LispWorks® for Macintosh

CAPI User Guide

Version 6.1

Copyright and Trademarks

Address

LispWorks Ltd
St. John’s Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839 (toll-free)

From elsewhere: +44 1223 421860

Fax

From North America: 617 812 8283

From elsewhere: +44 870 2206189

www.lispworks.com

Contents

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.

Assumptions

The CAPI documentation assumes that you are familiar with:

- LispWorks
- Common Lisp and CLOS, the Common Lisp Object System
- Mac OS X.

Illustrations in this manual show the CAPI running on Mac OS X 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, is numbered and printed like this.

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

A Description of the Contents

This guide 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.

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

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

Chapter 4, *General Considerations*, covers some general issues that you should be aware of when using CAPI, including information about the host window system.

Chapter 3, *Creating Common Windows*, introduces more of the fundamental CAPI elements and common themes. These elements are explained in greater detail in the remainder of the manual.

Chapter 6, *Choices*, 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.

Chapter 7, *Laying Out CAPI Panes* introduces the idea of *layouts*. These let you combine different CAPI elements inside a single window.

Chapter 8, *Modifying CAPI Windows*, outlines basic techniques for modifying existing windows.

Chapter 9, *Creating Menus*, shows you how to add menus to a window.

Chapter 10, *Defining Interface Classes*, introduces the macro **`define-interface`**. This macro can be used to define interface classes composed of CAPI elements — either the predefined elements explained elsewhere in this manual or your own.

Chapter 11, *Prompting for Input*, discusses the ways in which dialog boxes may be used to prompt the user for input.

Chapter 12, *Creating Your Own Panes*, shows you how you can define your own classes when those provided by the CAPI are not sufficient for your needs.

Chapter 13, *Graphics Ports*, provides information on the Graphics Ports package, which provides a selection of drawing and image tranformation 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 and the *LispWorks CAPI Reference Manual*.

Chapter 14, *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.

Chapter 15, *Printing from the CAPI—the Hardcopy API*, describes the programmatic printing of Graphics Ports.

Chapter 16, *Drag and Drop*, describes how you can implement drag and drop in your CAPI application.

The Reference Manual

The second part of the CAPI documentation is the *LispWorks CAPI Reference Manual*. This provides a complete description of every CAPI class, function and macro, and also provides a reference chapter on the Graphics Port functions. Entries are listed alphabetically, and the typographical conventions used are similar to those used in *Common Lisp: the Language* (2nd Edition) (Steele, 1990).

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 Mac OS X. Using CAPI with Motif is deprecated.

1.2 The history of the CAPI

Window-based applications written with LispWorks 3 and previous used CLX², 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.

1.3 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.

A different approach would have been to simulate the window objects and their look and feel. This approach is problematic. Because the library makes itself entirely responsible for the application's look and feel, it may not simulate it correctly in obscure cases. Also, manufacturers occasionally change the look and feel of their window systems. Applications written with a library that simulates window objects will continue to have the old look and feel until the application is recompiled with an updated library.

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.

1.3.1 CAPI Classes

There are four basic objects in the CAPI model: *interfaces*, *menus*, *panes* and *layouts*.

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.

An interface can contain a number of menus which are collected together on a menu bar. Each menu on the menu bar can contain menu items or other menus. 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`**, **`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`**.

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.

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")
```

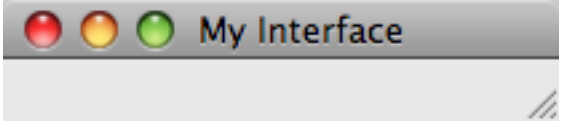
2.2 Creating a window

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

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

(display *)
```

Figure 2.1 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 Chapter 7.

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 Chapter 10, "Defining Interface Classes" on how to display CAPI elements in an interface.

Note that a displayed CAPI element should only be accessed in its own thread. See "The correct thread for CAPI operations" on page 25 for more information about this.

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

1. Enter the following into a listener:

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

```
(contain *)
```

Figure 2.2 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:

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

```
(contain *)
```

Figure 2.3 Specifying a callback

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

A dialog appears containing the message “Hello World”.

Figure 2.4 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

Creating Common Windows

So far you have only seen two types of CAPI element: the interface (which is the top level CAPI element, and is present in any CAPI window) and the **push-button**. This section shows how you can use the CAPI to create other common windowing elements you are likely to need.

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 future examples.

3.1 Generic properties

Because CAPI elements are just like CLOS classes, many elements share a common set of properties. 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** keywords. 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** keyword. 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 **gp: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)))
```

Figure 3.1 An example of the use of font descriptions

3.1.4 Mnemonics

This section applies to Windows and GTK+ only.

Underlined letters in menus, titles and buttons are called mnemonics. The user can select the element by pressing the corrresponding 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 “Mnemonics in buttons” on page 22 and the *LispWorks CAPI Reference Manual*.

For information on controlling mnemonics in button panels, see “Mnemonics in button panels” on page 36. For information on controlling mnemonics in menus, see “Mnemonics in menus” on page 84.

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** in the *LispWorks CAPI Reference Manual*.

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 XP can hide mnemonics when the user is not using the keyboard. This is controlled by

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

3.2 Specifying titles

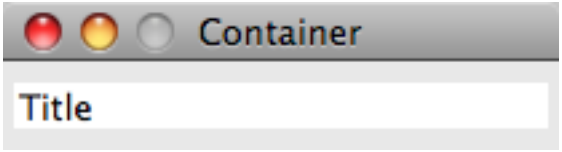
It is possible to specify a title for a window, or part of a window. Several of the examples that you have already seen have used titles. There are two ways that you can create titles: by using the **title-pane** class, or by specifying a title directly to any subclass of **titled-object**.

3.2.1 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)
```

Figure 3.2 A title pane



3.2.2 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 graphing them using the class browser.

3.2.2.1 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 acheived by using **set-default-interface-prefix-suffix** and/or specializing **interface-extend-title** as illustrated in “Controlling the interface title” on page 103.

3.2.2.2 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.

3.3 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 are used for entering short pieces of text, and editor panes are commonly used for dealing with large amounts of text such as files. Rich text panes are available on Cocoa and Windows, supporting formatted text.

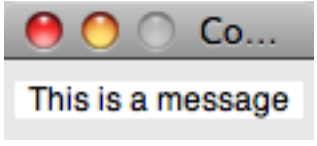
3.3.1 Display panes

Display panes can be used 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** keyword 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)
```

Figure 3.3 A display pane



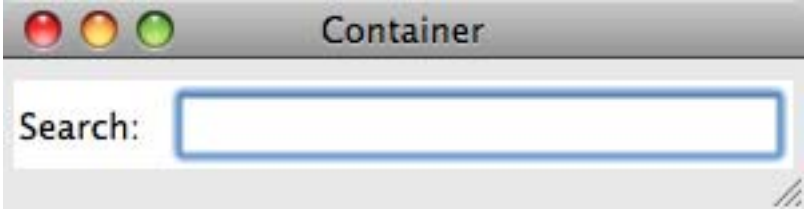
Note that the window title, which defaults to "Container" for windows created by **contain**, may appear truncated.

3.3.2 Text input panes

When you want the user to enter a line of text — for instance a search string — a text input pane can be used.

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

(contain text)
```

Figure 3.4 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 add toolbar buttons for easier user input 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**.

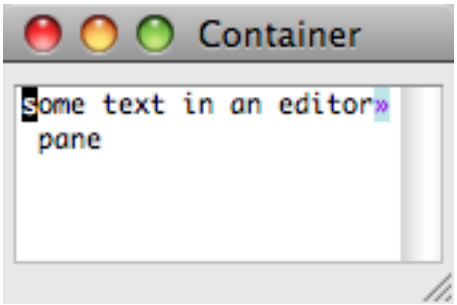
3.3.3 Editor panes

Editor panes can be created using the **editor-pane** element.

```
(setq editor
      (make-instance 'editor-pane
                     :text
                     "some text in an editor pane"))

(contain editor)
```

The Editor tool in the LispWorks IDE, as described in the *LispWorks IDE User Guide* and the *LispWorks Editor User Guide*, uses **editor-pane**.

Figure 3.5 An editor pane



Note: when you supply the **:buffer-name** initarg and / or the **:text** initarg with positive length, then the **editor-pane** initially displays a new buffer containing that text and / or with the specified buffer name. If you do not supply one of those arguments, then the **editor-pane** displays some existing editor buffer chosen at random. See the *LispWorks CAPI Reference Manual* for details.

The cursor in an **editor-pane** blinks on and off under the control of the **editor-pane-blink-rate** mechanism.

An **editor-pane** can be made non-editable by users with the initarg **:enabled :read-only**, or completely disabled with **:enabled nil**.

3.4 Displaying formatted text

Two classes allow you to display the Rich Text and HTML formats.

3.4.1 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 the example in:

examples/capi/applications/rich-text-editor.lisp

3.4.2 HTML

On Microsoft Windows and Cocoa, **browser-pane** allows you to display HTML, navigate, refresh, handle errors, redirect to another URL, and so on.

3.5 Stream panes

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

3.5.1 Collector panes

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

```
1. (contain (make-instance 'collector-pane
                           :title "Example collector pane:"))
2. (princ "abc" (collector-pane-stream *))
```

3.5.2 Interactive streams

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

```
(contain (make-instance 'interactive-stream
                        :title "Stream:"))
```

3.5.3 Listener panes

The **listener-pane** class is a subclass of **interactive-stream**, 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:"))
```

3.6 Miscellaneous button elements

A variety of different buttons can be created for use in an application. These include push buttons, which you have already seen, and check buttons. Button panels can also be created, and are described in Chapter 6, “Choices”.

3.6.1 Push buttons

You have already seen push buttons in earlier examples. 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.

3.6.2 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)
```

Figure 3.6 A check button

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

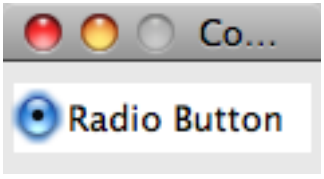
Like push buttons, check buttons can be disabled by specifying **:enabled nil**.

3.6.3 Radio buttons

Radio buttons can be created explicitly although they are usually part of a button panel as described in Chapter 6, *Choices*. The **:selected** keyword is used to specify whether or not the button is selected, and the **:text** keyword can be used to label the button.

```
(contain (make-instance 'radio-button
                        :text "Radio Button"
                        :selected t))
```

Figure 3.7 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.

3.6.4 Mnemonics in buttons

The initarg **:mnemonic** allows you to supply a character, integer or symbol specifying 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"))
```

3.7 Adding a toolbar to an interface

Top level interfaces can have a toolbar, which is typically displayed at the top of the window. On Cocoa, this will be a standard foldable toolbar.

The end user can raise a customization dialog to choose which items appear on the toolbar. See the *toolbar-items* and *toolbar-states* initargs for **interface** and the functions **interface-toolbar-state**, **interface-default-toolbar-states**, **interface-update-toolbar** and **interface-customize-toolbar**.

3.8 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.

3.8.1 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 the example in **examples/capi/graphics/pinboard-help.lisp**.

3.8.2 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** in the *LispWorks CAPI Reference Manual* for an example of a tooltip on a **text-input-pane**

3.8.3 Tooltips for toolbar buttons

You can implement tooltips for a **toolbar-button** exactly as for collections and so on as described in “Tooltips for collections, elements and menu items” on page 24. Supply *help-key* for the **toolbar-button** and a *help-callback* for the **interface**. For an example of this see **examples/capi/elements/tool-bar.lisp**.

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. For an example of this see **examples/capi/applications/simple-symbol-browser.lisp**.

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.

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 “Connecting an interface to an application” on page 100 is modified only by callbacks which run in the demo interface’s own process, and so there is no need to use **`execute-with-interface`** or **`apply-in-pane-process`**.

4.2 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 Mac OS X 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 Mac OS X 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 “Resizing and positioning” on page 70

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

5

Host Window System Configuration

This chapter describes how the host window system affects the appearance of CAPI windows, and how to configure it.

5.1 Properties of the host window system

This section describes properties of the host window system that affect the appearance and behavior of CAPI windows.

5.1.1 Using Windows themes

On Microsoft Windows XP, Vista and Windows 7 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.

5.1.2 Matching resources

You can configure the LispWorks IDE and your application to use resources on GTK+ and Motif. 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+ and Motif use to match resources for each widget.

5.1.2.1 Matching 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 calling **(setf 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 **examples/gtk/**.

5.1.2.2 Matching resources on X11/Motif

widget-name is used as described for GTK+ in “Matching resources on GTK+” on page 28, 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 6.1 and illustrates resources you may wish to change.

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.

5.1.2.3 Resources for LispWorks CAPI applications

Delivered applications which need fallback resources should pass the **:application-class** and **:fallback-resources** keys described in the *LispWorks CAPI Reference Manual* under **convert-to-screen**.

There is an example showing how to make a CAPI GUI configurable by GTK+ resources in **examples/capi/elements/gtk-resources.lisp**. To construct custom resources for your CAPI/GTK+ application, see the example resource files in **examples/gtk/**.

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.

5.1.2.4 X resources for in-place completion windows

The special window described in “In-place completion” on page 122 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**.

5.1.3 The break gesture

If a CAPI window is busy and unresponsive you can use the break gesture **Command+Ctrl+,** (comma) to regain control.

Note that this break gesture is specific to the window system your CAPI program is running in.

5.2 Using Motif

This section describes how to use the Motif window system on supported platforms.

5.2.1 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. For more information about this, see the *screen* argument to **display** and **convert-to-screen**.

5.2.2 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. For more information about this, see 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 *LispWorks Release Notes and Installation Guide* for further information on installing this option.

5.2.3 Using Motif on SPARC Solaris and HP-UX

LispWorks on SPARC Solaris and HP-UX does not support GTK+, and Motif is the only supported window system. You do not need to load it or specify the *screen* explicitly on these platforms.

6

Choices

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.

Some of the examples in this chapter require the functions **test-callback** and **hello** which were introduced in Chapter 3, "Creating Common Windows".

6.1 Button 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.

6.1.1 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 you click 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:

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

(contain *)
```

Figure 6.1 A push button panel

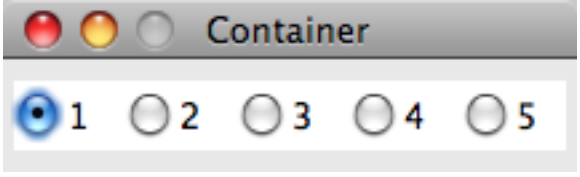
The layout of a button panel (for instance, whether items are listed vertically or horizontally) can be specified using the **:layout-class** keyword. This can take two values: **'column-layout** if you wish buttons to be listed vertically, and **'row-layout** if you wish them to be listed horizontally. 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 Chapter 7, "Laying Out CAPI Panes".

6.1.2 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)
```

Figure 6.2 A radio button panel



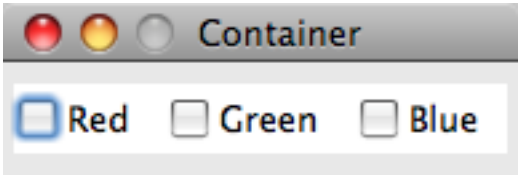
6.1.3 Check button panels

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")))
```

Figure 6.3 A check button panel



6.1.4 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.

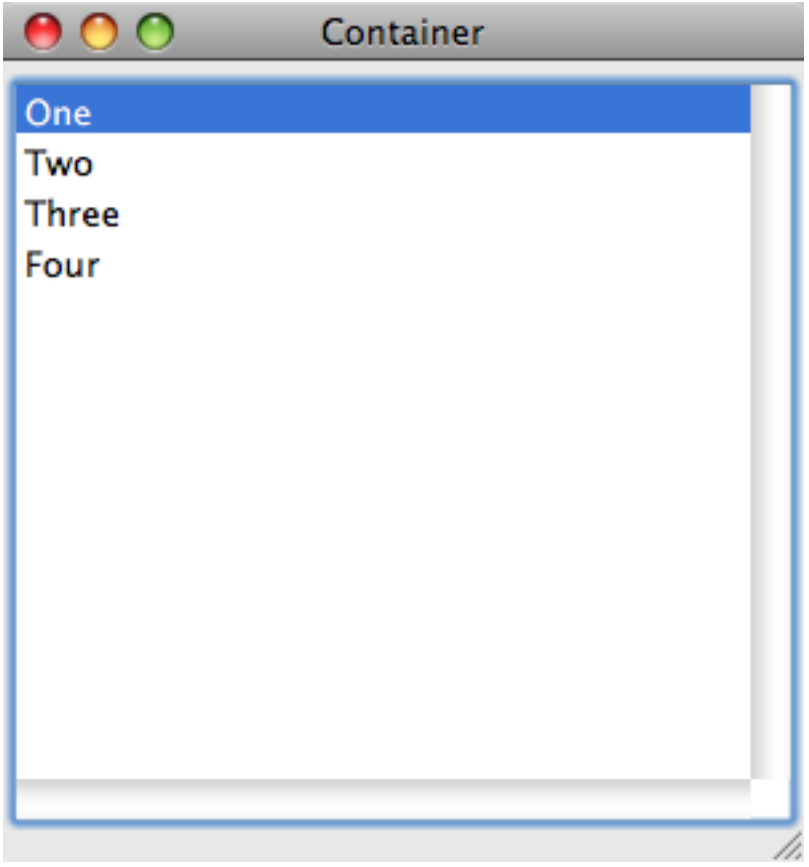
6.2 List panels

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)
```

Figure 6.4 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:

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

(contain *)
```

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

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

(contain *)
```

6.2.1 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 Section 6.2.2

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**.

6.2.2 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 whilst holding down the **`Shift`** key.

6.2.3 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:

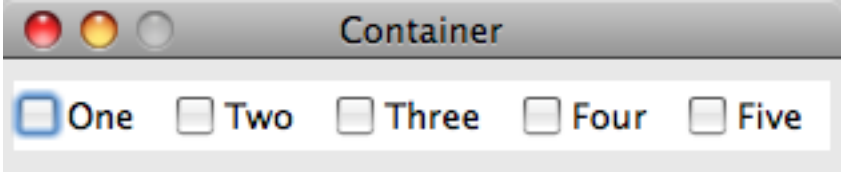
- **`:selection-callback`** — called when a selection is made
- **`:retract-callback`** — called when a selection is retracted

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:

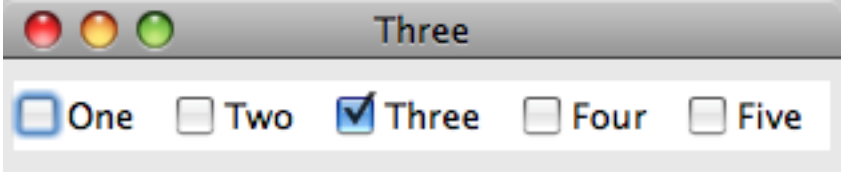
```
(defun set-title (data interface)
  (setf (interface-title interface)
        (format nil "~A" (string-capitalize data))))

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

(contain *)
```

Figure 6.5 The example check button panel before the callback.



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

Figure 6.6 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 pane, 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

6.2.4 Selections

List panels — all choices, in fact — can have selections, and you can set them from within Lisp. This is useful for providing default settings in a choice, or when a user selection has an effect on other settings than just the one they made.

The selection is represented as a vector of offsets into the list of the choice's items, unless it is a single-selection choice, in which case it is just represented as an offset.

The initial selection is controlled with the initarg **`:selection`**. The accessor **`choice-selection`** is provided.

6.2.5 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*.

6.2.6 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.

6.3 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.

6.3.1 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**.

6.3.2 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*.

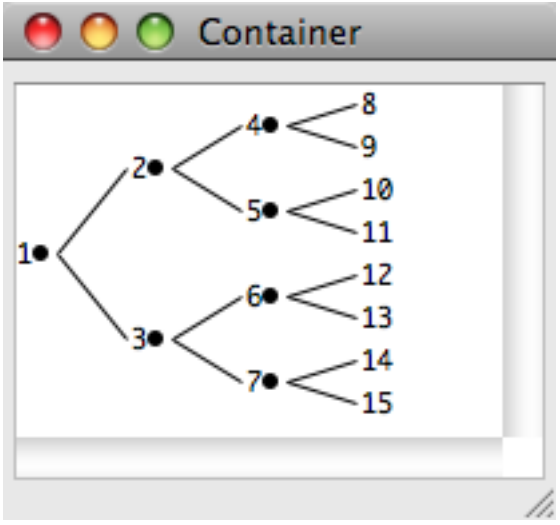
6.4 Graph panes

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.

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)))))))
```

Figure 6.7 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 *LispWorks CAPI Reference Manual*.

graph-pane provides a gesture which expands or collapses a node, depending on it 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).

6.5 Option panes

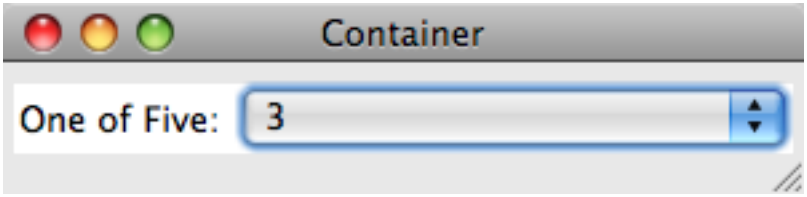
Option panes, created with the **option-pane** class, display the current selection from a single-selection list. When you click on the option pane, the list appears and you can make another selection from it. Once the selection is made, it is displayed in the option pane.

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:"))
```

Figure 6.8 An option pane



6.5.1 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

examples/capi/choice/option-pane-with-images.lisp

6.6 Text input choice

A **text-input-choice** class is provided which allows arbitrary text input augmented with a choice like an **option-pane**.

See **examples/capi/choice/text-input-choice.lisp**.

6.7 Menu components

Menus (covered in Chapter 9) 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 “Grouping menu items together” on page 78 for more details.

6.8 General properties of choices

The behaviors you have seen so far are mostly general properties of choices rather than being specific to a particular choice. These general properties are summarized below.

6.8.1 Interaction

All choices have an interaction style, controlled by the **:interaction** initarg. The **radio-button-panel** and **check-button-panel** are simply **button-panels** with their interactions set appropriately. The possibile 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 a compilation error.

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

6.8.2 Selections

All choices have a selection. This is a state representing the items currently selected. The selection is represented as a vector of offsets into the list of the choice's items, unless it is a single-selection choice, in which case it is just represented as an offset.

The initial selection is controlled with the initarg **:selection**. The accessor **choice-selection** is provided.

Generally, it is easier to refer to the selection in terms of the items selected, rather than by offsets, 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** .

6.8.3 Callbacks

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). See the **callbacks** entry in the *LispWorks CAPI Reference Manual* for a complete description of **:callback-type** values.

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 a compile-time error.

7

Laying Out CAPI Panes

So far, you have seen how you can create a variety of different window elements using the CAPI. Up to now, though, you have only created interfaces which contain one of these elements. The CAPI provides a series of layout elements which allow you to combine several elements in a single window. This chapter provides an introduction to the different types of layout available and the ways in which each can be used.

Layouts are created just like any other CAPI element, by using **make-instance**. Each layout must contain a description of the CAPI elements it contains, given as a list to the **:description** keyword.

A layout is used to group any instances of **simple-pane** and its subclasses (for instance all the elements you met in the last chapter), and pinboard object and its subclasses (discussed in Chapter 12, "Creating Your Own Panes"). 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))
```

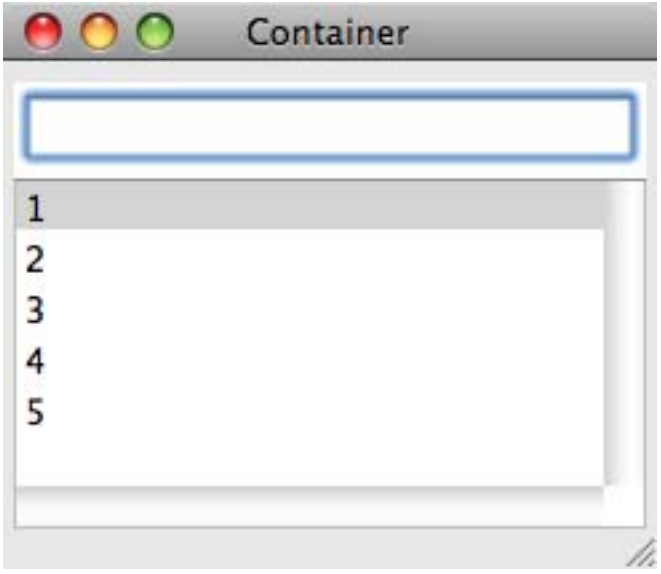
7.1 Organizing panes in columns and rows

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)))))
```

Figure 7.1 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)))
```

Figure 7.2 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 whilst others in the same row or column adjust proportionately when the interface is resized:

```
(contain
 (make-instance
  'column-layout
  :description
  (list
   (make-instance 'editor-pane
                  :text "Resizable"
                  :visible-min-height '(:character 1))
   (make-instance 'editor-pane
                  :text "Fixed"
                  :visible-min-height '(:character 1))
   (make-instance 'editor-pane
                  :text
                  (format nil "Resizable~%Resizable~%Resizable")
                  :visible-min-height '(:character 3)))
  :y-ratios '(1 nil 3)
  ))
```

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 :grey))
```

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 :grey))
```

7.2 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.

7.2.1 Grid layouts

Whereas row and column layouts only allow you to position a pane horizontally *or* vertically (depending on which class you use), 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

```
examples/capi/layouts/titles-in-grid.lisp
```

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

```
examples/capi/layouts/extend.lisp
```

7.2.2 Simple layouts

Simple layouts control the layout of only one pane. Where possible, the pane is resized to fit the layout. Simple layouts are sometimes useful when you need to encapsulate a pane.

7.2.3 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 Chapter 12, "Creating Your Own Panes".

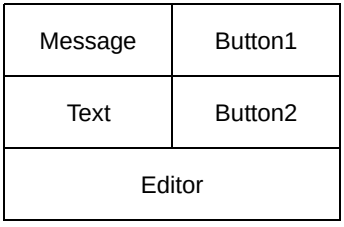
7.3 Combining different layouts

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 Figure 7.3. 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.

Figure 7.3 A sample layout

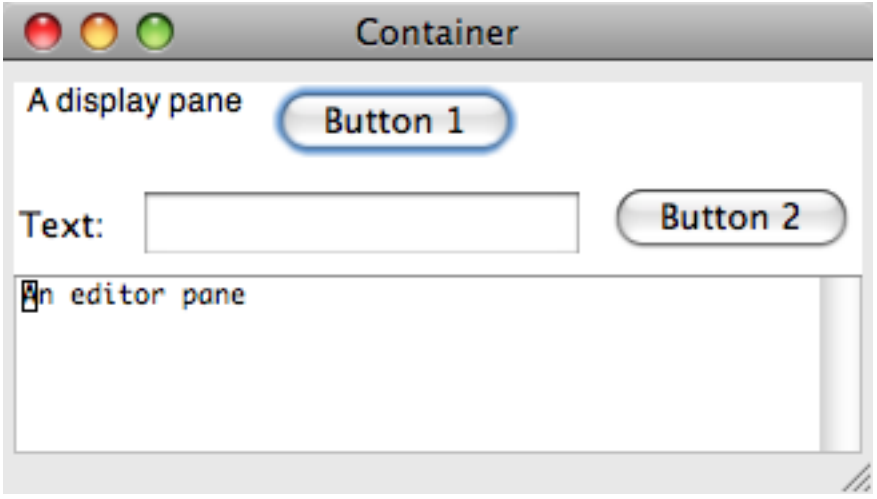


The layout in Figure 7.3 can be achieved by creating two row layouts: one containing the display pane and a button, and one containing the text input pane and the other button, and then creating a column layout which uses these two row layouts and the editor.

```
(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)))
```

Figure 7.4 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 than each instance of a CAPI element must not be used in more than one place at the same time.

7.4 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.

7.4.1 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.

7.4.2 Width and Height Constraints

In the CAPI, there are three kinds of constraint: external, visible and internal. The following hints are recognized by all layouts:

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.

Internal constraints control the size of region used to display the contents of the pane:

`: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.

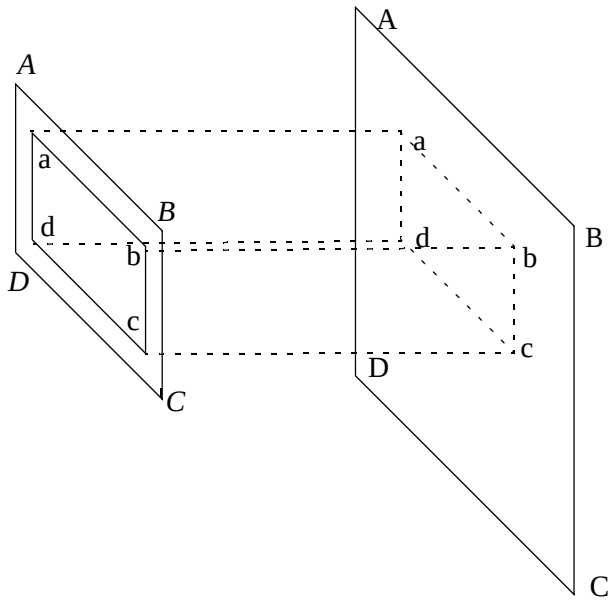
Initargs **:min-width**, **:max-width**, **:min-height** and **:max-height** are deprecated. They are synonyms for the visible constraints **:visible-min-width** and so on.

Each external size is the same as the visible size plus the borders.

For a non-scrolling pane, the internal constraints are the same as the visible constraints.

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. Usually the internal constraints are computed by the widget. Here is an illustration of the external, internal and visible sizes in a scrolling pane. *ABCD* is the external size, abcd is the visible size, and ABCD is the internal size:

Figure 7.5 External, visible and internal sizes:



7.4.3 Constraint Formats

Hints can take arguments in a number of formats, which are described in full in the *LispWorks CAPI Reference Manual*. 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.

7.4.3.1 Character constraints

In “Combining different layouts” on page 56, 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.

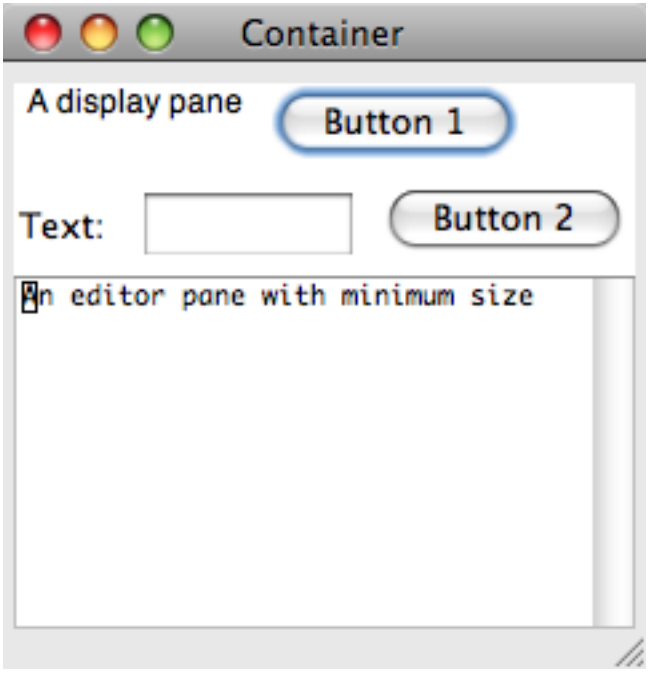
```
(setq editor2
      (make-instance 'editor-pane
                     :text "An editor pane with minimum size"
                     :visible-min-width '(:character 30)
                     :visible-min-height '(:character 10)))
```

Now display a window similar to the last example, but with the **editor2** editor pane. Note that it is only the decription 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:

Figure 7.6 The result of resizing the sample layout



7.4.3.2 String constraints

To make a pane that is wide enough to accomodate 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.

7.4.4 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 “Character constraints” on page 61 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 synchronised 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.

7.4.4.1 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.

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 constraints on the heights of 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)))
```

7.5 Advanced pane layouts

Until now you have used layouts for CAPI elements in which the constituents were displayed in fixed positions set out by the CAPI. In this chapter we will be looking at a number of ways in which users can select the layout and display of CAPI elements in an interface once an instance of the interface has been displayed.

The portable techniques are the use of dividers, switchable layouts and tab layouts. On Microsoft Windows, there is also Multiple-Document Interface (MDI).

Throughout this section we will be using three predefined panes, which you should define before proceeding.

```
(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))
```

7.5.1 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 given as a list to the **`:description`** keyword. Here is an example:

```
(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)
```

7.5.2 Tab layouts

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. Don't forget to close the switchable layout window created in the last example before displaying this:

```
(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 *)
```

Figure 7.7 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
               :key-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))
```

7.5.3 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 non-movable visible element between adjacent panes, 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 :grey))
```

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

7.5.4 Multiple-Document Interface (MDI)

In LispWorks for Windows, the CAPI supports MDI through the class **document-frame**. See the entry for **document-frame** in the *LispWorks CAPI Reference Manual*.

MDI is not supported on other platforms.

8

Modifying CAPI Windows

An interface or its children can be altered 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`**. See the *LispWorks CAPI Reference Manual* for details of these functions.

8.1 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.

8.2 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 “Changing the constraints” on page 63.

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 Mac OS X menu bar or the Microsoft Windows taskbar, as described in “Positioning CAPI windows” on page 71.

8.2.1 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 Mac OS X menu bar) therefore you cannot be certain of the position of *W1*.

8.3 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)
```

Elsewhere this manual shows how an **`editor-pane`** can be scrolled using editor commands.

8.3.1 Automatic scrolling

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

8.4 Swapping panes and layouts

The class **switchable-layout** is useful when your interface has several panes of which exactly one should be visible at any time. The class **tab-layout** provides similar functionality in a Window-system specific way. See "Advanced pane layouts" on page 64

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: you must not reuse already-displayed panes in a CAPI layout.

8.5 Specifying panes and layouts dynamically

If you create a `row-layout` or `column-layout` with an empty *description* then you can populate these layouts dynamically

To do this, use `make-instance` to create the panes, and pass a list of pane objects to `(setf layout-description)` in the layout's process. This can be done in an `initialize-instance :after` method.

8.6 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 *LispWorks CAPI Reference Manual* entry for the particular pane class and its superclasses.

8.6.1 Updating windows in real time

If your code needs to cause visible updates whilst 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.

8.7 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 initally. For details see the documentation for `top-level-interface-display-state` in the *LispWorks CAPI Reference Manual*.

8.8 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.

8.9 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`**.

9

Creating Menus

You can create menus for an application using the **menu** class.

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"))
```

9.1 Creating a menu

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

1. Enter the following into a Listener:

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

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

(display *)
```

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.

2. Enter the following into a Listener:

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

(make-instance 'menu
    :title "Baz"
    :items (list 1 2 * 4 5)
    :callback 'test-callback)

(contain *)
```

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.

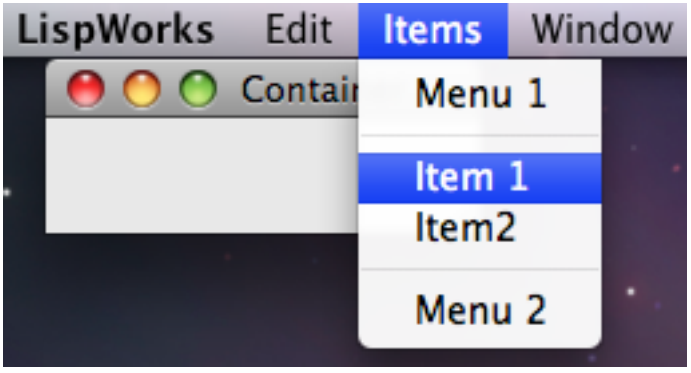
9.2 Grouping menu items together

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" "item2")
                      :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)
```

Figure 9.1 A menu



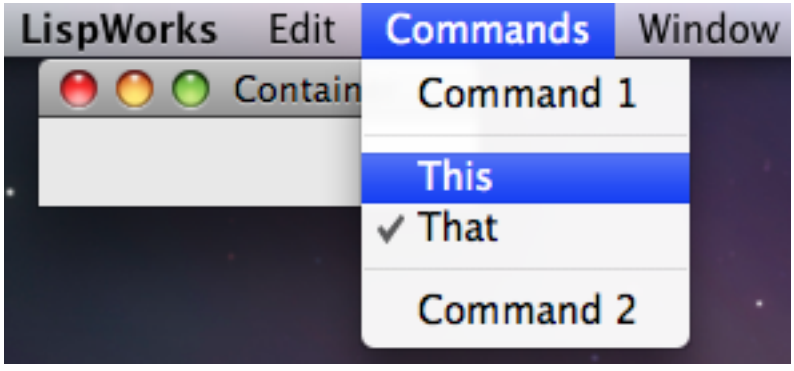
Menu components allow you to specify, via the **:interaction** keyword, selectable menu items — either as multiple-selection or single-selection items. This is like having radio buttons or check boxes as items in a menu, and is a popular technique among many GUI-based applications.

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)
```

Figure 9.2 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.

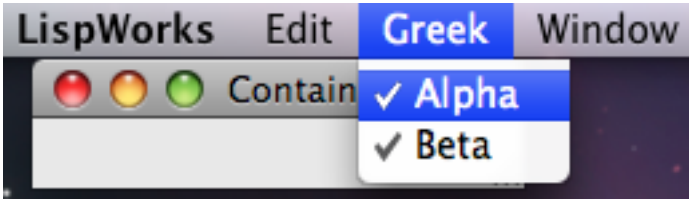
This example also illustrates the use of more than one callback in a menu, which of course is the usual case when you are developing real applications. Choosing either of the radio buttons displays one message on the screen, and choosing either **Command1** or **Command2** returns the arguments of the 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))
```

Figure 9.3 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 modifer keys. See “Alternative menu items” on page 85 for more information.

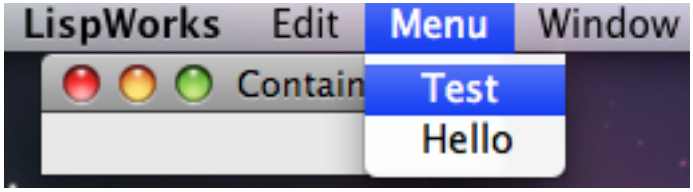
9.3 Creating individual menu items

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)
```

Figure 9.4 Individual menu items



Remember that each instance of a menu item must not be used in more than one place at a time.

9.4 The CAPI menu hierarchy

The combination of menu items, menu components and menus can create a hierarchical structure as shown schematically in Figure 9.5 and graphically in Figure 9.6. 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))
```

Figure 9.5 A schematic example of a menu hierarchy

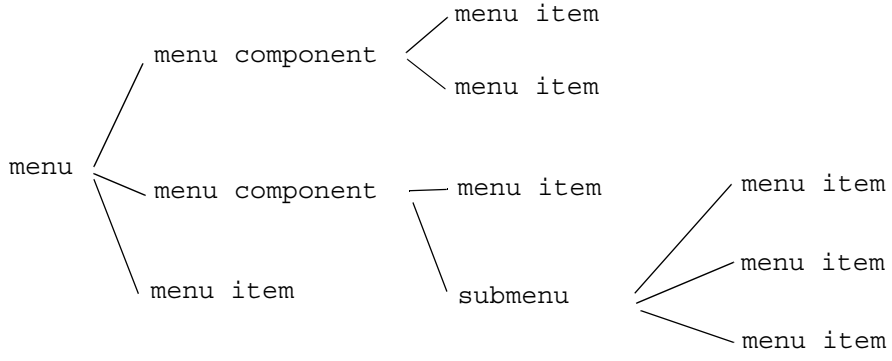
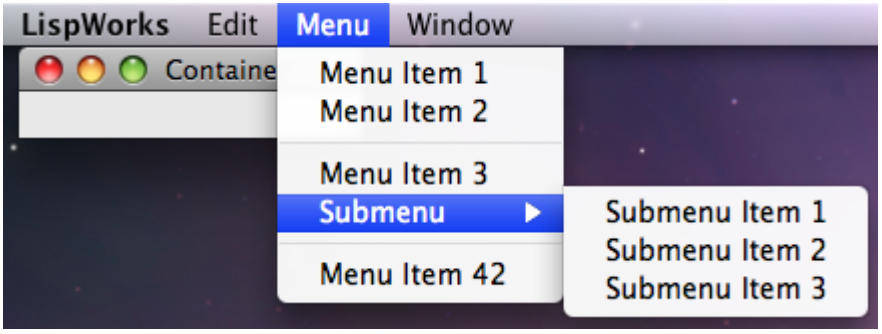


Figure 9.6 An example of a menu hierarchy



9.5 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))))
```

9.6 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 initarg **`: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 “Grouping menu items together” on page 78. 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 no 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

```
examples/capi/elements/accelerators.lisp
```

Note: Accelerators of alternative items do not work on Motif.

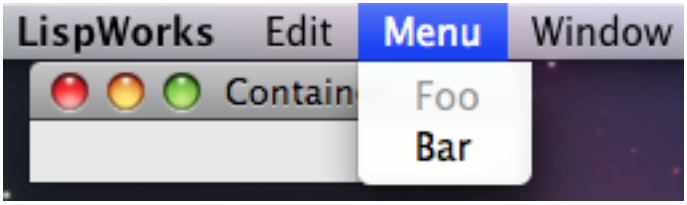
9.7 Disabling menu items

A function can be specified via the **`:enabled-function`** initarg, 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"))))
```

Figure 9.7 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.

9.7.1 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`**.

9.8 Menus with images

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

```
examples/capi/elements/menu-with-images.lisp
```

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.

9.9 The Edit menu on Cocoa

LispWorks now 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 "Adding menus" on page 97.

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.

9.10 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 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 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*.

9.11 The Application menu

The CAPI includes an interface to the Application menu supporting standard Mac OS X behaviors in your delivered LispWorks applications.

See the examples in:

```
examples/capi/applications/cocoa-application
examples/delivery/macos/simple-application.lisp
examples/delivery/macos/full-application.lisp
```

and the manual entries in the *LispWorks CAPI Reference Manual*.

10

Defining Interface Classes

So far we have looked at various components for building interfaces. The CAPI provides all these and more, but instead of continuing with our exploration of the various classes provided, let us see how what we have learned so far can be combined into a single, non-trivial interface class.

10.1 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 **viewer-pane** in the example below, 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**.

10.2 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:

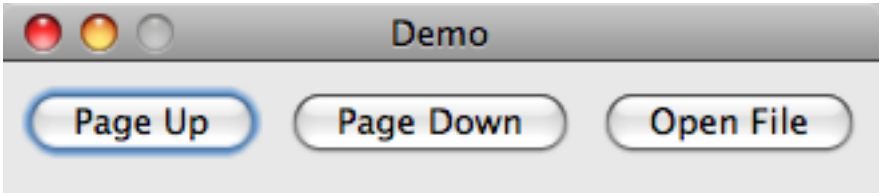
```
(make-instance 'demo)

(display *)
```

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.

Figure 10.1 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".

10.2.1 How the example works

Examine the **`define-interface`** call to see how this interface was built. The first part of the call to define interface 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`** call occurs after these **`defclass`**-like preliminaries. The remainder of a **`define-interface`** call lists all 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 pane.

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 given so far 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 three buttons side by side at the top of the pane.

10.3 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 Chapter 7, *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 for the editor pane in the layouts section. 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 row-with-editor-pane))
  (row-of-buttons row-layout
                  '(page-up page-down open-file))
  (row-with-editor-pane row-layout
                        '(viewer)))
```

This creates another **row-layout** for the new pane and then encapsulates the two row layouts into a third **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, **define-interface** uses the first one listed.

By putting the layout of buttons and the layout with the editor pane in a column layout, their relative position has been controlled: the buttons appear in a row above the editor pane.

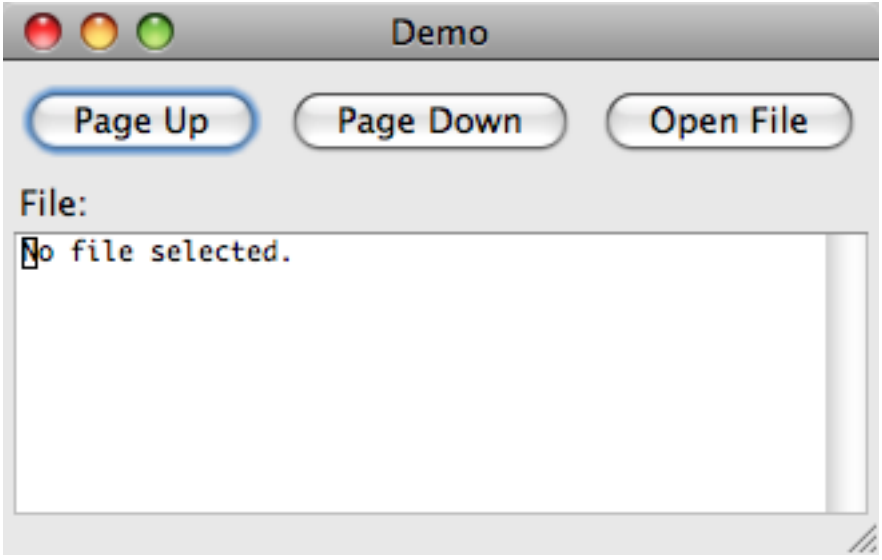
The code for the new interface is now as follows:

```
(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 row-with-editor-pane))
   (row-of-buttons row-layout
                   '(page-up page-down open-file))
   (row-with-editor-pane row-layout
                         '(viewer)))
  (:default-initargs :title "Demo"))
```

Displaying an instance of the interface by entering the line of code below produces the window in Figure 10.2:

```
(display (make-instance 'demo))
```

Figure 10.2 A CAPI interface with editor pane



10.3.1 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.

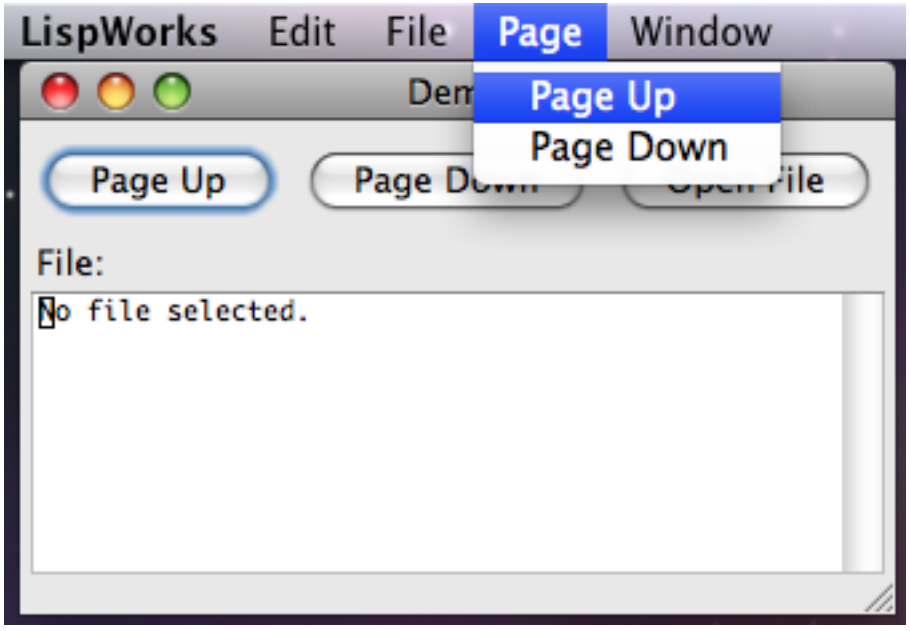
In this instance the menu item descriptions are just strings naming each item, but you may wish to supply initialization arguments for an item — in which case you would enclose the name and those arguments in a list.

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 row-with-editor-pane))
    (row-of-buttons row-layout
                    '(page-up page-down open-file))
    (row-with-editor-pane row-layout
                          '(viewer)))
  (:menus
   (file-menu "File"
              ("Open"))
   (page-menu "Page"
              ("Page Up" "Page Down")))
  (:menu-bar file-menu page-menu)
  (:default-initargs :title "Demo"))
```

Figure 10.3 A CAPI interface with menu items



The menus contain the items specified — try it out to be sure.

10.4 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))
            (with-open-file (stream file)
              (let ((buffer
                     (make-array 1024
                                 :element-type
                                 (stream-element-type stream)
                                 :adjustable t
                                 :fill-pointer 0)))
                (do ((char (read-char stream nil nil)
                           (read-char stream nil nil)))
                    ((null char))
                  (vector-push-extend char buffer))
                (subseq buffer 0)))))))
```

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.

```
(define-interface demo ()
  ()
  (: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))
   (:layouts
     (main-layout column-layout
                  '(row-of-buttons row-with-editor-pane))
     (row-of-buttons row-layout
                     '(page-up page-down open-file))
     (row-with-editor-pane row-layout
                           '(viewer)))
   (: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"))
```

10.5 Controlling the interface title

You can add dynamic control of window titles using the functions illustrated in the section.

Firstly we add a counter to the title of new **demo** windows:

```
(defvar *demo-title-counter* 0)

(defmethod capi:interface-extend-title ((self demo) title)
  (let ((counter
         (or (capi:capi-object-property self 'my-title-counter)
             (setf
              (capi:capi-object-property self 'my-title-counter)
              (incf *demo-title-counter*)))))
    (format nil "~A - ~D"
            (call-next-method)
            counter)))

(capi:display (make-instance 'demo))
```

Then we specify a common prefix for all interface window titles. Note that this will affect all interfaces in the current session:

```
(capi:set-default-interface-prefix-suffix
 :prefix "My " :suffix nil )

(capi:display (make-instance 'demo))
```

10.6 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 Mac OS X 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.

10.6.1 Support for multiple monitors

CAPI supports multiple monitors by providing APIs (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 Mac OS X menu bar.

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

11

Prompting for Input

A dialog is a window that receives some input from the user and returns it to the application. For instance, if the application wants to know where to save a file, it could prompt the user with a file 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 work 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.

11.1 Some simple dialogs

The simplest form of dialog is a message dialog. The function **display-message** behaves very much like **format**.

```
(display-message "Hello world")
```

Figure 11.1 A message dialog


```
(display-message
             "This function is ~S"
             'display-message)
```

Figure 11.2 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?")
```

Figure 11.3 A message dialog prompting for confirmation

For more control over such a dialiog, use the function **prompt-for-confirmation**. See the *LispWorks CAPI Reference Manual* for details.

11.2 Prompting for values

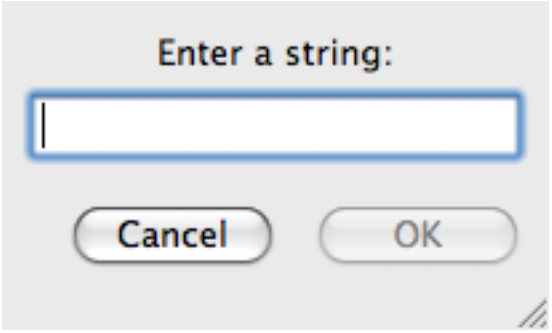
The CAPI provides a number of different dialogs for accepting values from the user, ranging from accepting strings to accepting whole Lisp forms to be evaluated.

11.2.1 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:")
```

Figure 11.4 A dialog prompting for a string



An initial value can be placed in the dialog by specifying the keyword argument **:initial-value**.

11.2.2 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)
```

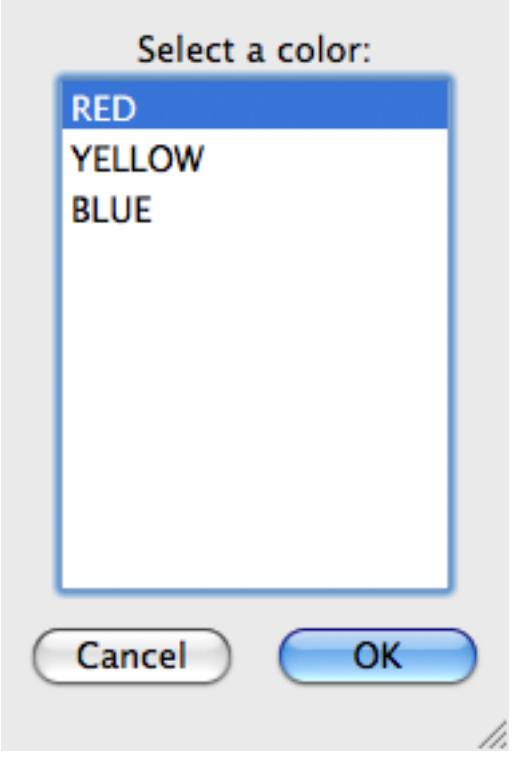
Try also the function **prompt-for-number**.

11.2.3 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:")
```

Figure 11.5 A dialog prompting for a selection from a list



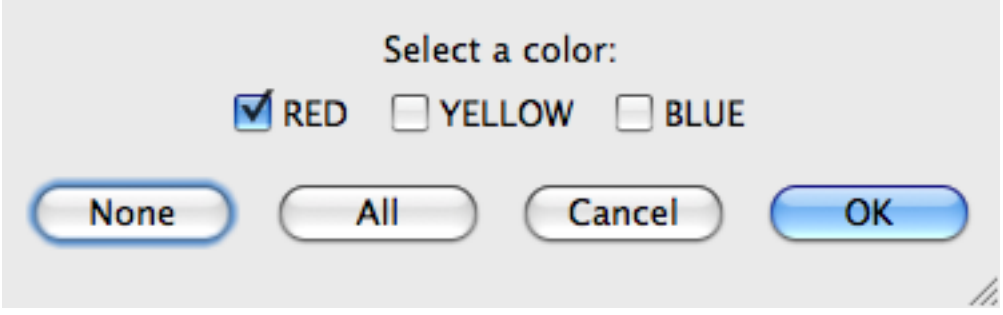
You can also specify the interaction style that you would like for your dialog, which can be any of the interactions accepted by a choice. The specification of the interaction style to this choice is made using the keyword argument **:interaction**:

```
(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)
```

Figure 11.6 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))
```

Figure 11.7 Selection from a column of buttons

There is a more complex example in

```
examples/capi/choice/prompt-with-buttons.lisp
```

11.2.4 Prompting for files

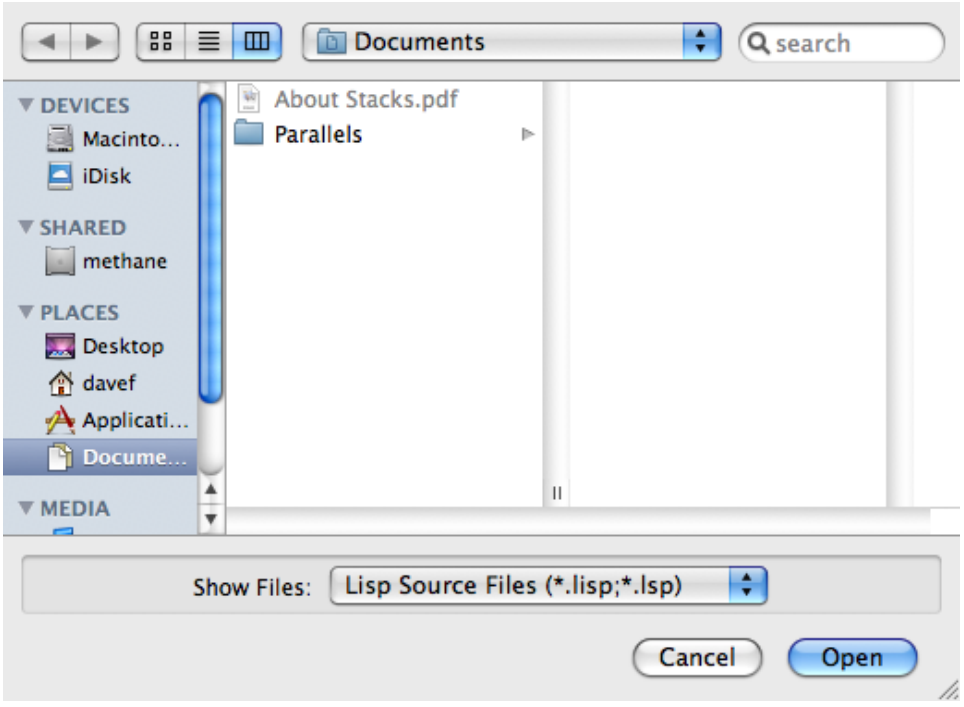
To prompt for a file, use the function **prompt-for-file**:

```
(prompt-for-file
 "Enter a file:")
```

You can also specify a starting pathname:

```
(prompt-for-file
  "Enter a filename:"
  :pathname (sys:get-folder-path :documents))
```

Figure 11.8 Selection of a file



Try also the function **prompt-for-directory**.

11.2.5 Prompting for fonts

To obtain a **gp:font** object from the user call **prompt-for-font**.

11.2.6 Prompting for colors

To obtain a color specification from the user call **prompt-for-color**.

11.2.7 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, like all those in this chapter so far, prevents input to other LispWorks windows while it is displayed. For information about creating dialog sheets which are not application-modal, see “Window-modal Cocoa dialogs” on page 115.

11.3 Window-modal Cocoa dialogs

By default, CAPI dialogs on Cocoa use sheets which are application-modal. This means that the application waits until the sheet is dismissed and does not allow the user to work with its other windows until then.

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.

11.3.1 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**.

11.3.2 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 “Prompting for Lisp objects” on page 115.

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 “Prompting for Lisp objects” on page 115.

For more details, see the page for **with-dialog-results** in the *LispWorks CAPI Reference Manual*.

11.4 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.

11.4.1 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.

11.4.2 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 process (see the *process* argument to **display**). Creating cross-process 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*.

11.5 Creating your own dialogs

The CAPI provides a number of built-in dialogs which should cover the majority of most peoples needs. However, there is always the occasional need to create custom built dialogs, and the CAPI makes this very simple, using the function **display-dialog** which displays any CAPI interface as a dialog, and

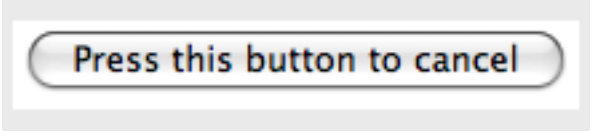
the functions **exit-dialog** and **abort-dialog** as the means to return from such a dialog.

11.5.1 Using display-dialog

Here is a very simple example that displays a **Cancel** button in a dialog, and when that button is pressed the dialog is cancelled. Note that **display-dialog** must receive an interface, so an interface is created for the button by using the function **make-container**.

```
(display-dialog
 (make-container
  (make-instance
    'push-button
    :text "Press this button to cancel"
    :callback 'abort-dialog)
   :title "My Dialog"))
```

Figure 11.9 A cancel button



The function **abort-dialog** cancels the dialog returning the values **nil** and **nil**, which represent a return result of **nil** and the fact that the dialog was cancelled, respectively. Note also that **abort-dialog** accepts any values and just ignores them.

The next problem is to create a dialog that can return a result. Use the function **exit-dialog** which returns the value passed to it from the dialog. The example below shows a simple string prompter.

```
(display-dialog
  (make-container
   (make-instance
    'text-input-pane
    :callback-type :data
    :callback 'exit-dialog)
   :title "Enter a string:"))
```

Both of these examples are very simple, so here is a slightly more complicated one which creates a column containing both a text-input-pane and a **Cancel** button.

```
(display-dialog
 (make-container
  (list
   (make-instance
    'text-input-pane
    :callback-type :data
    :callback 'exit-dialog)
   (make-instance
    'push-button
    :text "Cancel"
    :callback 'abort-dialog))
  :title "Enter a string:"))
```

Note that this looks very similar to the dialog created by **prompt-for-string** except for the fact that it does not provide the standard **OK** button.

It would be simple to add an **OK** button in the code above, but since almost every dialog needs these standard buttons, the CAPI provides a higher level function called **popup-confirmer** that adds the standard buttons for you. Also it arranges for the **OK** and **Cancel** buttons to respond to the **Return** and **Escape** keys respectively. **popup-confirmer** is discussed in the next section.

11.5.2 Using popup-confirmer

The function **popup-confirmer** is a higher level function provided to add the standard buttons to user dialogs, and it is nearly always used in preference to **display-dialog**. 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")
```

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.

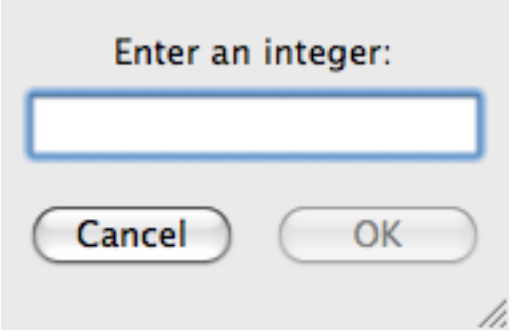
In order to do this conversion, **popup-confirmer** provides an alternative exit function to the usual **exit-dialog**. This is called **exit-confirmer**, and it does all of the necessary work on exiting.

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)
```

Figure 11.10 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 text-input-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)
```

11.5.3 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.

11.6 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. Certain gestures including **Up**, **Down** and **Return** operate on the special window and allow selection of an item. 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.

11.6.1 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 "Programmatic control of in-place completion" on page 126 for information on how to implement it.

11.6.1.1 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 *LispWorks Editor User Guide* for information on binding these commands to keyboard gestures. See **call-editor** in the *LispWorks CAPI Reference Manual* for information on calling them from CAPI.

11.6.1.2 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 **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 *LispWorks 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.

11.6.1.3 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).

11.6.1.4 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.

11.6.2 Programmatic control of in-place completion

You can add in-place completion to your application as described in this section.

11.6.2.1 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 *LispWorks CAPI Reference Manual* for details.

11.6.2.2 Editor panes

An **`editor-pane`** does in-place completion when your code calls the function **`editor:complete-in-place`**.

11.6.2.3 Other CAPI panes

You can also implement in-place completion on arbitrary CAPI panes by calling **`prompt-with-list-non-focus`**.

12

Creating Your Own Panes

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.

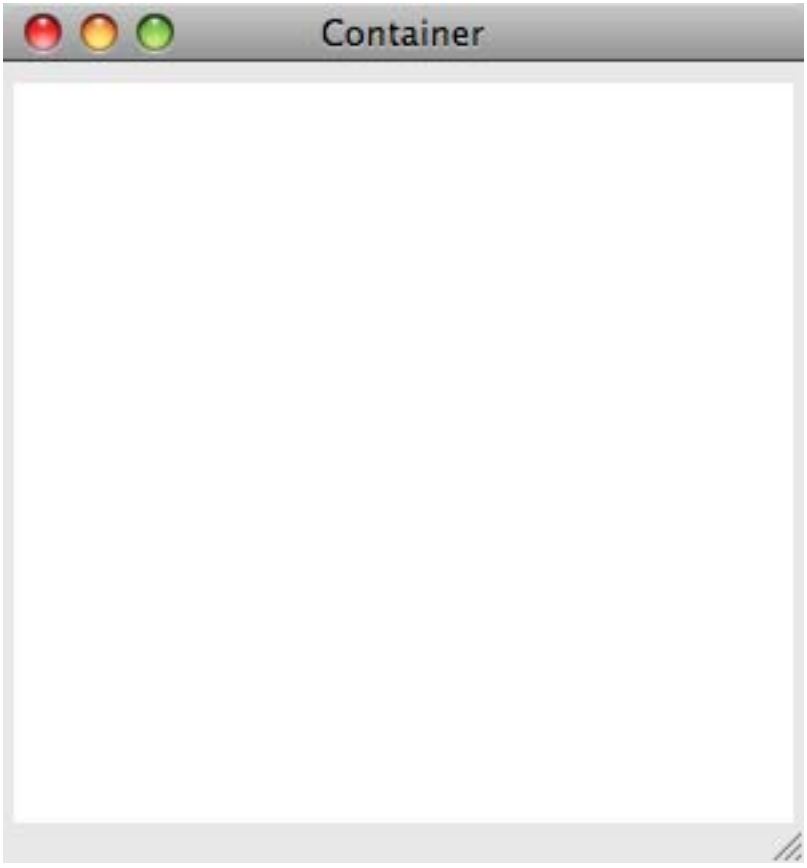
An **`output-pane`** is a fully functional graphics port. This allows it to use all of the graphics ports functionality to create graphics, and it also has a powerful input model which allows it to receive mouse and keyboard input.

12.1 Displaying graphics

The following is a simple example demonstrating how to create an **`output-pane`** and then how to draw a circle on it.

```
(setq output-pane
    (contain
         (make-instance 'output-pane)
         :best-width 300
         :best-height 300))
```

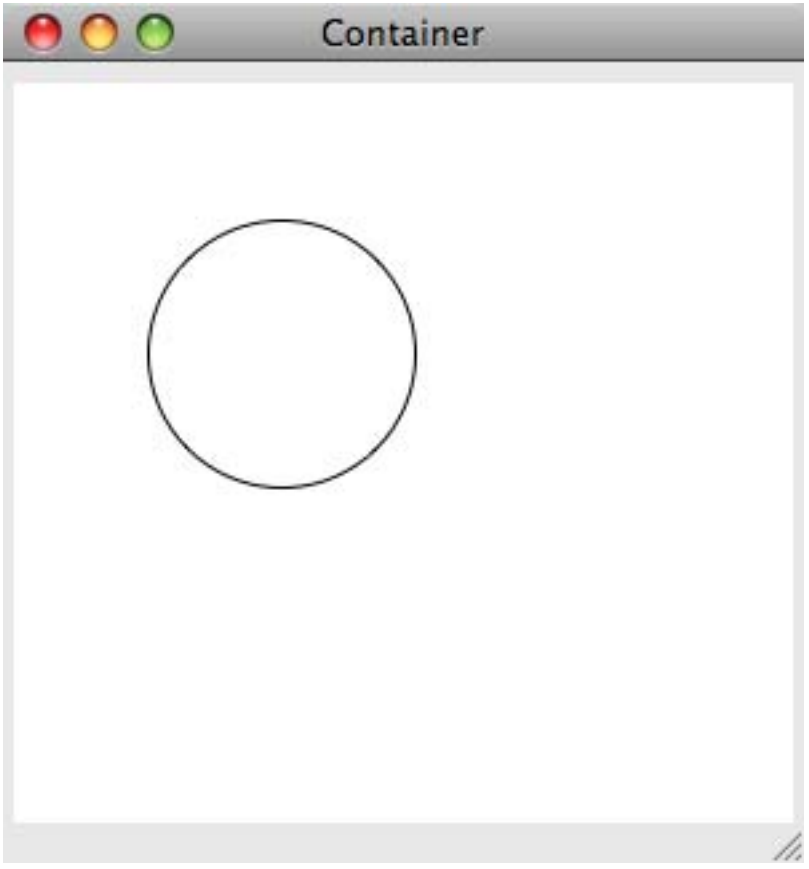
Figure 12.1 An empty output pane



Now you can draw a circle in the empty output pane by using the graphics ports function **draw-circle**. Note that the drawing function must be called in the process of the interface containing the output pane:

```
(capi:apply-in-pane-process
 output-pane 'gp:draw-circle output-pane 100 100 50)
```

Figure 12.2 An output pane containing a circle



Notice that this circle is not permanently drawn on the **output-pane**, and when the window is next redisplayed it vanishes. To prove this to yourself, force the window to be redisplayed (for example by iconifying or resizing it). At this point, you can draw the circle again yourself but it will not happen automatically.

```
(capi:apply-in-pane-process
 output-pane 'gp:draw-circle output-pane 100 100 50)
```

In order to create a permanent display, you need to provide a function to the **output-pane** that is called to redraw sections of the output-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 **output-pane** and the region in question.

For example, to create a pane that has a permanent 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 *LispWorks CAPI Reference Manual*.

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))
```

12.2 Receiving input from the user

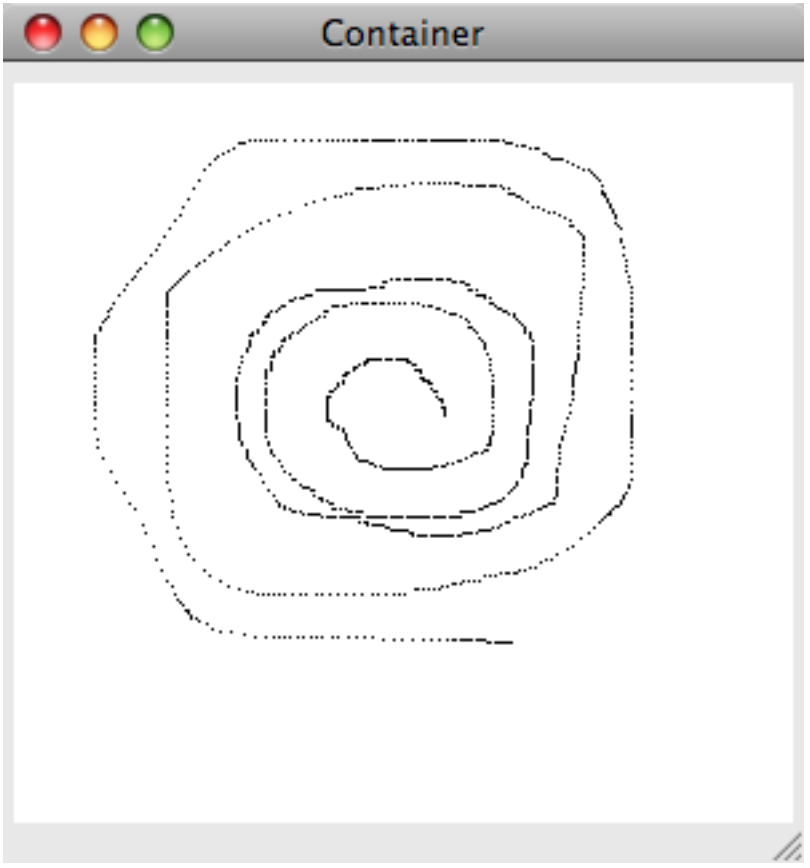
You now know enough to be able to create new classes of window which can display arbitrary graphics, but to be able to create interactive windows you need to be able to receive events. The CAPI supports this through the use of an input model, which is a mapping of events to the callbacks that should be run when they occur.

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 as follows:

```
(contain
 (make-instance
  'output-pane
  :input-model '(((:motion :button-1)
                  gp:draw-point))))
```

Figure 12.3 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

examples/capi/graphics/pinboard.lisp

See the manual page for **output-pane** in the *LispWorks CAPI Reference Manual* for the full *input-model* syntax..

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.

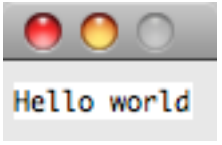
These graphical objects are called *pinboard objects*, as they can only be displayed if they are contained within a **pinboard-layout**. To define a pinboard-object, you define a subclass of **drawn-pinboard-object** and specify a drawing routine for it (and you can also specify constraints on the size of your object). You can then make instances of these objects and place them into layouts just as if they were ordinary panes. You can also place these objects inside layouts as long as there is a **pinboard-layout** higher up the layout hierarchy that contains the panes.

Note: **pinboard-object**s are implement 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**.

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"))
```

Figure 12.4 A pinboard object



There is another example illustrating **item-pinboard-object** in the file

examples/capi/graphics/pinboard-object-text-pane.lisp

12.3.1 Buffered drawing

Where the display of an **output-pane** is complex you may see flickering on screen on some platforms. Typcailly 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 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-objects** 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

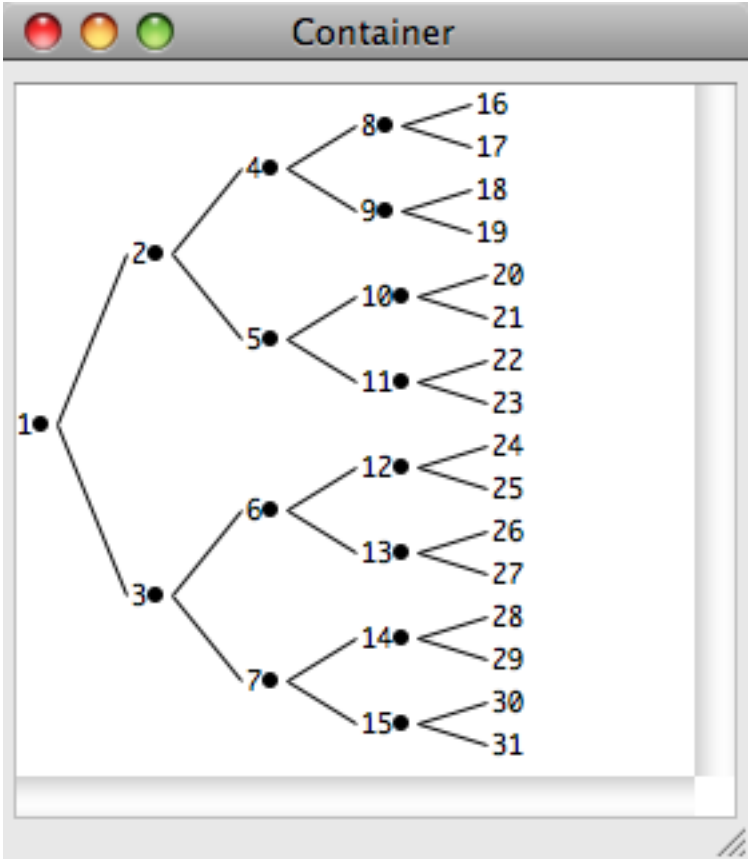
which pinboard object was under the pointer at the time, and it can then use this information to change the selection.

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)
```

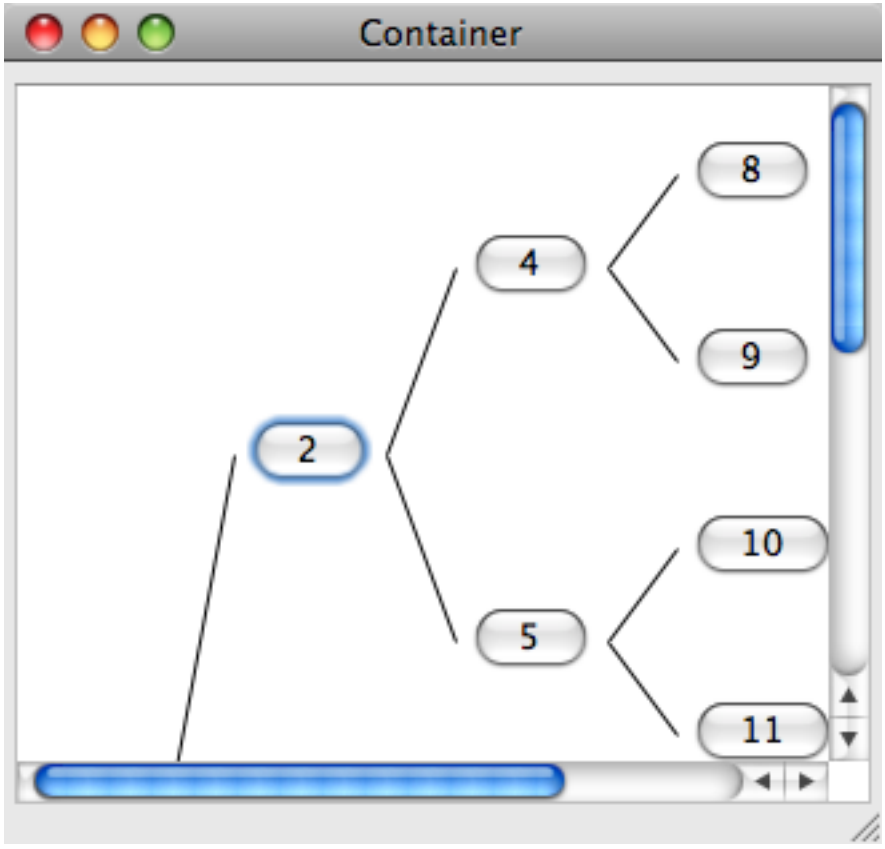
Figure 12.5 A graph pane with pinboard object nodes



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)
```

Figure 12.6 A graph pane with push-button nodes



12.3.3 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)
```

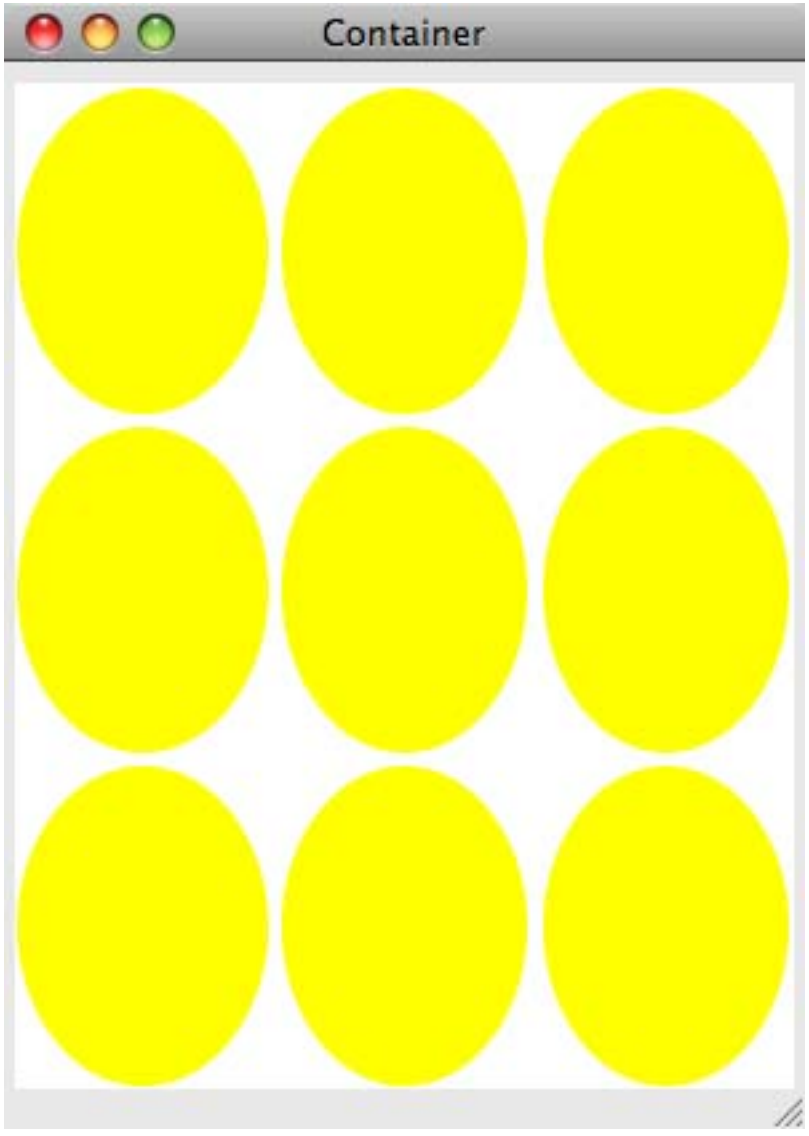
Figure 12.7 An ellipse-pane class

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. See the *LispWorks CAPI Reference Manual* for more details of **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 place 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)
```

Figure 12.8 Nine ellipse-pane instances in a layout



13

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 interned in and 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 **capi:output-pane** or a subclass, and are integral part of the CAPI panes system.
Pixmap ports	These are used for off-screen drawing. Once the drawing is completed they can be copied to another port (typically the screen, with **copy-area**), or converted to an image.
Printer ports	These are used for drawing to a printer.

Metafile ports — These are used for recording drawing operations so that the drawing can be realized later.

13.1.1 Creating instances

Graphics ports instances are created or temporarily redirected by any of these interfaces:

On-screen ports — **make-instance** with **capi:output-pane** or any subclass (including **capi:editor-pane**, **capi:pinboard-layout** and **capi:graph-pane**).

See the *LispWorks CAPI Reference Manual* documentation for **capi:output-pane** and the other CAPI classes.

Pixmap ports — **create-pixmap-port** and **with-pixmap-graphics-port**.

Metafile ports — **capi:with-internal-metafile** and **capi:with-external-metafile**.

Printer ports — **capi:with-print-job** and **capi:simple-print-port**.

See the *LispWorks CAPI Reference Manual* for full reference entries on all the Graphics Ports functions, macros, classes and types.

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 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* in the page for **graphics-state** in the *LispWorks CAPI Reference Manual*.

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*. See the page for **graphics-state** in the *LispWorks CAPI Reference Manual*.

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 exanple, 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 “Creating instances” on page 146.

These examples in the **examples/capi/graphics/** directory demonstrate features that are available only with *drawing-mode* **:quality**:

catherine-wheel.lisp
: Rotating a string.

compositing-mode-simple.lisp
: Using *compositing-mode*.

compositing-mode.lisp
: Using *compositing-mode*.

images-with-alpha.lisp
: Using *compositing-mode*, transforming an image.

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** in the *LispWorks CAPI Reference Manual*.

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 the **with-graphics-state** macro. 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))
```

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* color is used for the drawing.

Note: displaying images is described in "Working with images" on page 156.

13.4.1 Text

You can draw text with the functions **draw-string** and **draw-character**.

To control the font used, see "Portable font descriptions" on page 154.

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 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.

13.5.1 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 *(a, b)* and it is transformed to the point *Q (a', b')*

$P \Rightarrow Q$ or $(a, b) \Rightarrow (a', b')$

$a' = pa + rb$, $b' = qa + sb$.

$$Q = PM, \text{ where } M = \begin{vmatrix} p & q \\ r & s \end{vmatrix}$$

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:

$$Q = PM$$
$$= (a \;\; b \;\; 1) \begin{vmatrix} p & q & 0 \\ r & s & 0 \\ u & v & 1 \end{vmatrix}$$

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 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.

13.5.2 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 the Graphics Ports chapter of the *LispWorks CAPI Reference Manual* - see the entries for functions with names beginning **draw-**.

13.6 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.

13.6.1 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.

13.6.2 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+.

13.7 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 and images can be loaded 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 purpose of the *owner* slot in **`pixmap-graphics-port-mixin`**. The conversion of colors to pixel values 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.

13.7.1 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.

13.8 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`**.

13.8.1 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:

Table 13.1 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	

13.8.2 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 **capi:interface-display**, or a *display-callback* or possibly a *create-callback*. See **capi:interface** for more information about these initargs.

13.9 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 **capi:output-pane** (or a subclass of this). You can create a **image** object from:

- A file of recognized image type
- A registered image identifier (see “Registering images”)
- An **external-image** object
- An on-screen 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.

capi:image-pinboard-object, **capi:button**, **capi:list-view**, **capi:tree-view** and **capi:toolbar** all support images. There is also limited support for images in **capi:menu**. These classes handle the drawing and freeing for you.

13.9.1 Image formats supported for reading from disk and drawing

This table lists the formats supported at the time of writing:

Table 13.2 Operating system and supported image types

OS	Supported Image Types
Microsoft Windows	BMP, DIB, GIF, JPEG, PNG, TIFF, EMF, ICO
Mac OS X	BMP, DIB, GIF, JPEG, TIFF, PICT and many others. Also EPS, PDF
GTK+	BMP, DIB, GIF, JPEG, PNG, TIFF and many others.
X/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 X/Motif LispWorks uses the freeware **`imlib`** library which seems to be in all Linux distributions. However it is not in some UNIX systems, so you may need to install it.

Note: On Microsoft Windows ICO images are supported for certain situations such as **`capi:button`** and **`draw-image`** - see the *LispWorks CAPI Reference Manual* for details.

Note: On Microsoft Windows LispWorks additionally supports Windows Icon files with scaling - see **`load-icon-image`** in the *LispWorks CAPI Reference Manual*.

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.

13.9.2 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:

`examples/capi/graphics/images-with-alpha.lisp`

13.9.3 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.

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 the **`examples/capi/buttons/button.lisp`** example.

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 “Pixmap graphics ports” on page 153.

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)))
```

13.9.3.1 Transparency

An External Image representing an image with a color map can specify a transparent color. When converted and drawn, this color is drawn using the background color of the port.

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.

Note: *transparent-color-index* works only for images with a color map - those with 256 colors or less.

13.9.3.2 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 “Loading images” . Such conversions are cached but you can remove the caches by **clear-external-image-conversions**.

You an also convert an **image** to an **external-image** by calling **externalize-image**.

13.9.4 Registering images

One way to load an **image** is via a registered image identifier.

To establish a registered image identifer call **register-image-translation**:

```
(gp:register-image-translation
 'info-image
  (gp:read-external-image "info.bmp"
                          :transparent-color-index 7))
```

You can then do:

```
(gp:load-image port 'info-image)
```

to obtain the **image** object.

13.9.5 Loading images

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** 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 suports images. For example you can specify an *image* in a **capi: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 **:create-callback** of the interface or in the first **:display-callback** of the pane.

13.9.6 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**.

13.9.7 Drawing images

The function to draw an image is **draw-image**.

This must be called in the same process as the pane. Use **apply-in-pane-process**, **apply-in-pane-process-if-alive** or **execute-with-interface** if necessary.

13.9.8 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 a **capi: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. The pixel values are like those returned from **color:convert-color** and can be converted to RGB using **color: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 coordinaes of each pixel to write, 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.

There is an example that demonstrates the uses of Image Access objects in:

examples/capi/graphics/image-access.lisp

This further example demonstrates the uses of Image Access objects with colors that have an alpha component:

examples/capi/graphics/image-access-alpha.lisp

13.9.9 Creating external images from Graphics Ports operations

To create an **external-image** object from an on-screen window, use **with-pixmap-graphics-port** as in this example:

```
(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 **capi: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

The Color System

14.1 Introduction

The LispWorks 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.

For example, the call

```
(gp:draw-line my-port x1 y1 x2 y2
    :foreground :navyblue)
```

uses the keyword symbol **:navyblue** for the color of the line.

Colors are looked up in a color database. The LispWorks image is delivered with a large color database already loaded (approximately 660 entries.) The color database contains color-specs which define the colors in terms of a standard color model. When the drawing function is executed, the color-spec is converted into a colormap index (or “pixel” value).

The LispWorks Color System has facilities for:

- Defining new color aliases in one of several color models
- Loading the color database from a file of color descriptions
- Converting color specifications between color models

- Defining new color models

It is accessible from the **color** package, and all symbols mentioned in this chapter are assumed to be external to this package unless otherwise stated. You can qualify them all explicitly in your code, for example **color:apropos-color-names**.

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

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

This creates a package in which all the **color** symbols (and for convenience, **capi** as well) are accessible. To run the examples in this chapter, evaluate the form above and then:

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

The color-models available by default are RGB, HSV and GRAY.

14.1.1 Rendering of colors

Some colors do not render exactly as expected in some CAPI classes such as **capi:title-pane** - it depends on the palette provided by the rendering system.

However, **capi:output-pane** and its subclasses support non-standard palettes.

14.2 Reading the color database

To find out what colors are defined in the color database, use the functions **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**.

14.3 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:

```
(make-rgb 0.0s0 1.0s0 0.0s0) =>
  #(:RGB 0.0s0 1.0s0 0.0s0)
```

defines the color green in the RGB color model. (Note that short-floats are used; this results in the most efficient color conversion process. However, any floating-point number 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 only supported on Cocoa and Windows.

The predicate **color-spec-p** can be used to test for color-spec objects. The function **color-spec-model** returns the model in which a color-spec object has been defined.

14.4 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 *LispWorks CAPI Reference Manual* entry 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 Section 14.3 on page 167, 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)
 "")
```

14.5 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.

Table 14.1 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)

Table 14.1 Color models defined by default

Model	Name	Component: Range
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 3.0 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.

Conversion to pixel values is done by **convert-color**.

14.6 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"))
```

To clear the color 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**.

14.7 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**.

15

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 and is not discussed here.

15.1 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**.

15.1.1 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.

15.2 Printer definition files

On GTK+ and 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 “PPD files” on page 174 for information about PPD files). When a user saves a printer configuration, the system writes such 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.

15.3 PPD files

To fully use the functionality of a Postscript printer on GTK+ and 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.

15.4 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.

15.5 Handling pages—page on demand printing

In *Page on Demand Printing,* the application provides code to print 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 iterates over all pages in the document.

15.6 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 prints each page of the document in order. Page on Demand printing uses the **`with-page`** macro, with each invocation of the **`with-page`** macro contributing a new page to the document.

15.7 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.

15.7.1 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 **examples/capi/printing/fit-to-page.lisp**.

15.8 Other printing functions

On GTK+ and Motif, printers can be added, removed and configured via **printer-configuration-dialog**. On Microsoft Windows, the equivalent functionality is on 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 provides a means of printing plain text to a printer. The functions **print-text**, **print-file** and **print-editor-buffer** can be used for this.

16

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`**.

16.1 Overview of drag and drop in CAPI

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.

16.1.1 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 Mac OS X Finder, and vice versa.

You can also specify private data formats that work only within the same CAPI application image.

16.2 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.

16.2.1 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.

16.2.1.1 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

```
examples/capi/choice/drag-and-drop.lisp
```

16.2.2 Dragging values from an output-pane

To implement dragging in 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. See the example file in

```
examples/capi/output-panes/drag-and-drop.lisp
```

Note: If you implement dragging of text in an **editor-pane**, you will use EDITOR functions such as **editor:points-to-string** to obtain the value for the **:string** format.

16.2.3 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 a **gp:image**. See "Working with images" on page 156 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 **gp: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 **gp: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 explictly if freeing can wait (which is probably true in most cases).

See the example in **examples/capi/choice/list-panel-drag-image.lisp**

:filename-list

Receives a list of files. Is understood by other applications such as the Mac OS X Finder and Windows Explorer.

You can also use private formats, named by arbitrary keywords, which will work only in the same Lisp image.

16.2.4 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.

16.3 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 was made available.

16.3.1 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 with 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.

16.3.2 Dropping in a choice

Addtionally 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.

16.3.2.1 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)))
    (: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 “Example: dragging from a tree” on page 178.

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))))))
```

16.3.3 Dropping text in an editor-pane

Supply the special *drop-callback* **:default** to implement dropping text in an **editor-pane**.

16.3.4 Dropping in an output-pane

Addtionally 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.

16.4 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.

Index

E

F

G

M

N

O

P

Q

R

S

T