **geometry-drawing-object** does something to the geometry, that is set up some Graphics Ports transformation, and then draw all its objects inside this context. For lists the elements are drawn in the same context in which the list is drawn. Leaf **drawing-object**s actually draw something.

parent, root, and root pane

When the drawing operation reaches a **drawing-object**, it is because it is inside the hierarchy inside a **compound-drawing-object** or directly inside the hierarchy under an **objects-displayer** or a **pinboard-objects-displayer**. This **compound-drawing-object**, **objects-displayer** or **pinboard-objects-displayer** is the "parent" of the **drawing-object** for this drawing operation, and determines its geometry. During the drawing operation there is also the "root" (the **objects-displayer** or **pinboard-objects-displayer** from which the drawing started), and the "root pane" (the **objects-displayer** when the root is an **objects-displayer**, or the pane of the **pinboard-objects-displayer**).

Note that "parent", "root" and "root pane" of a **drawing-object** are transient concepts, and are applicable only inside the context of a drawing operation of the **drawing-object**. The same **drawing-object** may be drawn many times, with (potentially) different "parent", "root" and "root pane". It can be even drawn concurrently with different "root panes".

Notes

drawing-objects should not be made by **cl:make-instance**. See the entries for the subclasses for how to make them.

See also

objects-displayer
pinboard-objects-displayer
14.1 Lower level - drawing objects and objects displayers

fit-object
make-absolute-drawing
make-absolute-drawing*
position-object
position-and-fit-object
rotate-object

Functions

Summary

Create a **geometry-drawing-object**, where the *sub-object* is the **drawing-object**.

Package

lw-gt

Signatures

fit-object *drawing-object intended-width intended-height* **&key** *data function* **=>** *geometry-drawing-object*

make-absolute-drawing &rest *drawing-objects* **=>** *geometry-drawing-object*

make-absolute-drawing* *drawing-objects* **=>** *geometry-drawing-object*

position-object *drawing-object* **&key** *left-margin left-ratio right-margin right-ratio top-margin top-ratio bottom-margin bottom-ratio data function* **=>** *geometry-drawing-object*

position-and-fit-object *drawing-object intended-width intended-height* **&key** *left-margin left-ratio right-margin right-ratio top-margin top-ratio bottom-margin bottom-ratio data function* **=>** *geometry-drawing-object*

rotate-object *drawing-object angle* **&key** *left-margin left-ratio bottom-margin bottom-ratio data function* **=>** *geometry-drawing-object*

Arguments

drawing-object⇓ A "drawing-object-spec".

intended-width⇓, *intended-height*⇓
Real numbers or **nil**.

data⇓ Any Lisp object.

function⇓ A function designator or **nil**.

drawing-objects⇓ A list of "drawing-object-specs".

left-margin⇓, *left-ratio*⇓
Real numbers or **nil**.

right-margin⇓, *right-ratio*⇓
Real numbers or **nil**.

top-margin⇓, *top-ratio*⇓
Real numbers or **nil**.

bottom-margin⇓, *bottom-ratio*⇓
Real numbers or **nil**.

angle⇓ A real number or **nil**.

Values

geometry-drawing-object
A **geometry-drawing-object**.

Description

The functions **fit-object**, **make-absolute-drawing**, **make-absolute-drawing***, **position-object**, **position-and-fit-object** and **rotate-object** are the "geometry" functions. Each creates a **geometry-drawing-object**, where the *sub-object* slot contains *drawing-object*.

Each *drawing-object* argument must be a "drawing-object-spec", which means either an instance of (a subclass of) **drawing-object** or a list of "drawing-object-specs".

position-object

When drawing, the **geometry-drawing-object** created by **position-object** computes its own position and size based on the keyword arguments and the position and size of its parent (see **drawing-object** for the meaning of "parent"). It then establishes a Graphics Ports translation to translate from its parent's left/bottom corner to its own left/bottom corner, and draws its *sub-object*.

left-margin, *left-ratio*, *right-margin*, *right-ratio*, *top-margin*, *top-ratio*, *bottom-margin* and *bottom-ratio* specify how to compute the left, right, bottom and top of the positioning object with respect to its parent. For each side, the value is computed by multiplying the ratio by the relevant dimension (width for left and right, height for top and bottom), and then add (for left and bottom) or subtract (for right and top) the margin. Note that the vertical coordinate is 0 at the bottom and increases towards the top.

The default values of *right-ratio* and *top-ratio* are 1, and the default values of all the other keyword arguments are 0, making it compute the same position and size as the parent.

Notes

1. The width and height of a positioning object are not used explicitly, but will be used by any child object that is itself a **geometry-drawing-object**.
2. A positioning **geometry-drawing-object** does not cause any scaling.
3. Calling **position-object** without passing right and top values is a useful way to just shift objects around, but the resulting width and height are probably not useful. If *drawing-object* contains drawing objects that need the width and height (result of **fit-object**, **position-and-fit-object**, or **rotate-object**), you probably need to set the right and top too.

fit-object

When drawing, the **geometry-drawing-object** created by **fit-object** computes scaling factors for the horizontal and vertical dimensions by dividing its width and height, which it inherits from its parent, by its *intended-width* and *intended-height*. It then establishes a Graphics Ports scaling transformation with these factors, and draws its *sub-object*.

position-and-fit-object

position-and-fit-object creates a **drawing-object** that performs the equivalent of using **position-object** with the result of calling **fit-object** with *drawing-object*. In other words, it first positions and then fits.

rotate-object

When drawing, the **geometry-drawing-object** created by **rotate-object** computes the transform for rotating the object by *angle* radians around the point specified by the keyword arguments (default to left-bottom corner). *left-margin*, *left-ratio*, *bottom-margin* and *bottom-ratio* are used to compute the center of rotation, using the same algorithm as in **position-object**.

rotate-object does not affect the width and height of the drawing, but since the drawing itself is rotated, the direction in which the width and height apply are rotated too. For example, if you rotate by **pi/2**, the width is in the vertical dimension on the screen.

make-absolute-drawing and make-absolute-drawing*

make-absolute-drawing and **make-absolute-drawing*** create an object that displays *drawing-objects* in "absolute mode", which means drawing without scaling or rotation, but still taking account of the translation. When using a metafile, the absolute drawing is into the metafile. When the metafile is drawn, it normally scales and this scales everything, including absolute drawings.

Notes

1. Inside the "absolute" scope, the y increases downwards rather than upwards.
2. An example where absolute drawing is useful is drawing of strings and some associated **`drawing-object`**s inside a larger object, where you want to allow the larger object to scale and rotate and the strings displayed in the correct place, but you want the strings to be upright and optimal size for readability.

data and *function*

data argument can be anything, and is stored in the **`geometry-drawing-object`**, and can be accessed by **`compound-drawing-object-data`**. It can be used to keep arbitrary data, and is also used by **`compute-drawing-object-from-data`**.

function is used by **`compute-drawing-object-from-data`** only. See **`compute-drawing-object-from-data`**.

`geometry-drawing-object` objects can be used repeatedly and concurrently in the same or different panes. The *sub-object* can be changed dynamically by using **`(setf compound-drawing-object-sub-object)`** from any thread, but if it is already being displayed, you will need to ensure that they are redrawn. See **`force-objects-redraw`**.

See also

`drawing-object`
`compound-drawing-object`
`objects-displayer`
`pinboard-objects-displayer`
`force-objects-redraw`
14.1 Lower level - drawing objects and objects displayers

force-objects-redraw *Function*

Summary

Forces redrawing of objects.

Package

`lw-gt`

Signature

`force-objects-redraw` *pane*

Arguments

pane⇓ An **`objects-displayer`** or a **`pinboard-objects-displayer`**.

Description

The function **`force-objects-redraw`** forces redrawing of the objects in the *drawing-object* slot of *pane*.

pane should be either an **`objects-displayer`** or a **`pinboard-objects-displayer`**. When **`force-objects-redraw`** is called on any other object it silently does nothing.

`force-objects-redraw` uses **`apply-in-pane-process`**, so can be used on any process.

Notes

In the case of **objects-displayer**, **force-objects-redraw** forces redrawing of the *drawing-object* of the **objects-displayer** and the **drawing-object**s and any **pinboard-objects-displayer** objects in the *description* of the **objects-displayer**, but does not force redraw of other **pinboard-object**s. **force-objects-redraw** is needed when you set the *sub-object* slot in any of the **drawing-object**s inside a hierarchy, because setting does not cause automatic redrawing.

See also

objects-displayer
pinboard-objects-displayer
14.1 Lower level - drawing objects and objects displayers

generate-bar-chart *Function*

Summary

Generate a list of **drawing-object**s which display the bars of a bar chart.

Package

lw-gt

Signature

generate-bar-chart *values* **&key** *function start-position step-position width orientation colors title-position argument font base title-color absolute-p* **=>** *bars*

Arguments

values⇓	A list.
function⇓	A function of one or two arguments, depending on *argument*.
start-position⇓	The position of the first bar.
step-position⇓	The distance between bars.
width⇓	The width of a bar.
orientation⇓	One of the keywords **:rightward**, **:leftward**, **:downward** and **:upward**.
colors⇓	A list of colors.
title-position⇓	One of the keywords **:middle**, **:top**, **:bottom**, **:right** and **:left**, or **nil**.
argument⇓	A Lisp object.
font⇓	A font specification.
base⇓	The position of the "base" of each bar.
title-color⇓	A color specification.
absolute-p⇓	A boolean.

Values

bars A list of **drawing-object**s.

Description

The function **generate-bar-chart** generates a list of **drawing-object**s which display the bars of a bar chart.

values is a list giving the values that need displaying. There is a bar for each element in the list.

For each element in *values*, **generate-bar-chart** uses the function *function* to find the length of the bar and a title to add to it. If *argument* is non-nil, *function* is called with two arguments: *argument* and the element of *values*. Otherwise, *function* is called with one argument, the element. *function* must return the length of the bar, and optionally the title as a second return value. The default value of *argument* is **nil**.

If *function* is not supplied, the default function checks if the element is a list, and if it is returns the first element of it as the length and the second element as the title. If it is not a list it returns it and **nil** as the second value.

generate-bar-chart then generates a **drawing-object** that draws the bar, which is a rectangle with length being the result of the function and width *width*. The default value of *width* is 1.

For *orientation* **:upward** or **:downward**, the "length dimension" is vertical, and the "width dimension" is the horizontal, and the reverse for the other orientations. The default value of *orientation* is **:upward**.

The position of the rectangle in the "length dimension" is from *base* to (+ *base* length) for *orientation* **:upward** and **:leftward**, and from *base* to (- *base* length) for the other orientations. The default value of *base* is 0.

start-position and *step-position* determine the position of the center of the rectangle in the "width dimension". Hence the position of the *n*'th rectangle in the "width dimension" is from:

```
(- (+ #START-POSITION (* (1- n) #STEP-POSITION)) (/ #WIDTH 2))
```

to:

```
(+ (+ #START-POSITION (* (1- n) #STEP-POSITION)) (/ #WIDTH 2))
```

The default value of *start-position* is 1. The default value of *step-position* is (* 3 *width*).

The color of the rectangle is taken from the items of *colors* in turn, starting again from the beginning when reaching the end. The default value of *colors* is **(:red :green :blue :yellow :purple)**.

generate-bar-chart then also computes where the string should appear with respect to the bar, depending on *title-position*, generates a drawing object using **make-draw-string**, passing it *font*, *absolute-p* and *title-color*. *title-position* **nil** means the end of the bar. The default value of *font* is the font of the pane. *absolute-p* determines whether the title is drawn in absolute mode. The default value of *absolute-p* is **t**.

See also

drawing-object
14.1 Lower level - drawing objects and objects displayers

generate-graph-from-pairs *Function*

Summary

Generates a drawing object which draws lines connecting points.

Package

`lw-gt`

Signature

`generate-graph-from-pairs` *x-y-pairs* **`&key`** *thickness color x-offset y-offset x-scale y-scale* **`=>`** *drawing-object*

Arguments

x-y-pairs⇓	A list.
thickness⇓	A positive real number.
color⇓	A Color specification.
x-offset⇓, *y-offset*⇓	Non-negative real numbers.
x-scale⇓, *y-scale*⇓	Positive real numbers.

Values

drawing-object	A **`drawing-object`**.

Description

The function **`generate-graph-from-pairs`** generates a "graph", which is a drawing object which draws lines connecting the points in *x-y-pairs*.

x-y-pairs must be a list where each element is a list of length 2 specifying a point as a pair of coordinates (*x*, *y*).

x-scale, *y-scale*, *x-offset* and *y-offset* are used to scale and offset the graph. Each *x* value is multiplied by *x-scale* and then *x-offset* is added, and similarly for the *y* value. The default value of both *x-offset* and *y-offset* is 0. The default value of both *x-scale* and *y-scale* is 1.

thickness specifies the thickness of the line, which is not scaled (it passes **`:scale-thickness nil`** to **`make-draw-lines`**). *thickness* defaults to 1.

color specifies the foreground color of the line *color* defaults to **`:red`**.

Notes

`generate-graph-from-pairs` is a quite thin interface on top of **`make-draw-lines`**. If it does not do what you want, you can easily replace it by your own code.

See also

`generate-graph-from-graph-spec`
`drawing-object`

14.2 Higher level - drawing graphs and bar charts

generate-grid-lines *Function*

Summary

Generate a grid of lines, to be used for drawing graphs of functions or bar charts.

Package

lw-gt

Signature

generate-grid-lines (&key *x-offset y-offset x-spacing y-spacing horizontal-count vertical-count width height thickness vertical-thickness minor-thickness minor-vertical-thickness left-thickness right-thickness top-thickness bottom-thickness major-x-step major-y-step color vertical-color major-color major-vertical-color left-color right-color top-color bottom-color* **) =>** *list*

Arguments

x-offset⇓, *y-offset*⇓ Non-negative real numbers.

x-spacing⇓, *y-spacing*⇓

Positive real numbers.

horizontal-count⇓, *vertical-count*⇓

nil or positive integers.

width⇓, *height*⇓ **nil** or positive real numbers.

thickness⇓, *vertical-thickness*⇓, *minor-thickness*⇓, *minor-vertical-thickness*⇓, *left-thickness*⇓, *right-thickness*⇓, *top-thickness*⇓, *bottom-thickness*⇓

Positive real numbers. Each defaults to 1.

major-x-step⇓, *major-y-step*⇓

nil or integers.

color⇓, *vertical-color*⇓, *major-color*⇓, *major-vertical-color*⇓, *left-color*⇓, *right-color*⇓, *top-color*⇓, *bottom-color*⇓

Colors in the standard definition. Each defaults to **:gray**.

Values

list A list of **drawing-object**s.

Description

The function **generate-grid-lines** generates a grid of lines, to be used for drawing graphs of functions or bar charts.

generate-grid-lines returns a list of **drawing-object**s which when drawn display a grid of horizontal and vertical lines, according to the supplied specification.

The grid is made of vertical lines spaced regularly in the horizontal dimension, and horizontal lines spaced regularly in the vertical dimension. The specification of the graph is conceptual starting from 0 and increasing in both dimensions. This does not affect what values the graph shows, because these are defined by the labels which are produced separately (typically by **`generate-labels`**).

x-offset and *y-offset* specify the offset of the origin of the graph, which means the position of the first horizontal and vertical line respectively, and where the other horizontal and vertical lines start. The default value of both *x-offset* and *y-offset* is 0.

x-spacing and *y-spacing* specify the gaps in the horizontal and vertical dimensions respectively (that is, the distance between the lines). The default value of both *x-spacing* and *y-spacing* is 1.

horizontal-count and *vertical-count* specify the numbers of lines in the horizontal and vertical dimensions respectively (that is, the number of lines).

The length of the horizontal (vertical) lines is computed by the product *x-spacing* * *horizontal-count* (*y-spacing* * *vertical-count*).

width and *height* are used only when *horizontal-count* or *vertical-count* respectively is **`nil`**, to compute the value of *horizontal-count* or *vertical-count*, by truncating *width* or *height* by *x-spacing* or *y-spacing*.

major-x-step and *major-y-step* specify that each *major-x-step*'th (horizontally) or *major-y-step*'th (vertically) line is "major", which means drawn with (potentially) different thickness and color (see below).

thickness, vertical-thickness, minor-thickness, minor-vertical-thickness, left-thickness, right-thickness, top-thickness amd *bottom-thickness* specify the thickness of the lines. *color, vertical-color, major-color,major-vertical-color, left-color, right-color, top-color* and *bottom-color* specify the color of the lines. The default values for these arguments are shown in **Default values for **-thickness* and **-color* arguments to `generate-grid-lines`**:

Default values for **-thickness* and **-color* arguments to **`generate-grid-lines`**

Argument	Default value
thickness	1
vertical-thickness	*thickness*
major-thickness	*thickness*
major-vertical-thickness	*major-thickness*
top-thickness	*major-thickness*
bottom-thickness	*major-thickness*
left-thickness	*major-vertical-thickness*
right-thickness	*major-vertical-thickness*
color	**`:gray`**
vertical-color	*color*
major-color	*color*
major-vertical-color	*major-color*
top-color	*major-color*
bottom-color	*major-color*
left-color	*major-vertical-color*
right-color	*major-vertical-color*

The *top-*, bottom-*, left-*, right-** variables specify the values for the outer lines of the grid. The *major-** variables specify the values for the major lines, the other variables specify the values for the ordinary lines. The *vertical-** variables specify the values for the vertical lines, the other variables for the horizontal.

Notes

1. To actually be displayed, the result of **generate-grid-lines** must be in a hierarchy which is rooted in an **objects-displayer** or a **pinboard-objects-displayer**.
2. The result of **generate-grid-lines** is a list of **drawing-object**, so it is a valid "drawing-object-spec". It will be typically be grouped together with some other "drawing-object-specs", for example labels for the graph, by simply listing them, and then positioned and fitted by passing it to **position-object** or **fit-object** or **position-and-fit-object**.
3. The function **generate-labels** is intended to be useful to generate the labels.
4. *x-offset* and *y-offset* are useful for leaving space for the labels.
5. The units of the numbers that in the location of the lines are abstract, not pixels, and will typically correspond to the units of the data that the graph displays. They will be in pixels only if there is no fitting around the graph. For example, if you make the grid from 0 to 9 in the x dimension, and then fit to *natural-width* 10, that is you pass the result, or an object that contains the result in its hierarchy, to **fit-object** with the *natural-width* 10, the graph will take 90% of the width of the **geometry-drawing-object** that **fit-object** generated, whatever that is.

See also

drawing-object
generate-graph-from-graph-spec
14.2 Higher level - drawing graphs and bar charts

generate-labels *Function*

Summary

Return the labels of a graph of a function.

Package

lw-gt

Signature

generate-labels *horizontal-p start step range* **&key** *print-function decimal-point color x-adjust y-adjust absolute-p* **=>** *labels*

Arguments

horizontal-p⇓	A boolean.
start⇓	A real number.
step⇓	A real number.
range⇓	A positive real number.
print-function⇓	**nil**, or a function of one argument which takes a real and returns a string.
decimal-point⇓	An integer or **nil**.
color⇓	A color specification in the Color system.
x-adjust⇓, *y-adjust*⇓	**nil**, a number, or one of the keywords **:center** and **:end-align**.

absolute-p⇓ A boolean.

Values

labels⇓ A list of **drawing-object**s.

Description

The function **generate-labels** returns a list *labels* of **drawing-object**s, which are supposed to be the labels of a graph of a function.

generate-labels generates a list of drawing objects, which draw strings representing numbers and positioned in regular intervals in one dimension and fixed value in the other dimension.

horizontal-p specifies the dimension. When *horizontal-p* is true, the objects are placed in a row with regular horizontal intervals, otherwise they are spaced in a column with regular vertical intervals.

start determines the lowest value, *range* determines the range of values, and *step* determines the distance between neighbouring values. When *step* is negative, *start* is on the right (or top) and the values increase from right to left (or top to bottom).

For each value, **generate-labels** generates a string. If *print-function* is a function, it is called with the value and must return the string. Otherwise **generate-labels** makes the string using *decimal-point* and the value as follows:

```
(format nil "~,vf" decimal-point value)
```

It then uses **make-draw-string** to generate a **drawing-object**, adjusting the position by *x-adjust* horizontally and *y-adjust* vertically and using *color* as the foreground color and make it "absolute mode" depending on *absolute-p*. It then positions the object (using **position-object**) at the right place. The default value of *x-adjust* is **:center** if *horizontal-p* is true, and **:end-align** otherwise. The default value of *y-adjust* is -1 if *horizontal-p* is true, and **:center** otherwise. The default value of *color* is **:black**.

generate-labels returns a list of **drawing-object**s, which is a valid "drawing-object-spec".

Notes

1. **generate-labels** will typically be used in conjunction with **generate-grid-lines**.
2. **generate-labels** is quite a simple function. If it does not do what you want, you can improve it easily by writing your own version.
3. The defaults for *x-adjust* and *y-adjust* are what you typically use when the labels are at the left and bottom of the graph. To put the labels somewhere else in the graph, use **position-object** on *labels* to move it around. If you want the labels at the top, change *y-adjust* to 0 when passing *horizontal-p* true (so the labels are above the line), and then use **position-object** with *bottom-margin* the height of the grid to move the whole row of labels:

   ```
   (position-object (generate-labels ... :y-adjust 0)
                    :bottom-margin grid-height)
   ```

   To move the column to the right, change *x-adjust* to **nil** and use *left-margin*.
4. The size on the screen would normally be scaled by using **fit-object** on the result.

See also

fit-object

position-object
generate-grid-lines
drawing-object
14.2 Higher level - drawing graphs and bar charts

geometry-drawing-object *Class*

Summary

A **drawing-object** which when drawn changes the geometry of the drawing.

Package

lw-gt

Superclasses

compound-drawing-object

Description

The class **geometry-drawing-object** is a **drawing-object** which when drawn changes the geometry of the drawing by establishing a Graphics Ports transformation, and then draws the *sub-object* (slot inherited from **compound-drawing-object**) in this context.

See also

compound-drawing-object

make-a-drawing-call
make-draw-arc
make-draw-circle
make-draw-ellipse
make-draw-line
make-draw-lines
make-draw-polygon
make-draw-rectangle *Functions*

Summary

Create and return an **apply-drawing-object**.

Package

lw-gt

Signatures

make-a-drawing-call *function arguments* **&optional** *pass-pane-p* **=>** *apply-drawing-object*

make-draw-arc *x y width height start-angle sweep-angle* **&rest** *args* **=>** *apply-drawing-object*

make-draw-circle *x y radius* **&rest** *args* **=>** *apply-drawing-object*

make-draw-ellipse *x y x-radius y-radius* **&rest** *args* **=>** *apply-drawing-object*

make-draw-line *from-x from-y to-x to-y* **&rest** *args* **=>** *apply-drawing-object*

make-draw-lines *lines* **&rest** *args* **=>** *apply-drawing-object*

make-draw-polygon *points* **&rest** *args* **=>** *apply-drawing-object*

make-draw-rectangle *x y width height* **&rest** *args* **=>** *apply-drawing-object*

Arguments

function⇓ A function designator.

arguments⇓ A list.

pass-pane-p⇓ A generalized boolean.

x⇓, *y*⇓, *width*⇓, *height*⇓, *start-angle*⇓, *sweep-angle*⇓

Real nunbers.

args⇓ Other drawing function arguments.

radius⇓, *x-radius*⇓, *y-radius*⇓

Real numbers.

from-x⇓, *from-y*⇓, *to-x*⇓, *to-y*⇓

Real numbers.

lines⇓ A sequence of real numbers of the form *x1 y1 x2 y2*.

points⇓ A sequence of real numbers of the form *x y*.

Values

apply-drawing-object An **apply-drawing-object**.

Description

Each of the functions **make-a-drawing-call**, **make-draw-line**, **make-draw-lines**, **make-draw-polygon**, **make-draw-ellipse**, **make-draw-circle**, **make-draw-rectangle** and **make-draw-arc** creates and returns an **apply-drawing-object**.

For **make-a-drawing-call**, the drawing is done by applying the function *function* to *arguments*. When *pass-pane-p* is true, *function* is applied to the "root pane" (see **drawing-object**) followed by *arguments*. *function* should typically draw something, but it does not have to, and may do other things. The default value of *pass-pane-p* is true.

For the other functions, the drawing is done using the corresponding Graphics Ports function:

make-draw-arc **draw-arc**

make-draw-circle **draw-circle**

make-draw-ellipse **draw-ellipse**

make-draw-line **draw-line**

make-draw-lines **draw-lines**

make-draw-polygon **draw-polygon**

make-draw-rectangle **draw-rectangle**

x, *y*, *width*, *height*, *start-angle*, *sweep-angle*, *args*, *radius*, *x-radius*, *y-radius*, *from-x*, *from-y*, *to-x*, *to-y*, *lines* and *points* are interpreted as for the corresponding Graphics Ports function (except that *y* is interpreted from the bottom, see below).

Once created, the drawing object can be used in the *drawing-object* slot of an **objects-displayer** or a **pinboard-objects-displayer**, but more commonly it would be passed to one of the positioning/fitting functions (**position-object**, **fit-object** and so on), which will position and scale it with, by drawing the object inside a context of Graphics Ports transformation.

At the top level, the y coordinate is reversed, so *y* is measured from the bottom of the **objects-displayer** or **pinboard-objects-displayer**, as opposed to the default for Graphics Ports which is from the top down. A fitting object in the hierarchy may change that.

apply-drawing-objects can be used repeatedly and concurrently in the same or different panes. The ones that are created by the **make-draw-*** functions are fixed, but for objects created by **make-a-drawing-call**, the supplied function may depend on values that change, and hence needs to be redisplayed when these values change. Use **force-objects-redraw** on the root of the hierarchy (an **objects-displayer** or a **pinboard-objects-displayer**) to do that.

See **drawing-object** for description of the drawing operation.

See also

objects-displayer
pinboard-objects-displayer
position-object
fit-object
position-and-fit-object
14.1 Lower level - drawing objects and objects displayers

make-basic-graph-spec
basic-graph-spec-p
copy-basic-graph-spec
generate-graph-from-graph-spec *Functions*

Summary

Create a **basic-graph-spec** object.

Package

lw-gt

Signatures

make-basic-graph-spec *function start-x step-x range* **&key** *color thickness name x-offset y-offset x-scale y-scale var1 var2 var3 var4 var5 var6* **=>** *basic-graph-spec*

basic-graph-spec-p *object* **=>** *boolean*

copy-basic-graph-spec *basic-graph-spec* **=>** *basic-graph-spec*

generate-graph-from-graph-spec *basic-graph-spec* **=>** *drawing-object*

Arguments

function⇓ A function of two arguments x and y.

start-x⇓, *step-x*⇓, *range*⇓
Real numbers.

color⇓ A color specification in the Color system.

thickness⇓ A positive real numbers.

name⇓ A Lisp object.

x-offset⇓, *y-offset*⇓, *x-scale*⇓, *y-scale*⇓
Real numbers.

var1⇓, *var2*⇓, *var3*⇓, *var4*⇓, *var5*⇓, *var6*⇓
Lisp objects.

object⇓ A Lisp object.

basic-graph-spec⇓ A **basic-graph-spec** object.

Values

basic-graph-spec A **basic-graph-spec** object.

boolean A boolean.

drawing-object A **drawing-object**.

Description

The function **make-basic-graph-spec** creates a **basic-graph-spec** object. This object can be modified by the **basic-graph-spec-*** accessors. The function **generate-graph-from-graph-spec** generates the graph using the current values in the **basic-graph-spec** object, which is a **drawing-object** which when drawn draws the graph, which means drawing a line between each two successive points.

function must be a function of two arguments: the **basic-graph-spec** and the x value. It needs to return the corresponding y value.

start-x, *step-x* and *range* define which x values to use: the first value is *start-x*, and then increase by *step-x* until the x is greater than (+ *start-x range*). For each x value, **generate-graph-from-graph-spec** calls *function* with *basic-graph-spec* and the x value to generate the y value.

x-scale and *y-scale* (default to 1) are used to scale the x and y after calling *function*, by multiplying the x and y by *x-scale* and *y-scale* respectively.

x-offset and *y-offset* (default to 0) are used to translate the scaled values of x and y by adding *x-offset* and *y-offset* to the scaled x and y.

The scaled and transformed pair x, y define a point. **generate-graph-from-graph-spec** then generates a **drawing-object** that draws a line between each two successive points.

thickness and *color* specify the thickness and the color of the lines. The lines are drawn with *scale-thickness* **nil**.

name, *var1*, *var2*, *var3*, *var4*, *var5* AND *var6* are arbitrary values, which you can use to store anything that the function needs to compute the y value. The system does not read or write them.

The function **copy-basic-graph-spec** can be used to copy a *basic-graph-spec*.

The fimctopm **basic-graph-spec-p** is the predicate, which returns true if *object* is a **basic-graph-spec** and false otherwise.

See also

basic-graph-spec
generate-graph-from-pairs
drawing-object
14.2 Higher level - drawing graphs and bar charts

make-draw-string *Function*

Summary

Creates a **string-drawing-object**.

Package

lw-gt

Signature

make-draw-string *string font-descriptor* **&rest** *arguments* **&key** *x-adjust y-adjust absolute* **&allow-other-keys =>** *string-drawing-object*

Arguments

string⇓	A string.
font-descriptor⇓	A **font-description** object, an integer or **nil**.
arguments⇓	Other keyword arguments for **draw-string**.
x-adjust⇓, *y-adjust*⇓	One of the keywords **:end-align** and **:center**, or a number.
absolute⇓	A generalized boolean.

Values

string-drawing-object	A **string-drawing-object**.

Description

The function **make-draw-string** creates a **string-drawing-object**, which draws the string using **draw-string**.

string is the string to draw.

font-descriptor can be a **font-description** specifying the font to use. It can also be an integer specifying the size only, which is equivalent to:

```
(gp:make-font-description :size font-descriptor)
```

font-descriptor can also be **`nil`** meaning using the default font of the root pane.

When *absolute* is non-nil, the string is drawn in "absolute mode", which means ignoring scaling and rotation. The default value of *absolute* is **`nil`**.

x-adjust and *y-adjust* specify adjustment to the position of the string. The adjustments are done independently vertically and horizontally. The drawing point is the left/corner of the current geometry (inherited from the parent). If *x-adjust* and *y-adjust* are not supplied, the string is drawn at the drawing point. Note that this means that the descent part is below this point. If *x-adjust* and/or *y-adjust* are supplied, they can be one of:

`:end-align` Align the "end" (right side or top) of the string with the drawing point.

`:center` Align the center of the string with the drawing point.

A number Multiply by the average width (*x-adjust*) or height (*y-adjust*) of the font and add to the drawing point.

Any other value of *x-adjust* or *y-adjust* is regarded as no adjustment. Adjustments are applied in the same scope as drawing the string, which means they are scaled or not depending on the value *absolute*. However, the y direction still increases upwards when computing the y adjustment.

arguments can also contain all the keyword arguments that **`draw-string`** takes, but **`:font`** is overridden by *font-descriptor*.

See **`drawing-object`** about the drawing operation and the meaning of "parent" and "root pane".

See also

`drawing-object`
14.1 Lower level - drawing objects and objects displayers

make-pinboard-objects-displayer *Function*

Summary

Creates a **`pinboard-objects-displayer`**.

Package

`lw-gt`

Signature

`make-pinboard-objects-displayer` *drawing-object* **`&rest`** *args* **`&key`** *use-metafile natural-width natural-height* **`&allow-other-keys =>`** *pinboard-objects-displayer*

Arguments

drawing-object⇓ A "drawing-object-spec".

args⇓ Initargs for **`pinboard-object`**.

use-metafile⇓ A generalized boolean.

natural-width⇓, *natural-height*⇓

Integers.

Values

pinboard-objects-displayer

A **`pinboard-objects-displayer`**.

Description

The function **`make-pinboard-objects-displayer`** creates a **`pinboard-objects-displayer`**, which is a subclass of **`pinboard-object`**. The **`pinboard-objects-displayer`** draws the **`drawing-object`** *drawing-object*.

drawing-object must be a "drawing-object-spec", which means either an instance of (a subclass of) **`drawing-object`** or a list of "drawing-object-specs".

use-metafile specifies whether to use an internal metafile. When *use-metafile* is true the **`pinboard-objects-displayer`** draws the objects to a metafile, and then draws the metafile to the screen. *natural-width* and *natural-height* determine the size of the metafile to use. They are ignored if *use-metafile* is false. The default value of *use-metafile* is **`t`**.

The default value of *natural-width* x *natural-height* is 800 x 600.

args can contain all the initargs of **`pinboard-object`**. In particular, all the geometry initargs can be used to define the initial geometry. The geometry can be changed later by **`(setf capi:static-layout-child-geometry)`** and the related functions.

See also

`drawing-object`
`objects-displayer`
`pinboard-objects-displayer`
14.1 Lower level - drawing objects and objects displayers

objects-displayer *Class*

Summary

A subclass of **`pinboard-layout`**, which adds displaying of hierarchial objects.

Package

`lw-gt`

Superclasses

`pinboard-layout`

Initargs

`:drawing-object`	A **`drawing-object`** or a list (see Description below).
`:use-metafile`	A generalized boolean.
`:natural-width`	Integers.

Accessors

`objects-displayer-objects`

Description

The class **`objects-displayer`** is a subclass of **`pinboard-layout`** that in addition to **`pinboard-object`**s can also have "drawing objects" which contain hierarchies of graphics. These objects are created by the **`make-draw-*`** functions and the positioning functions (**`position-and-fit-object`**, **`position-object`**, **`fit-object`**). An **`objects-displayer`** can also have in its *description* **`pinboard-objects-displayer`**s, which can also contain hierarchies of drawings.

drawing-object is either a "drawing-object-spec", which is an instance of a subclass of **`drawing-object`**, or a list of "drawing-object-specs". The value can be modified later by **`(setf objects-displayer-drawing-object)`**. The drawing objects in the *objects* slot are displayed after any **`pinboard-object`**s in the *layout-description* of *pane* (if any) are displayed. If it is a list, they are displayed according to the order in the list. This is implemented via a *display-callback*, so you cannot use **`:display-callback`** in an **`objects-displayer`**.

Objects which are the result of the positioning functions are being positioned and scaled again when the **`objects-displayer`** is resized, before being displayed.

use-metafile specifies whether the drawing of the objects should be done via a metafile. When using a metafile, the objects are first drawn to an internal metafile, which is then drawn to the pane. The result is another scaling (between the size of the metafile and the size of pane). Note that means that objects that are drawn in their "absolute" size (not inside a fitting object, or explicitly absolute) are resized at that stage. Drawing via a metafile makes resizing better and faster.

When *use-metafile* is true, *natural-width* and *natural-height* define the size of the metafile to create in pixels. For objects that are supposed to be drawn in their absolute size, that will affect how much they are actually resized. The default value of *use-metafile* is true. The default value of *natural-width* x *natural-height* is 800 x 600.

Objects in the *drawing-object* list or inside the hierarchy inside any of these objects may change, which may require redisplaying it. The function **`force-objects-redraw`** can be used to force redrawing all the objects.

Notes

The drawing via the metafile is applicable only to the drawing objects, not to the **`pinboard-object`**s in the *layout-description* of the *pane*.

See also

`position-object`
`fit-object`
`position-and-fit-object`
`make-draw-line`
`make-draw-lines`
`make-draw-arc`
`make-draw-polygon`
`make-draw-ellipse`
`make-draw-circle`
`make-draw-rectangle`
`force-objects-redraw`
14.1 Lower level - drawing objects and objects displayers

pinboard-objects-displayer *Class*

Summary

A **`pinboard-object`** which draws its *drawing-object*.

Package

lw-gt

Superclasses

pinboard-object

Accessors

pinboard-objects-displayer-objects

Description

The class **pinboard-objects-displayer** draws its *drawing-object*.

Like other **pinboard-object**s, to be displayed a **pinboard-objects-displayer** needs to be added to the *description* of a **pinboard-layout**, using the standard CAPI interface of **pinboard-layout**, that is **:description** passed to **cl:make-instance**, **(setf capi:layout-description)**, or **manipulate-pinboard**.

When displayed, a **pinboard-objects-displayer** draws its *drawing-object*. If it was created with *use-metafile* **t** (see **make-pinboard-objects-displayer**), it draws to a metafile of the size indicated by *natural-width* and *natural-height*, and then draws the metafile to the screen using its own geometry as the target rectangle. Otherwise it may draw to the screen or use a pixmap cache.

The *drawing-object* in the **pinboard-objects-displayer** can be changed by **(setf pinboard-objects-displayer-drawing-object)**, which automatically forces it to be redisplayed. If any of the objects inside the hierarchy below the *drawing-object* changes, there is no forced redisplay. You need to use **force-objects-redraw** on the **pinboard-objects-displayer** (or the parent **objects-displayer**) to redisplay.

See also

make-pinboard-objects-displayer

string-drawing-object *Class*

Summary

A **drawing-object** which draws its string.

Package

lw-gt

Superclasses

drawing-object

Description

The class **string-drawing-object** draws its string. Instances are created by **make-draw-string**. See **make-draw-string** for the details.

string-drawing-object objects can be used repeatedly and concurrently in the same or different panes.

See also

make-draw-string

24 COLOR Reference Entries

This chapter describes symbols available in the **color** package.

apropos-color-alias-names *Function*

Summary

Returns color aliases containing a given string.

Package

color

Signature

apropos-color-alias-names *substring* **=>** *list*

Arguments

substring⇓ A string.

Values

list A list of symbols.

Description

The function **apropos-color-alias-names** returns a list of symbols whose symbol-names contain *substring* and which are defined as aliases in the color-database defining color aliases. By convention these are in the keyword package.

Examples

In this example, a color alias is defined for the color **indianred1**. **apropos-color-alias-names** only returns this alias, rather than both the alias and the original color, despite the similarity in the names.

```
CL-USER 8 > (color:define-color-alias :myindianred1
                              :indianred1)
(#S(COLOR-ALIAS COLOR :INDIANRED1))

CL-USER 9 > (color:apropos-color-names "INDIANRED1")
(:INDIANRED1 :MYINDIANRED1)

CL-USER 10 > (color:apropos-color-alias-names "INDIANRED1")
(:MYINDIANRED1)

CL-USER 11 >
```

See also

apropos-color-names
apropos-color-spec-names
get-all-color-names
15 The Color System

apropos-color-names *Function*

Summary

Returns colors and color aliases containing a given string.

Package

color

Signature

apropos-color-names *substring* **=>** *list*

Arguments

substring⇓	A string.

Values

list	A list of symbols.

Description

The function **apropos-color-names** returns a list of symbols whose symbol-names contain *substring* and which are present in the color-database defining color aliases. By convention these are in the keyword package.

Examples

```
COLOR-4> (color:apropos-color-names "RED")
 (:ORANGERED3 :ORANGERED1 :INDIANRED3 :INDIANRED1
  :PALEVIOLETRED :RED :INDIANRED :INDIANRED2
  :INDIANRED4 :ORANGERED :MEDIUMVIOLETRED
  :VIOLETRED :ORANGERED2 :ORANGERED4 :RED1 :RED2 :RED3
  :RED4 :PALEVIOLETRED1 :PALEVIOLETRED2 :PALEVIOLETRED3
  :PALEVIOLETRED4 :VIOLETRED3 :VIOLETRED1 :VIOLETRED2
  :VIOLETRED4)
```

See also

apropos-color-alias-names
apropos-color-spec-names
get-all-color-names
15 The Color System

apropos-color-spec-names *Function*

Summary

Returns colors containing a given string.

Package

color

Signature

apropos-color-spec-names *substring* **=>** *list*

Arguments

substring⇓	A string.

Values

list	A list of symbols.

Description

The function **apropos-color-spec-names** returns a list of symbols whose symbol-names contain *substring* and which are defined as original entries in the color-database defining color aliases. By convention these are in the keyword package.

Examples

```
CL-USER 14 > (color:define-color-alias :mygray100 :gray100)
(#S(COLOR-ALIAS COLOR :GRAY100))

CL-USER 15 > (color:apropos-color-names "GRAY100")
(:MYGRAY100 :GRAY100)

CL-USER 16 > (color:apropos-color-spec-names "GRAY100")
(:GRAY100)

CL-USER 17 >
```

See also

apropos-color-alias-names
apropos-color-names
get-all-color-names
15 The Color System

color-alpha *Function*

Summary

Returns the alpha component of a color specification.

Package

`color`

Signature

`color-alpha` *color-spec* **`&optional`** *default* **`=>`** *alpha*

Arguments

color-spec⇓	A color specification.
default⇓	A number between 0 and 1.

Values

alpha	The alpha component of *color-spec*.

Description

The function **`color-alpha`** returns the alpha component on *color-spec*, which is a color specification in any model.

If *color-spec* does not have an alpha component, then *default* is returned.

The default value of *default* is 1.0.

See also

`make-hsv`
`make-rgb`
`make-gray`

color-blue
color-green
color-red
color-hue
color-saturation
color-value *Functions*

Summary

Returns the associated component of a color specification.

Package

color

Signatures

color-blue *color-spec* **=>** *color-component*

color-green *color-spec* **=>** *color-component*

color-red *color-spec* **=>** *color-component*

color-hue *color-spec* **=>** *color-component*

color-saturation *color-spec* **=>** *color-component*

color-value *color-spec* **=>** *color-component*

Arguments

color-spec⇓	A color specification.

Values

color-component	A color component from the appropriate color model.

Description

These functions return the specified component of *color-spec*.

If *color-spec* is not from the appropriate color model (**:rgb** in the case of **color-red**, **color-green** and **color-blue**, and **:hsv** in the case of **color-hue**, **color-saturation** and **color-value**) then the component is calculated.

Examples

```
CL-USER 31 > (color:make-rgb 1.0s0 0.0s0 0.0s0)
#(:RGB 1.0S0 0.0S0 0.0S0)

CL-USER 32 > (color:color-red *)
1.0S0

CL-USER 33 > (color:color-green **)
0.0S0

CL-USER 34 > (color:color-value ***)
1.0S0

CL-USER 35 >
```

See also

make-hsv
make-rgb
make-gray
color-model
color-level

color-database *Variable*

Summary

The current color-database.

Package

`color`

Initial Value

The colors are in the file **`config/colors.db`**.

Description

The variable **`*color-database*`** is the current color-database.

Examples

To replace the current color database with a new one, do the following:

```
(setf color:*color-database* (color:make-color-db))
```

See also

`delete-color-translation`
`read-color-db`
`load-color-database`
15.4 Loading the color database

color-from-premultiplied *Function*

Summary

Transforms a color to its un-premultiplied version.

Package

`color`

Signature

`color-from-premultiplied` *color* **`=>`** *result*

Arguments

color⇓ A color-spec.

Values

result A color-spec.

Description

The function **`color-from-premultiplied`** transforms a color, which is assumed to be premultiplied, to its un-premultiplied version.

color should be a color-spec (see **15.1 Color specs**).

If *color* is RGB with alpha it is transformed to its RGB un-premultiplied version. Otherwise *color* is returned without a change.

Notes

You get premultiplied colors when using Image Access, either by unconverting (using **`unconvert-color`**) the result of **`image-access-pixel`**, or by reading the values from the vector that is filled by **`image-access-pixels-from-bgra`**.

See also

`color-to-premultiplied`
`image-access-pixel`
`image-access-pixels-to-bgra`
`image-access-pixels-from-bgra`
13.10.8 Image access

color-level *Function*

Summary

Returns the gray level of a color specification.

Package

`color`

Signature

`color-level` *color-spec* **`=>`** *gray-level*

Arguments

color-spec⇓ A color specification.

Values

gray-level Color component from the **`:gray`** model.

Description

The function **`color-level`** return the gray level of *color-spec*. If *color-spec* is not from the **`:gray`** model, the component is calculated.

Examples

```
CL-USER 2 > (color:make-gray 0.66667s0)
#(:GRAY 0.66667S0)

CL-USER 3 > (color:color-level *)
0.66667S0

CL-USER 4 >
```

See also

make-hsv
make-rgb
make-gray
color-model
color-blue
15.3 Color models

color-model *Function*

Summary

Returns the model of a color-spec.

Package

color

Signature

color-model *color-spec* **=>** *color-model*

Arguments

color-spec⇓	A color specification.

Values

color-model	**:gray**, **:rgb**, or **:hsv**.

Description

The function **color-model** returns the model of *color-spec*.

Examples

```
CL-USER 29 > (color:make-gray 0.66667s0)
#(:GRAY 0.66667S0)

CL-USER 30 > (color:color-model *)
:GRAY

CL-USER 31 >
```

See also

`make-hsv`
`make-rgb`
`make-gray`
`color-blue`
`color-level`
15.1 Color specs

colors= *Function*

Summary

Tests to see if two colors are equal.

Package

`color`

Signature

`colors=` *color1 color2* **`&optional`** *tolerance* **`=>`** *bool*

Arguments

color1⇓	A color specification.
color2⇓	A color specification.
tolerance⇓	A tolerance level within which *color1* and *color2* may vary. The default value is **`0.001s0`**.

Values

bool	**`t`** if the two colors are equal within the given tolerance, **`nil`** otherwise.

Description

The function **`colors=`** return **`t`** if *color1* and *color2* are equal, within the tolerance *tolerance*.

See also

`ensure-color`
`ensure-rgb`
`convert-color`
15 The Color System

color-to-premultiplied *Function*

Summary

Transform a color to its premultiplied version.

Package

`color`

Signature

`color-to-premultiplied` *color* **`=>`** *result*

Arguments

color⇓	A color-spec.

Values

result	A color-spec.

Description

The function **`color-to-premultiplied`** transforms a color to its premultiplied version, which is needed when modifying images using Image Access.

color must be a color-spec, such as the result of a call to **`make-rgb`** (see **15.1 Color specs**).

If *color* does not have an alpha component, it is returned without a change. If it does have alpha, it is transformed to RGB if needed, and premultiplied, returning a premultiplied RGB color.

Notes

You need to premultiply when setting pixels using Image Access in an image with alpha. The result is unconverted, so when using **`image-access-pixel`** it still needs to be converted (by **`convert-color`**).

See also

`color-from-premultiplied`
`image-access-pixel`
`image-access-pixels-to-bgra`
`image-access-pixels-from-bgra`
13.10.8 Image access

color-with-alpha *Function*

Summary

Adds a specified alpha component to a color.

Package

`color`

Signature

`color-with-alpha` *color alpha* **`=>`** *color-spec*

Arguments

color⇓	A color specification.
alpha⇓	A real in the inclusive range [0,1].

Values

color-spec⇓	A color specification, or **`nil`**.

Description

The function **`color-with-alpha`** returns a color like the argument *color* but with alpha component *alpha*.

color needs to be a color specification, either a keyword naming a color (a member of the result of calling **`get-all-color-names`**), or a color-spec (for example the result of **`make-rgb`**).

alpha must be a real in the inclusive range [0,1], otherwise an error is signaled. *alpha* = 0 means *color-spec* is transparent, *alpha* = 1 means it is solid.

`color-with-alpha` returns a color-spec, or **`nil`** if *color* is not recognized.

See also

`get-all-color-names`
`make-rgb`
15.1 Color specs

convert-color *Function*

Summary

Return the representation of a color specification on a given graphics port.

Package

`color`

Signature

convert-color *port color* **&key** *errorp* **=>** *color-rep*

Arguments

port⇓	A graphics port.
color⇓	A color specification.
errorp⇓	A generalize boolean.

Values

color-rep⇓	Representation of *color* on *port*.

Description

The function **convert-color** returns the representation of *color* on the given graphics port *port*.

If *errorp* is **t** (the default), then **convert-color** checks for errors. Otherwise **nil** might be returned.

Notes

color-rep might be a "pixel" value, which corresponds to an index into the default colormap. It is more efficient to use the result of **convert-color** in place of its argument in drawing function calls, but the penalty is the risk of erroneous colors being displayed should the colormap or the colormap entry be changed.

See also

colors=
ensure-color
ensure-rgb
unconvert-color
13.10.8 Image access
15 The Color System

define-color-alias *Function*

Summary

Lets you define an alias for a color specification or alias.

Package

color

Signature

define-color-alias *name color* **&optional** *if-exists* **=>** *name*

Arguments

name⇓	The name of the new alias.

color⇓ A color specification for the new alias.

if-exists⇓ One of **`:replace`**, **`:error`** or **`:ignore`**.

Values

name The name of the new alias.

Description

The function **`define-color-alias`** defines *name* to be a color alias for *color*, which may be another color alias or a color spec.

When *color* is a color spec rather than another color name, the entry is better described as a "color translation" rather than a "color alias". In particular, calling **`get-color-alias-translation`** on *name* will just return *name*. **`get-color-spec`** with *name* will return *color*.

if-exists controls what happens in *name* is already a known alias:

`:replace` Replace any existing alias.

`:error` Raise an error if alias is already defined.

`:ignore` Ignore redefinition of an alias.

if-exists defaulrs to **`:replace`**.

Examples: 1

```
CL-USER 16 > (color:define-color-alias :mygray :darkslategray)
:mygray

CL-USER 17 > (color:define-color-alias :mygray :darkslategray
                                       :error)

Error: :MYGRAY names an existing alias for #(:RGB 0.1843133S0 0.309803S0 0.309803S0)
  1 (continue) Replace :MYGRAY with the alias :DARKSLATEGRAY
  2 Continue, without redefining alias :MYGRAY
  3 Try a new name for the alias, instead of :MYGRAY
  4 (abort) Return to level 0.
  5 Return to top loop level 0.
  6 Destroy process.

Type :c followed by a number to proceed or type :? for other options

CL-USER 18 : 1 >
```

Examples: 2

```
CL-USER 19 > (color:define-color-alias :lispworks-blue
                (color:make-rgb 0.70s0 0.90s0 0.99s0))
:lispworks-blue

CL-USER 20 >
```

See also

`get-color-alias-translation`
`get-color-spec`

15 The Color System

define-color-models *Macro*

Summary

Defines **all** the color models.

Package

`color`

Signature

`define-color-models` *model-descriptors* **`=>`** *color-models*

Arguments

model-descriptors⇓ A list, each element being a model-descriptor.

Values

color-models The color models defined.

Description

The macro **`define-color-models`** defines the color models in *model-descriptors*.

A model descriptor has the syntax:

(*model-name component-descr****)**

A *component-descr* is a list:

(*component-name lowest-value highest-value***)**

The default color models are defined by the following form:

```
(color:define-color-models ((:rgb (red 0.0 1.0)
                                  (green 0.0 1.0)
                                  (blue 0.0 1.0))
                            (:hsv (hue 0.0 5.99999)
                                  (saturation 0.0 1.0)
                                  (value 0.0 1.0))
                            (:gray (level 0.0 1.0))))
```

If you want to keep existing color models, add your new ones to this list: only one **`define-color-models`** form is recognized. The form should be compiled.

Examples

To replace the HSV color model with a CMYK model, while retaining the other color models:

```
(define-color-models ((:rgb (red 0.0 1.0)
```

```
                    (green 0.0 1.0)
                    (blue 0.0 1.0))
              (:cmyk (cyan 0.0 1.0)
                     (magenta 0.0 1.0)
                     (yellow 0.0 1.0)
                     (black 0.0 1.0)
              (:gray (level 0.0 1.0))))
```

See also

15 The Color System

delete-color-translation *Function*

Summary

Removes an entry from the color-database.

Package

`color`

Signature

`delete-color-translation` *color-name*

Arguments

color-name⇓	A defined color spec or alias.

Description

The function **`delete-color-translation`** removes the entry for *color-name* from the curent color-database. Both original entries and aliases can be removed.

See also

`load-color-database`
`*color-database*`
`read-color-db`
15 The Color System

ensure-color *Function*

Summary

Return a color specification in the model of a supplied color spec.

Package

`color`

Signature

ensure-color *color-spec* *match-color-spec* **=>** *result*

Arguments

color-spec⇓	A color specification.
match-color-spec⇓	A color specification.

Values

result⇓	A color specification.

Description

The function **ensure-color** returns a color specification for *color-spec*, in the color model of *match-color-spec*. This allows you to convert color specifications from one model to another with having to explicitly state the color model.

If *color-spec* has an alpha component, then *result* has that same alpha component.

Examples

```
(ensure-color (make-rgb 1 1 0 0.75) (make-hsv 0 0 0))
=>
#(:HSV 1 1 1 0.75)
```

See also

convert-color
colors=
ensure-model-color
15 The Color System

ensure-model-color *Function*

Summary

Converts a color specification to a given model.

Package

color

Signature

ensure-model-color *color-spec* *model* **=>** *result*

Arguments

color-spec⇓	A color specification.
model⇓	A color-model (**:rgb**, **:hsv** or **:gray**).

Values

result⇓ A color specification.

Description

The function **`ensure-model-color`** returns a color specification for *color-spec* in the color model specified by *model*.

If *color-spec* has an alpha component, then *result* has that same alpha component.

Examples

```
(ensure-model-color (make-rgb 1 1 0 0.75) :hsv)
=>
#(:HSV 1 1 1 0.75)
```

See also

`convert-color`
`colors=`
`ensure-color`
`ensure-rgb`
15 The Color System

ensure-rgb
ensure-hsv
ensure-gray *Functions*

Summary

Returns a color specification for a particular model.

Package

`color`

Signatures

`ensure-rgb` *color-spec* **`=>`** *result*

`ensure-hsv` *color-spec* **`=>`** *result*

`ensure-gray` *color-spec* **`=>`** *result*

Arguments

color-spec⇓ A color specification.

Values

result⇓ A color specification.

Description

The functions **ensure-rgb**, **ensure-hsv** and **ensure-gray** each return a color specification matching the supplied *color-spec*, but in the appropriate model.

If *color-spec* is in the same model, it is just returned. Otherwise a new color specification for that model is calculated. Thus, **ensure-rgb** returns a color specification in the RGB color model, whatever color model is used in *color-spec*.

If *color-spec* has an alpha component, then *result* has that same alpha component.

Examples

```
(ensure-hsv (make-rgb 1 1 0 0.75))
=>
#(:HSV 1 1 1 0.75)


(ensure-gray (make-rgb 0 0 1 0.75))
=>
#(:GRAY 0.33333302S0 0.75)
```

See also

convert-color
colors=
ensure-color
ensure-model-color
15.3 Color models

get-all-color-names *Function*

Summary

Returns a list of all color-names in the color database.

Package

color

Signature

get-all-color-names &optional *sort* **=>** *color-names*

Arguments

sort⇓	If **t**, sort list of color names alphanumerically. By default, this is **nil**.

Values

color-names	A list of all color names in the color database.

Description

The function **get-all-color-names** returns a list of all color-names in the color database. By convention these are symbols in the keyword package. The returned list is alphanumerically sorted on the symbol-names if *sort* is non-nil.

See also

apropos-color-names
apropos-color-spec-names
apropos-color-alias-names
15 The Color System

get-color-alias-translation *Function*

Summary

Return the ultimate color name associated a color alias.

Package

color

Signature

get-color-alias-translation *color-alias* **=>** *color-name*

Arguments

color-alias⇓	A defined color alias.

Values

color-name	The color name associated with *color-alias*.

Description

The function **get-color-alias-translation** returns the ultimate color name associated with *color-alias*.

Examples

```
CL-USER 23 > (color:define-color-alias :lispworks-blue
               (color:make-rgb 0.70s0 0.90s0 0.99s0))
:lispworks-blue

CL-USER 24 > (color:define-color-alias
               :color-background :lispworks-blue)
:color-background

CL-USER 25 > (color:define-color-alias
               :listener-background :color-background)
:listener-background

CL-USER 26 > (color:get-color-alias-translation
                   :listener-background)
:LISPWORKS-BLUE

CL-USER 27 > (color:get-color-alias-translation
                   :color-background)
:LISPWORKS-BLUE
```

```
CL-USER 28 >
```

See also

define-color-alias
get-color-spec
15 The Color System

get-color-spec *Function*

Summary

Returns the color-spec for a color.

Package

color

Signature

get-color-spec *color* **=>** *color-spec*

Arguments

color⇓ A defined color specification, color alias, or an original color name.

Values

color-spec A color specification.

Description

The function **get-color-spec** returns the color-spec for *color*, which can be a color-spec, a color-alias, or an original color name.

Examples

```
CL-USER 28 > (color:define-color-alias :lispworks-blue
               (color:make-rgb 0.70s0 0.90s0 0.99s0))
(#S(COLOR-ALIAS COLOR #(:RGB 0.699999S0 0.9S0 0.99S0)))

CL-USER 29 > (color:define-color-alias
               :color-background :lispworks-blue)
(#S(COLOR-ALIAS COLOR :LISPWORKS-BLUE))

CL-USER 30 > (color:define-color-alias
               :listener-background :color-background)
(#S(COLOR-ALIAS COLOR :COLOR-BACKGROUND))

CL-USER 31 > (color:get-color-spec :listener-background)
#(:RGB 0.699999S0 0.9S0 0.99S0)

CL-USER 32 > (color:get-color-spec :color-background)
#(:RGB 0.699999S0 0.9S0 0.99S0)
```

```
CL-USER 33 > (color:get-color-spec :lispworks-blue)
#(:RGB 0.699999S0 0.9S0 0.99S0)

CL-USER 34 > (color:get-color-spec
              #(:RGB 0.70s0 0.90s0 0.99s0))
#(:RGB 0.699999S0 0.9S0 0.99S0)

CL-USER 35 >
```

See also

define-color-alias
get-color-alias-translation
15 The Color System

load-color-database *Function*

Summary

Loads a color database.

Package

color

Signature

load-color-database *data*

Arguments

data⇓	A description of a color database.

Description

The function **load-color-database** loads the color database with color definitions contained in *data*, which should have been obtained via the functions **read-color-db**. The colors thus defined may not be replaced by color aliases.

See also

color-database
delete-color-translation
read-color-db
15 The Color System

make-gray *Function*

Summary

Returns a color specification in the gray model.

Package

color

Signature

make-gray *level* **&optional** *alpha* **=>** *color-spec*

Arguments

level⇓	A color component used to define the gray level required.
alpha⇓	A number between 0 and 1, or **nil**.

Values

color-spec	A color specification.

Description

The function **make-gray** returns a color-spec in the **:gray** model with component *level*.

Note that short-floats are used for the component; this results in the most efficient color conversion process. However, any floating point number type can be used.

alpha indicates the alpha value of the color. 0 means it is transparent, 1 means it is solid. If *alpha* is **nil** or not specified then the color does not have an alpha component and it is assumed to be solid.

Examples

```
CL-USER 25 > (color:make-gray 0.66667s0)
#(:GRAY 0.66667S0)
```

See also

make-hsv
make-rgb
color-model
color-blue
color-level
color-alpha
15.1 Color specs

make-hsv *Function*

Summary

Returns a color specification in the hue-saturation-value model.

Package

color

Signature

make-hsv *hue saturation value* **&optional** *alpha* **=>** *color-spec*

Arguments

hue⇓	A hue component.
saturation⇓	A saturation component.
value⇓	A value component.
alpha⇓	A number between 0 and 1, or **nil**.

Values

color-spec	A color specification.

Description

The function **make-hsv** return a color-spec in the **:hsv** model with components *hue*, *saturation* and *value*.

Note that short-floats are used for each component; this results in the most efficient color conversion process. However, any floating-point number type can be used.

alpha indicates the alpha value of the color. 0 means it is transparent, 1 means it is solid. If *alpha* is **nil** or not specified then the color does not have an alpha component and it is assumed to be solid.

Examples

```
CL-USER 27 > (color:make-hsv 1.2s0 0.5s0 0.9s0)
#(:HSV 1.2S0 0.5S0 0.9S0)
```

See also

make-rgb
make-gray
color-model
color-blue
color-level
color-alpha
15.1 Color specs

make-rgb *Function*

Summary

Returns a color specification in the red-green-blue model.

Package

color

Signature

make-rgb *red green blue* **&optional** *alpha* **=>** *color-spec*

Arguments

red⇓	A red component.
green⇓	A green component.
blue⇓	A blue component.
alpha⇓	A number between 0 and 1, or **nil**.

Values

color-spec	A color specification.

Description

The function **make-rgb** returns a color-spec in the **:rgb** model with components *red*, *green* and *blue*.

Note that short floats are used for each component; this results in the most efficient color conversion process. However, any floating point number type can be used.

alpha indicates the alpha value of the color. 0 means it is transparent, 1 means it is solid. If *alpha* is **nil** or not specified then the color does not have an alpha component and it is assumed to be solid.

Examples

The object returned by the following call defines the color red in the RGB model:

```
CL-USER 25 > (color:make-rgb 1.0s0 0.0s0 0.0s0)
#(:RGB 1.0S0 0.0S0 0.0S0)
```

See also

make-hsv
make-gray
color-model
color-blue
color-level
color-alpha
15.1 Color specs

read-color-db *Function*

Summary

Reads the color definitions contained in a file.

Package

color

Signature

read-color-db &optional *file* **=>** *color-database*

Arguments

file⇓ A filename or pathname containing the color definitions to be read.

Values

color-database⇓ A database definition.

Description

The function **read-color-db** reads color definitions from *file*. *file* defaults to the default color definitions file in the LispWorks library.

The format of the file is:

```
#(:RGB 1.0s0 0.980391s0 0.980391s0)     snow
#(:RGB 0.972548s0 0.972548s0 1.0s0)     GhostWhite
...
```

Each line contains a color definition which consists of a color-spec and a name. The names are converted to uppercase and interned in the keyword package. Whitespace in names is preserved.

color-database can be passed to **load-color-database**.

See also

load-color-database
color-database
delete-color-translation
15 The Color System

unconvert-color *Function*

Summary

Returns a color specification for a color representation.

Package

color

Signature

unconvert-color *port* *color-rep* **=>** *color*

Arguments

port⇓ A graphics port.

color-rep⇓ A color representation on *port*.

Values

color A color specification.

Description

The function **`unconvert-color`** returns a color specification corresponding to the color representation *color-rep* on the Graphics Port *port*.

If *color-rep* is a color specification, a symbol or a color alias, then it is simply returned since the color system can interpret these directly.

Otherwise *color-rep* is assumed to be a color representation on *port*, like those returned by **`convert-color`** and **`image-access-pixel`**, and a corresponding RGB value is returned.

See also

`convert-color`
`image-access-pixel`
13.10.8 Image access

Index

A

C

F

I

M

N

O

P

Q

R

W

Y

Z

Numerics

Non-alaphanumerics