Common Lisp Interactive Objects

Kerry Kimbrough
Suzanne McBride
Lars Greninger

Texas Instruments Incorporated

Version 1.0
July, 1990

Contents

Chapter 1

Introduction

1.1 Overview

Common Lisp Interactive Objects (CLIO) is a set of CLOS classes that represent the standard components of an object-oriented user interface — such as text, menus, buttons, scroller, and dialogs. CLIO is designed to be a portable system written in Common Lisp and is based on other standard Common Lisp interfaces:

- CLX[5], the Common Lisp interface to the X Window System;
- CLUE[2], a portable Common Lisp user interface toolkit; and
- CLOS[1], the ANSI-standard Common Lisp Object System.

CLIO not only provides the basic components commonly used in constructing graphical user interfaces, but also specifies an application progam interface that is **look-and-feel independent**. That is, an application program can rely on the functional behavior of CLIO components without depending on the details of visual appearance and event handling.

CLIO components are those whose "look-and-feel" is typically specified by a comprehensive user interface **style guide**. A style guide describes a consistent, identifiable style shared by all application programs. OPEN LOOK[1] and Motif[2] are examples of such look-and-feel style guides. The CLIO interface is designed to support implementations that conform to these and other style guides.

The concept of look-and-feel independence means that the look-and-feel of CLIO components is encapsulated within the implementation of the CLIO interface. An application program can be ported to a different style guide simply by using a different implementation of the CLIO "library."

[1] OPEN LOOK is a trademark of AT&T
[2] Motif is a trademark of the Open Software Foundation

1.2 Summary of Features

The toolkit "intrinsics" used by CLIO are defined by CLUE. Thus, CLIO defines the application programmer interface to a set of **contact** classes — constructor functions, accessor functions, **resources**, and **callback** interfaces. See [2] for a complete description of contacts, callbacks, and resources.

CLIO does *not* define functions or mechanisms that control the handling of user input events. Such functions are related to the "feel" of a specific style guide and are thus implementation-dependent.

The types of classes defined by CLIO include text, images, controls, dialogs, and containers. All CLIO classes are subclass of a common base class — the `core` class.

1.2.1 Text

The `display-text` class represents text that is displayed but cannot be modified interactively. The text displayed is given by a source string, which may contain `#\newline` characters to indicate multiple lines. A `display-text` has a number of attributes — such as font, alignment, and margins — that control its presentation. A `display-text` also supplies operations which allow a user to select portions of the source string. The standard conventions specified by the X Window System Inter-Client Communications Convention Manual (ICCCM)[3] are used to support interchange of selected text.

The `edit-text` class represents text that can be interactively selected, deleted, or modified by a user. An `edit-text` shares the same presentation attributes as a `display-text`.

Additional classes — `display-text-field` and `edit-text-field` — are defined to provide efficient support for the common cases of single-line text fields.

1.2.2 Images

A **display-image** presents an array of pixels for viewing. The source array of a `display-image` may be either a `pixmap` or an `image` object. A `display-image` has several attributes — such as margins and gravity — that control its presentation.

1.2.3 Controls

CLIO **controls** are used to select or modify values that control the application or other parts of its user interface. CLIO contains four different types of controls: buttons, scales, items, and choices.

Buttons represent "switches" used to initate actions or modify values. An **action-button** allows a user to immediately invoke an action. A **toggle-button** represents a two-state switch which a user may turn "on" or "off." A button has a label which may be either a text string or a `pixmap`. A **dialog-button** is a specialized `action-button` which allows a user to immediately present a dialog, such as a menu.

A **scale** is used to present a numerical value for viewing and modification. CLIO scales include sliders and scrollers. A **scroller** is a scale which plays a specific user interface role — changing the viewing position of another user interface object. A `slider` has the same functional interface as a `scroller`, but it has a less specific role and typically has a different appearance and behavior. Sliders and scrollers may have either a horizontal or a vertical orientation.

Item classes represent objects which can appear as selectable items in menus. The item types defined by CLIO include **action-items**, which invoke an immediate action, and **dialog-items**, which display another menu or another type of dialog.

A **choice** contact is a composite contact used to contain a set of **choice items**. A choice contact allows a user to choose zero or more of the choice items which are its children. In order to operate correctly as a choice item, a child contact need not belong to any specific class, but it must obey a certain **choice item protocol**. Choice classes in CLIO include **choices** and **multiple-choices**.

1.2.4 Dialogs

Dialog classes are `shell`[3] subclasses used to display a set of application data and to report a user's response. CLIO defines dialogs for various types of user interactions. A **confirm** is a simple dialog which presents a message and allows a user to enter a "yes or no" response. A **menu** allows a user to select from a set of choice items. A **property-sheet** presents a set of related values for editing and allows a user to accept or cancel any changes. The most general type of dialog is a **command**, which presents not only a set of related value controls but also a set of application-defined controls which operate on the values.

1.2.5 Containers

A **container** is a composite contact used to manage a set of child contacts. For example, a **scroll-frame** is a CLIO container which contains a child called the "content" and which allows a user to view different parts of the content by manipulating horizontal and/or vertical scrolling controls. The scrolling controls are implemented by `scroller` contacts and are created automatically.

Some container classes are referred to as **layouts**. A layout is a type of container whose purpose is limited to providing a specific style of geometry management. Examples of CLIO layouts include forms and tables. A **form** manages the geometry of a set of children (or **members**) according to a set of constraints. Geometrical constraints are used to define the minimum/maximum size for each member, as well as the ideal/maximum/minimum space between members. A **table** arranges its members into an array of rows and columns. Row/column positions are defined as constraint resources of individual members

[3]For a complete discussion of shells, see [2].

1.3 Packages

All CLIO function symbols are assumed to be exported from a single package that represents an implementation of CLIO for a specific style guide. However, the name of the package which exports CLIO symbols is implementation-dependent.

1.4 Core Contacts

All CLIO contact classes are subclasses of the `core` class. The `core` class represents exactly those features common to the all CLIO classes. In general, the implementation of the `core` class also represents the characteristics shared by all component in a specific style guide. Functionally, the `core` class is defined by the accessor methods described below. In addition, `core` defines initargs which may be given to any of the CLIO constructor functions.

contact-foreground **Method, core**

```
(defmethod  contact-foreground
            ((core core))
(declare    (values pixel)))


(defmethod  (setf contact-foreground)
            (foreground
            (core core))
(declare    (values pixel)))
```

Return or change the foreground pixel used for output to a `core` contact. When changing the foreground pixel, `convert` is called to convert the new value to a pixel value, if necessary.

The `:foreground` initarg can be given to any CLIO constructor function to initialize the foreground pixel. By default, the initial foreground pixel for a `core` object is the same as its parent's. If the parent is not a `core` contact (for example, if the parent is a `root`), then the default initial foreground pixel is given by `*default-contact-foreground*`.

contact-border **Method, core**

```
(defmethod  contact-border
            ((core core))
(declare    (values (or (member :copy) pixel pixmap))))


(defmethod  (setf contact-border)
            (border
            (core core))
(declare    (values (or (member :copy) pixel pixmap))))
```

Return or change the contents of the border of a core contact. When changing the border, convert is called to convert the new value to a valid value, if necessary.

The :border initarg can be given to any CLIO constructor function to initialize the contact border. By default, the border for a core object is given by *default-contact-border*.

Chapter 2

Text

2.1 Display Text

A `display-text` presents multiple lines of text for viewing.

The source of a `display-text` is a text string. The following functions may be used to control the presentation of the displayed string.

- `display-gravity`
- `display-text-alignment`
- `display-text-font`

The following functions may be used to set the margins surrounding the displayed text.

- `display-bottom-margin`
- `display-left-margin`
- `display-right-margin`
- `display-top-margin`

2.1.1 Selecting and Copying Text

A `display-text` allows a user to interactively select source text and to transfer selected text strings using standard conventions for interclient communication (see Section 8.3). The specific interactive operations used to select and transfer text are implementation-dependent.

Selecting text causes a `display-text` to become the owner of the `:primary` selection, which then contains the selected text. The `display-text-selection` function may be used to return the currently-selected text. Copying selected text — accomplished by user interaction or by calling the `display-text-copy` function — causes the selected text to become the value of the `:clipboard` selection.

A `display-text` can handle requests from other clients to convert the `:primary` and `:clipboard` selections to the target types defined by the following atoms. (See [3] for a complete description of conventions for text target atoms.)

- The font character encoding atom. This is an atom identifying the character encoding used by the `display-text-font`.
- `:text`. This is equivalent to using the font character encoding atom.
- All other target atoms required by ICCCM. See Section 8.3.

2.1.2 Functional Definition

make-display-text **Function**

```
(defun    make-display-text
          (&rest initargs
          &key
          (alignment              :left)
          (border                 *default-contact-border*)
          (bottom-margin          :default)
          (display-gravity        :center)
          (font                   *default-display-text-font*)
          foreground
          (left-margin            :default)
          (right-margin           :default)
          (top-margin             :default)
          (source                 "")
          &allow-other-keys)
(declare  (values display-text)))
```

Creates and returns a `display-text` contact. The resource specification list of the `display-text` class defines a resource for each of the initargs above.

display-bottom-margin **Method, display-text**

```
(defmethod  display-bottom-margin
            ((display-text display-text))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-bottom-margin)
            (bottom-margin
            (display-text display-text))
(declare    (type (or (integer 0 *) :default) bottom-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the bottom margin. The height of the contact minus the bottom margin size defines the bottom edge of the clipping rectangle used when displaying the source. Setting the bottom margin to `:default` causes the value of `*default-display-bottom-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-text-alignment — Method, display-text

```
(defmethod  display-text-alignment
            ((display-text display-text))
(declare    (values (member :left :center :right))))
```

```
(defmethod  (setf display-text-alignment)
            (alignment
            (display-text display-text))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns or changes the horizontal alignment of text lines with respect to the bounding rectangle of the source.

`:left`	Lines are left-justified within the source bounding rectangle.
`:right`	Lines are right-justified within the source bounding rectangle.
`:center`	Lines are centered within the source bounding rectangle.

display-text-copy — Method, display-text

```
(defmethod  display-text-copy
            ((display-text display-text))
(declare    (values (or null sequence))))
```

Causes the currently selected text to become the current value of the `:clipboard` selection. The currently selected text is returned.

display-gravity **Method, display-text**

```
(defmethod  display-gravity
            ((display-text display-text))
(declare    (values gravity)))
```

```
(defmethod  (setf display-gravity)
            (gravity
            (display-text display-text))
(declare    (type gravity gravity))
(declare    (values gravity)))
```

Returns or changes the display gravity of the contact. See Section 8.2. Display gravity controls the alignment of the source bounding rectangle with respect to the clipping rectangle formed by the top, left, bottom, and right margins. The display gravity determines the source position that is aligned with the corresponding position of the margin rectangle.

display-left-margin **Method, display-text**

```
(defmethod  display-left-margin
            ((display-text display-text))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-left-margin)
            (left-margin
            (display-text display-text))
(declare    (type (or (integer 0 *) :default) left-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the left margin. The left margin size defines the left edge of the clipping rectangle used when displaying the source. Setting the left margin to `:default` causes the value of `*default-display-left-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-right-margin **Method, display-text**

```
(defmethod  display-right-margin
            ((display-text display-text))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-right-margin)
            (right-margin
            (display-text display-text))
(declare    (type (or (integer 0 *) :default) right-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the right margin. The width of the contact minus the right margin size defines the right edge of the clipping rectangle used when displaying the source. Setting the right margin to `:default` causes the value of `*default-display-right-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-text-font — Method, display-text

```
(defmethod  display-text-font
            ((display-text display-text))
(declare    (values font)))
```

```
(defmethod  (setf display-text-font)
            (font
            (display-text display-text))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification. Together with the `display-text-source`, this determines the `font` object used to display source characters.

display-text-selection — Method, display-text

```
(defmethod  display-text-selection
            ((display-text display-text))
(declare    (values (or null sequence))))
```

Returns a string containing the currently selected text (or `nil` if no text is selected).

display-text-source **Method, display-text**

```
(defmethod  display-text-source
            ((display-text display-text)
            &key
            (start 0)
            end)
(declare    (type (integer 0 *)            start)
            (type (or (integer 0 *) null)  end))
(declare    (values string)))
```

```
(defmethod  (setf display-text-source)
            (new-source
            (display-text display-text)
            &key
            (start 0)
            end
            (from-start 0)
            from-end)
(declare    (type stringable new-source)
            (type (integer 0 *)            start from-start)
            (type (or (integer 0 *) null)  end from-end))
(declare    (values string)))
```

Returns or changes the string displayed. As in `common-lisp:subseq`, the `start` and `end` arguments specify the substring returned or changed.

When changing the displayed string, the `from-start` and `from-end` arguments specify the substring of the `new-source` argument that replaces the source substring given by `start` and `end`.

display-top-margin **Method, display-text**

```
(defmethod  display-top-margin
            ((display-text display-text))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-top-margin)
            (top-margin
            (display-text display-text))
(declare    (type (or (integer 0 *) :default) top-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the top margin. The top margin size defines the top edge of the clipping rectangle used when displaying the source. Setting the top margin to `:default` causes the value of `*default-display-top-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

2.2 Display Text Field

A display-text-field presents a single line of text for viewing. The functional interface for display-text-field objects is much the same as that for display-text objects. However, some presentation attributes are not appropriate for single-line text (display-text-alignment, for example). Also, the implementation of a display-text-field may be simpler and more efficient.

2.2.1 Functional Definition

make-display-text-field **Function**

```
(defun    make-display-text-field
          (&rest initargs
          &key
          (border                      *default-contact-border*)
          (bottom-margin               :default)
          (display-gravity             :center)
          (font                        *default-display-text-font*)
          foreground
          (left-margin                 :default)
          (right-margin                :default)
          (top-margin                  :default)
          (source                      "")
          &allow-other-keys)
(declare  (values display-text-field)))
```

Creates and returns a display-text-field contact. The resource specification list of the display-text-field class defines a resource for each of the initargs above.

display-bottom-margin **Method, display-text-field**

```
(defmethod  display-bottom-margin
            ((display-text-field display-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-bottom-margin)
            (bottom-margin
            (display-text-field display-text-field))
(declare    (type (or (integer 0 *) :default) bottom-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the bottom margin. The height of the contact minus the bottom margin size defines the bottom edge of the clipping rectangle used when displaying the source. Setting the bottom margin to `:default` causes the value of `*default-display-bottom-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-gravity **Method, display-text-field**

```
(defmethod  display-gravity
            ((display-text-field display-text-field))
(declare    (values gravity)))
```

```
(defmethod  (setf display-gravity)
            (gravity
            (display-text-field display-text-field))
(declare    (type gravity gravity))
(declare    (values gravity)))
```

Returns or changes the display gravity of the contact. See Section 8.2. Display gravity controls the alignment of the source bounding rectangle with respect to the clipping rectangle formed by the top, left, bottom, and right margins. The display gravity determines the source position that is aligned with the corresponding position of the margin rectangle.

display-left-margin **Method, display-text-field**

```
(defmethod  display-left-margin
            ((display-text-field display-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-left-margin)
            (left-margin
            (display-text-field display-text-field))
(declare    (type (or (integer 0 *) :default) left-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the left margin. The left margin size defines the left edge of the clipping rectangle used when displaying the source. Setting the left margin to `:default` causes the value of `*default-display-left-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-right-margin **Method, display-text-field**

```
(defmethod  display-right-margin
            ((display-text-field display-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-right-margin)
            (right-margin
            (display-text-field display-text-field))
(declare    (type (or (integer 0 *) :default) right-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the right margin. The width of the contact minus the right margin size defines the right edge of the clipping rectangle used when displaying the source. Setting the right margin to `:default` causes the value of `*default-display-right-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-text-font **Method, display-text-field**

```
(defmethod  display-text-font
            ((display-text-field display-text-field))
(declare    (values font)))
```

```
(defmethod  (setf display-text-font)
            (font
            (display-text-field display-text-field))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification. Together with the `display-text-source`, this determines the `font` object used to display source characters.

display-text-source **Method, display-text-field**

```
(defmethod  display-text-source
            ((display-text-field display-text-field)
            &key
            (start 0)
            end)
(declare    (type (integer 0 *)                         start)
            (type (or (integer 0 *) null)               end))
(declare    (values string)))
```

```
(defmethod  (setf display-text-source)
            (new-source
            (display-text-field display-text-field)
            &key
            (start 0)
            end
            (from-start 0)
            from-end)
(declare    (type stringable new-source)
            (type (integer 0 *)                        start from-start)
            (type (or (integer 0 *) null)              end from-end))
(declare    (values string)))
```

Returns or changes the string displayed. As in `common-lisp:subseq`, the `start` and `end` arguments specify the substring returned or changed.

When changing the displayed string, the `from-start` and `from-end` arguments specify the substring of the `new-source` argument that replaces the source substring given by `start` and `end`.

display-top-margin **Method, display-text-field**

```
(defmethod  display-top-margin
            ((display-text-field display-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-top-margin)
            (top-margin
            (display-text-field display-text-field))
(declare    (type (or (integer 0 *) :default) top-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the top margin. The top margin size defines the top edge of the clipping rectangle used when displaying the source. Setting the top margin to `:default` causes the value of `*default-display-top-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

2.3 Edit Text

An `edit-text` presents multiple lines of text for viewing and editing.

The source of an `edit-text` is a text string. The **point** defines the index in the source where characters entered by a user will be inserted. The `edit-text-clear` function may be used to make the source empty.

The `:complete` callback is invoked when a user signals that text editing is complete. The `:point` callback is invoked whenever a user changes the insert point. `:insert` and `:delete` callbacks may be defined to validate each change to the source made by the user. The `:verify` callback is called before completion to validate the final source string. `:suspend` and `:resume` callbacks are invoked when user text editing is suspended and resumed.

Displayed source text may be selected by a user for interclient data transfer. The `display-text-selection` function may be used to return the currently-selected text. The current text selection is defined to be the source (sub)string between the point and the **mark**. An application program may change the point and the mark to change the current text selection. An `edit-text` provides methods that allow a user to transfer selected text using standard conventions for interclient communication.

The following functions may be used to control the presentation of the text source.

- `display-gravity`
- `display-text-alignment`
- `display-text-font`

The following functions may be used to set the margins surrounding the text source.

- `display-bottom-margin`
- `display-left-margin`
- `display-right-margin`
- `display-top-margin`

2.3.1 Selecting, Copying, Cutting, and Pasting Text

An `edit-text` allows a user to interactively select source text and to transfer selected text strings using standard conventions for interclient communication (see Section 8.3). The specific interactive operations used to select and transfer text are implementation-dependent.

Selecting text causes an edit-text to become the owner of the :primary selection, which then contains the selected text. The display-text-selection function may be used to return the currently-selected text. Copying or cutting selected text causes the selected text to become the value of the :clipboard selection.

The display-text-copy function causes the selected text to become the current value of the :clipboard selection. The edit-text-cut function causes the selected text to be deleted and the deleted text to become the value of the :clipboard selection. The edit-text-paste function causes the current contents of the :clipboard selection to be inserted into the source.

An edit-text can handle requests from other clients to convert the :primary and :clipboard selections to the target types defined by the following atoms. (See [3] for a complete description of conventions for text target atoms.)

- The font character encoding atom. This is an atom identifying the character encoding used by the display-text-font.
- :text. This is equivalent to using the font character encoding atom.
- All other target atoms required by ICCCM. See Section 8.3.

2.3.2 Functional Definition

make-edit-text **Function**

```
(defun    make-edit-text
          (&rest initargs
          &key
          (alignment          :left)
          (border             *default-contact-border*)
          (bottom-margin      :default)
          (display-gravity    :north-west)
          (font               *default-display-text-font*)
          foreground
          (left-margin        :default)
          mark
          point
          (right-margin       :default)
          (source             "")
          (top-margin         :default)
          &allow-other-keys)
(declare  (values edit-text)))
```

Creates and returns a edit-text contact. The resource specification list of the edit-text class defines a resource for each of the initargs above.

display-text-alignment **Method, edit-text**

```
(defmethod  display-text-alignment
            ((edit-text edit-text))
(declare    (values (member :left :center :right))))
```

```
(defmethod  (setf display-text-alignment)
            (alignment
            (edit-text edit-text))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns or changes the horizontal alignment of text lines with respect to the bounding rectangle of the source.

`:left`	Lines are left-justified within the source bounding rectangle.
`:right`	Lines are right-justified within the source bounding rectangle.
`:center`	Lines are centered within the source bounding rectangle.

display-bottom-margin **Method, edit-text**

```
(defmethod  display-bottom-margin
            ((edit-text edit-text))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-bottom-margin)
            (bottom-margin
            (edit-text edit-text))
(declare    (type (or (integer 0 *) :default) bottom-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the bottom margin. The height of the contact minus the bottom margin size defines the bottom edge of the clipping rectangle used when displaying the source. Setting the bottom margin to `:default` causes the value of `*default-display-bottom-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-gravity **Method, edit-text**

```
(defmethod  display-gravity
            ((edit-text edit-text))
(declare    (values gravity)))


(defmethod  (setf display-gravity)
            (gravity
            (edit-text edit-text))
(declare    (type gravity gravity))
(declare    (values gravity)))
```

Returns or changes the display gravity of the contact. See Section 8.2. Display gravity controls the alignment of the source bounding rectangle with respect to the clipping rectangle formed by the top, left, bottom, and right margins. The display gravity determines the source position that is aligned with the corresponding position of the margin rectangle.

display-left-margin **Method, edit-text**

```
(defmethod  display-left-margin
            ((edit-text edit-text))
(declare    (values (integer 0 *))))


(defmethod  (setf display-left-margin)
            (left-margin
            (edit-text edit-text))
(declare    (type (or (integer 0 *) :default) left-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the left margin. The left margin size defines the left edge of the clipping rectangle used when displaying the source. Setting the left margin to `:default` causes the value of `*default-display-left-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-right-margin **Method, edit-text**

```
(defmethod  display-right-margin
            ((edit-text edit-text))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-right-margin)
            (right-margin
            (edit-text edit-text))
(declare    (type (or (integer 0 *) :default) right-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the right margin. The width of the contact minus the right margin size defines the right edge of the clipping rectangle used when displaying the source. Setting the right margin to `:default` causes the value of `*default-display-right-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-text-copy — Method, edit-text

```
(defmethod  display-text-copy
            ((edit-text edit-text))
(declare    (values (or null string))))
```

Causes the currently selected text to become the current value of the `:clipboard` selection. The currently selected text is returned.

display-text-font — Method, edit-text

```
(defmethod  display-text-font
            ((edit-text edit-text))
(declare    (values font)))
```

```
(defmethod  (setf display-text-font)
            (font
            (edit-text edit-text))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification. Together with the `display-text-source`, this determines the `font` object used to display source characters.

display-text-selection — Method, edit-text

```
(defmethod  display-text-selection
            ((edit-text edit-text))
(declare    (values (or null string))))
```

Returns a string containing the currently selected text (or `nil` if no text is selected).

display-text-source — Method, edit-text

```
(defmethod  display-text-source
            ((edit-text edit-text)
            &key
            (start 0)
            end)
(declare    (type (integer 0 *)             start)
            (type (or (integer 0 *) null)   end))
(declare    (values string)))
```

```
(defmethod  (setf display-text-source)
            (new-source
            (edit-text edit-text)
            &key
            (start 0)
            end
            (from-start 0)
            from-end)
(declare    (type stringable new-source)
            (type (integer 0 *)             start from-start)
            (type (or (integer 0 *) null)   end from-end))
(declare    (values string)))
```

Returns or changes the string displayed. As in `common-lisp:subseq`, the `start` and `end` arguments specify the substring returned or changed.

When changing the displayed string, the `from-start` and `from-end` arguments specify the substring of the `new-source` argument that replaces the source substring given by `start` and `end`.

display-top-margin — Method, edit-text

```
(defmethod  display-top-margin
            ((edit-text edit-text))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-top-margin)
            (top-margin
            (edit-text edit-text))
(declare    (type (or (integer 0 *) :default) top-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the top margin. The top margin size defines the top edge of the clipping rectangle used when displaying the source. Setting the top margin to `:default` causes the value of `*default-display-top-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

edit-text-clear — Method, edit-text

```
(defmethod  edit-text-clear
            ((edit-text edit-text)))
```

Sets the source to the empty string.

edit-text-cut — Method, edit-text

```
(defmethod  edit-text-cut
            ((edit-text edit-text))
(declare    (values (or null string))))
```

Causes the selected text to be deleted and the deleted text to become the value of the `:clipboard` selection. Returns the deleted text, if any.

edit-text-mark — Method, edit-text

```
(defmethod  edit-text-mark
            ((edit-text edit-text))
(declare    (values (or null (integer 0 *)))))
```

```
(defmethod  (setf edit-text-mark)
            (mark
            (edit-text edit-text))
(declare    (type (or null (integer 0 *)) mark))
(declare    (values (or null (integer 0 *)))))
```

Returns or changes an index in the source used to access the currently-selected text. If `nil`, no text is selected. Otherwise, the selected text is defined to be the source substring between the mark and the point. In order to avoid surprising the user, an application program should change the mark only in response to some user action.

edit-text-paste — Method, edit-text

```
(defmethod  edit-text-paste
            ((edit-text edit-text))
(declare    (values (or null string))))
```

Causes the current contents of the `:clipboard` selection to be inserted into the source. Returns the inserted text, if any.

edit-text-point **Method, edit-text**

```
(defmethod  edit-text-point
            ((edit-text edit-text))
(declare    (values (or null (integer 0 *)))))

(defmethod  (setf edit-text-point)
            (point
            (edit-text edit-text)
            &key
            clear-p)
(declare    (type (or null (integer 0 *))      point)
            (type boolean                      clear-p))
(declare    (values (or null (integer 0 *)))))
```

Returns or changes the index in the source where characters entered by a user will be inserted. When changing the point, if `clear-p` is true, then the current text selection is cleared — that is, the mark is also set to the new point position. In order to avoid surprising the user, an application program should change the point only in response to some user action.

2.3.3 Callbacks

:complete **Callback, edit-text**

```
(defun  complete-function  ())
```

Invoked when the user indicates that all source modifications are complete. The `:verify` callback is always invoked before invoking the `:complete` callback. The `:complete` callback is called only if `:verify` returns true.

:delete **Callback, edit-text**

```
(defun    delete-function
          (edit-text
          start
          end)
(declare  (type edit-text edit-text)
          (type (or null (integer 0 *)) start end))
(declare  (values (or null (integer 0 *)) (or null (integer 0 *)))))
```

Invoked when the user deletes one or more source characters. The deleted substring is defined by the `start` and `end` indices. This callback allows an application to protect source fields from deletion or adjust the deleted text. The return values indicate the start and end indices of the source characters that should actually be deleted.

:insert **Callback, edit-text**

```
(defun    insert-function
          (edit-text
          start
          inserted)
(declare  (type edit-text edit-text)
          (type (integer 0 *) start)
          (type (or character string) inserted))
(declare  (values (or null (integer 0 *)) (or character string))))
```

Invoked when the user inserts one or more source characters. The inserted substring is defined by the `start` index and the `inserted` string or character. This callback allows an application to protect source fields from insertion or to adjust the inserted text. The return values indicate the start index and characters of the string actually inserted. If a `nil` start index is returned, then the insertion is not allowed.

:point **Callback, edit-text**

```
(defun    point-function
          (edit-text
          new-position)
(declare  (type edit-text edit-text)
          (type (or null (integer 0 *)) new-position)))
```

Invoked when the user explicitly changes the position of the point. This does not include implicit changes caused by inserting or deleting text.

:resume **Callback, edit-text**

```
(defun  resume-function  ())
```

Invoked when the user resumes editing on the `edit-text`. For example, this callback is usually invoked when the `edit-text` becomes the keyboard focus.

:suspend **Callback, edit-text**

```
(defun  suspend-function  ())
```

Invoked when the user suspends editing on the `edit-text`. For example, this callback is usually invoked when the `edit-text` ceases to be the keyboard focus.

:verify **Callback, edit-text**

```
(defun     verify-function
           (edit-text)
(declare   (type edit-text edit-text))
(declare   (values boolean string)))
```

Invoked when the user requests validation of the modified source. This callback allows an application to enforce constraints on the contents of the source. If the first return value is true, then the source satisfies application constraints. Otherwise, the second value is a string containing an error message to be displayed.

This callback is always invoked before invoking the `:complete` callback. The `:complete` callback is called only if `:verify` returns true.

2.4 Edit Text Field

An `edit-text-field` presents a single line of text for viewing and editing. An `edit-text-field` may optionally define a maximum number of characters for its source string.

Displayed source text may be selected by a user for interclient data transfer. An `edit-text-field` provides the same functions as an `edit-text` for selecting, copying, cutting, and pasting text (see Section 2.3.1).

The functional interface for `edit-text-field` objects is much the same as that for `edit-text` objects. However, some attributes are not appropriate for single-line text (`display-text-alignment`, for example). Also, the implementation of a `edit-text-field` may be simpler and more efficient.

2.4.1 Functional Definition

make-edit-text-field **Function**

```
(defun    make-edit-text-field
          (&rest initargs
          &key
          (border                   *default-contact-border*)
          (bottom-margin            :default)
          (display-gravity          :west)
          (font                     *default-display-text-font*)
          foreground
          (left-margin              :default)
          length
          mark
          point
          (right-margin             :default)
          (source                   "")
          (top-margin               :default)
          &allow-other-keys)
(declare  (values edit-text-field)))
```

Creates and returns a `edit-text-field` contact. The resource specification list of the `edit-text-field` class defines a resource for each of the initargs above.

display-bottom-margin **Method, edit-text-field**

```
(defmethod  display-bottom-margin
            ((edit-text-field edit-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-bottom-margin)
            (bottom-margin
            (edit-text-field edit-text-field))
(declare    (type (or (integer 0 *) :default) bottom-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the bottom margin. The height of the contact minus the bottom margin size defines the bottom edge of the clipping rectangle used when displaying the source. Setting the bottom margin to `:default` causes the value of `*default-display-bottom-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-gravity **Method, edit-text-field**

```
(defmethod  display-gravity
            ((edit-text-field edit-text-field))
(declare    (values gravity)))
```

```
(defmethod  (setf display-gravity)
            (gravity
            (edit-text-field edit-text-field))
(declare    (type gravity gravity))
(declare    (values gravity)))
```

Returns or changes the display gravity of the contact. See Section 8.2. Display gravity controls the alignment of the source bounding rectangle with respect to the clipping rectangle formed by the top, left, bottom, and right margins. The display gravity determines the source position that is aligned with the corresponding position of the margin rectangle.

display-left-margin **Method, edit-text-field**

```
(defmethod  display-left-margin
            ((edit-text-field edit-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-left-margin)
            (left-margin
            (edit-text-field edit-text-field))
(declare    (type (or (integer 0 *) :default) left-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the left margin. The left margin size defines the left edge of the clipping rectangle used when displaying the source. Setting the left margin to `:default` causes the value of `*default-display-left-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-right-margin **Method, edit-text-field**

```
(defmethod  display-right-margin
            ((edit-text-field edit-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-right-margin)
            (right-margin
            (edit-text-field edit-text-field))
(declare    (type (or (integer 0 *) :default) right-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the right margin. The width of the contact minus the right margin size defines the right edge of the clipping rectangle used when displaying the source. Setting the right margin to `:default` causes the value of `*default-display-right-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-text-copy **Method, edit-text-field**

```
(defmethod  display-text-copy
            ((edit-text-field edit-text-field))
(declare    (values (or null string))))
```

Causes the currently selected text to become the current value of the `:clipboard` selection. The currently selected text is returned.

display-text-font **Method, edit-text-field**

```
(defmethod  display-text-font
            ((edit-text-field edit-text-field))
(declare    (values font)))
```

```
(defmethod  (setf display-text-font)
            (font
            (edit-text-field edit-text-field))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification. Together with the `display-text-source`, this determines the `font` object used to display source characters.

display-text-selection **Method, edit-text-field**

```
(defmethod  display-text-selection
            ((edit-text-field edit-text-field))
(declare    (values (or null string))))
```

Returns a string containing the currently selected text (or `nil` if no text is selected).

display-text-source **Method, edit-text-field**

```
(defmethod  display-text-source
            ((edit-text-field edit-text-field)
            &key
            (start 0)
            end)
(declare    (type (integer 0 *)                start)
            (type (or (integer 0 *) null)      end))
(declare    (values string)))
```

```
(defmethod  (setf display-text-source)
            (new-source
            (edit-text-field edit-text-field)
            &key
            (start 0)
            end
            (from-start 0)
            from-end)
(declare    (type stringable new-source)
            (type (integer 0 *)               start from-start)
            (type (or (integer 0 *) null)     end from-end))
(declare    (values string)))
```

Returns or changes the string displayed. As in `common-lisp:subseq`, the `start` and `end` arguments specify the substring returned or changed.

When changing the displayed string, the `from-start` and `from-end` arguments specify the substring of the `new-source` argument that replaces the source substring given by `start` and `end`.

display-top-margin **Method, edit-text-field**

```
(defmethod  display-top-margin
            ((edit-text-field edit-text-field))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-top-margin)
            (top-margin
            (edit-text-field edit-text-field))
(declare    (type (or (integer 0 *) :default) top-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the top margin. The top margin size defines the top edge of the clipping rectangle used when displaying the source. Setting the top margin to `:default` causes the value of `*default-display-top-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

edit-text-clear — Method, edit-text-field

```
(defmethod  edit-text-clear
            ((edit-text-field edit-text-field)))
```

Sets the source to the empty string.

edit-text-cut — Method, edit-text-field

```
(defmethod  edit-text-cut
            ((edit-text-field edit-text-field))
(declare    (values (or null string))))
```

Causes the selected text to be deleted and the deleted text to become the value of the `:clipboard` selection. Returns the deleted text, if any.

edit-text-field-length — Method, edit-text-field

```
(defmethod  edit-text-field-length
            ((edit-text-field edit-text-field))
(declare    (values (or null (integer 0 *)))))
```

```
(defmethod  (setf edit-text-field-length)
            (length
            (edit-text-field edit-text-field))
(declare    (type (or null (integer 0 *)) length))
(declare    (values (or null (integer 0 *)))))
```

Returns or changes the maximum number of characters allowed in the source string of the `edit-text-field`. If `nil`, then the source can be any length.

edit-text-mark **Method, edit-text-field**

```
(defmethod  edit-text-mark
            ((edit-text-field edit-text-field))
(declare    (values (or null (integer 0 *)))))
```

```
(defmethod  (setf edit-text-mark)
            (mark
            (edit-text-field edit-text-field))
(declare    (type (or null (integer 0 *)) mark))
(declare    (values (or null (integer 0 *)))))
```

Returns or changes an index in the source used to access the currently-selected text. If `nil`, no text is selected. Otherwise, the selected text is defined to be the source substring between the mark and the point. In order to avoid surprising the user, an application program should change the mark only in response to some user action.

edit-text-paste **Method, edit-text-field**

```
(defmethod  edit-text-paste
            ((edit-text-field edit-text-field))
(declare    (values (or null string))))
```

Causes the current contents of the `:clipboard` selection to be inserted into the source. Returns the inserted text, if any.

edit-text-point **Method, edit-text-field**

```
(defmethod  edit-text-point
            ((edit-text-field edit-text-field))
(declare    (values (or null (integer 0 *)))))
```

edit-text-point **Method, edit-text-field**

```
(defmethod  edit-text-point
            ((edit-text-field edit-text-field))
(declare    (values (or null (integer 0 *)))))
```

```
(defmethod  (setf edit-text-point)
            (point
            (edit-text-field edit-text-field)
            &key
            clear-p)
(declare    (type (or null (integer 0 *))      point)
            (type boolean                      clear-p))
(declare    (values (or null (integer 0 *)))))
```

Returns or changes the index in the source where characters entered by a user will be inserted. When changing the point, if `clear-p` is true, then the current text selection is cleared — that is, the mark is also set to the new point position. In order to avoid surprising the user, an application program should change the point only in response to some user action.

2.4.2 Callbacks

:complete **Callback, edit-text-field**

```
(defun  complete-function  ())
```

Invoked when the user indicates that all source modifications are complete. The `:verify` callback is always invoked before invoking the `:complete` callback. The `:complete` callback is called only if `:verify` returns true.

:delete **Callback, edit-text-field**

```
(defun     delete-function
           (edit-text-field
           start
           end)
(declare   (type edit-text-field edit-text-field)
           (type (or null (integer 0 *)) start end))
(declare   (values (or null (integer 0 *)) (or null (integer 0 *)))))
```

Invoked when the user deletes one or more source characters. The deleted substring is defined by the `start` and `end` indices. This callback allows an application to protect source fields from deletion or adjust the deleted text. The return values indicate the start and end indices of the source characters that should actually be deleted.

:insert **Callback, edit-text-field**

```
(defun    insert-function
          (edit-text-field
          start
          inserted)
(declare  (type edit-text-field edit-text-field)
          (type (integer 0 *) start)
          (type (or character string) inserted))
(declare  (values (or null (integer 0 *)) (or character string))))
```

Invoked when the user inserts one or more source characters. The inserted substring is defined by the `start` index and the `inserted` string or character. This callback allows an application to protect source fields from insertion or to adjust the inserted text. The return values indicate the start index and characters of the string actually inserted. If a `nil` start index is returned, then the insertion is not allowed.

:point **Callback, edit-text-field**

```
(defun    point-function
          (edit-text-field
          new-position)
(declare  (type edit-text-field edit-text-field)
          (type (or null (integer 0 *)) new-position)))
```

Invoked when the user explicitly changes the position of the point. This does not include implicit changes caused by inserting or deleting text.

:resume **Callback, edit-text-field**

```
(defun  resume-function  ())
```

Invoked when the user resumes editing on the `edit-text-field`. For example, this callback is usually invoked when the `edit-text-field` becomes the keyboard focus.

:suspend **Callback, edit-text-field**

```
(defun  suspend-function  ())
```

Invoked when the user suspends editing on the `edit-text-field`. For example, this callback is usually invoked when the `edit-text-field` ceases to be the keyboard focus.

:verify **Callback, edit-text-field**

```
(defun    verify-function
          (edit-text-field)
(declare  (type edit-text-field edit-text-field))
(declare  (values boolean string)))
```

Invoked when the user requests validation of the modified source. This callback allows an application to enforce constraints on the contents of the source. If the first return value is true, then the source satisfies application constraints. Otherwise, the second value is a string containing an error message to be displayed.

This callback is always invoked before invoking the `:complete` callback. The `:complete` callback is called only if `:verify` returns true.

Chapter 3

Images

3.1 Display Image

A `display-image` presents an array of pixels for viewing.

The source of a `display-image` is a `pixmap` or an `image`. The following functions may be used to control the presentation of the displayed pixel array.

- `display-gravity`

The following functions may be used to set the margins surrounding the displayed pixel array.

- `display-bottom-margin`
- `display-left-margin`
- `display-right-margin`
- `display-top-margin`

3.1.1 Functional Definition

make-display-image **Function**

```
(defun    make-display-image
          (&rest initargs
          &key
          (border                    *default-contact-border*)
          (bottom-margin             :default)
          (display-gravity           :tiled)
          foreground
          (left-margin               :default)
          (right-margin              :default)
          (top-margin                :default)
          source
          &allow-other-keys)
(declare  (values display-image)))
```

Creates and returns a `display-image` contact. The resource specification list of the `display-image` class defines a resource for each of the initargs above.

display-bottom-margin **Method, display-image**

```
(defmethod  display-bottom-margin
            ((display-image display-image))
(declare    (values (integer 0 *))))


(defmethod  (setf display-bottom-margin)
            (bottom-margin
            (display-image display-image))
(declare    (type (or (integer 0 *) :default) bottom-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the bottom margin. The height of the contact minus the bottom margin size defines the bottom edge of the clipping rectangle used when displaying the source. Setting the bottom margin to `:default` causes the value of `*default-display-bottom-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-gravity **Method, display-image**

```
(defmethod  display-gravity
            ((display-image display-image))
(declare    (values (or (member :tiled) gravity))))
```

```
(defmethod  (setf display-gravity)
            (gravity
            (display-image display-image))
(declare    (type (or (member :tiled) gravity) gravity))
(declare    (values (or (member :tiled) gravity))))
```

Returns or changes the display gravity of the contact. See Section 8.2. Display gravity controls the alignment of the source bounding rectangle with respect to the clipping rectangle formed by the top, left, bottom, and right margins. The display gravity determines the source position that is aligned with the corresponding position of the margin rectangle.

If the display gravity is `:tiled`, then the image is tiled to fill the entire margin clipping rectangle.

display-left-margin — Method, display-image

```
(defmethod  display-left-margin
            ((display-image display-image))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-left-margin)
            (left-margin
            (display-image display-image))
(declare    (type (or (integer 0 *) :default) left-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the left margin. The left margin size defines the left edge of the clipping rectangle used when displaying the source. Setting the left margin to `:default` causes the value of `*default-display-left-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-right-margin — Method, display-image

```
(defmethod  display-right-margin
            ((display-image display-image))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-right-margin)
            (right-margin
            (display-image display-image))
(declare    (type (or (integer 0 *) :default) right-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the right margin. The width of the contact minus the right margin size defines the right edge of the clipping rectangle used when displaying the source. Setting the right margin to `:default` causes the value of `*default-display-right-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-image-source — Method, display-image

```
(defmethod  display-image-source
            ((display-image display-image))
(declare    (values (or pixmap image))))
```

```
(defmethod  (setf display-image-source)
            (source
            (display-image display-image))
(declare    (type (or pixmap image) source))
(declare    (values (or pixmap image))))
```

Returns or changes the image displayed.

display-top-margin — Method, display-image

```
(defmethod  display-top-margin
            ((display-image display-image))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-top-margin)
            (top-margin
            (display-image display-image))
(declare    (type (or (integer 0 *) :default) top-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the top margin. The top margin size defines the top edge of the clipping rectangle used when displaying the source. Setting the top margin to `:default` causes the value of `*default-display-top-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

Chapter 4

Control

4.1 Action Button

An action-button allows a user to immediately invoke an action.

An action-button label may be either a text string or a pixmap. The action-button font is used to display a text label. The :press callback is invoked when a user initiates operation of the action-button. The :release callback is invoked when a user terminates operation of the action-button. Typically, only a :release callback needs to be defined. Both :press and :release callbacks may be used to control a continuous action.

4.1.1 Functional Definition

make-action-button **Function**

```
(defun    make-action-button
          (&rest initargs
          &key
          (border                     *default-contact-border*)
          (font                       *default-display-text-font*)
          foreground
          (label                      "")
          (label-alignment            :center)
          &allow-other-keys)
(declare  (values action-button)))
```

Creates and returns a action-button contact. The resource specification list of the action-button class defines a resource for each of the initargs above.

button-font **Method, action-button**

```
(defmethod  button-font
            ((action-button action-button))
(declare    (values font)))
```

```
(defmethod  (setf button-font)
            (font
            (action-button action-button))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification for a text label. Together with the `button-label`, this determines the `font` object used to display label characters.

button-label **Method, action-button**

```
(defmethod  button-label
            ((action-button action-button))
(declare    (values (or string pixmap))))
```

```
(defmethod  (setf button-label)
            (label
            (action-button action-button))
(declare    (type (or stringable pixmap image) label))
(declare    (values (or string pixmap))))
```

Returns or changes the label contents. If a symbol is given for the label, it is converted to a string. If an `image` is given for the label, it is converted to a `pixmap`.

button-label-alignment **Method, action-button**

```
(defmethod  button-label-alignment
            ((action-button action-button))
(declare    (values (member :left :center :right))))
```

```
(defmethod  (setf button-label-alignment)
            (alignment
            (action-button action-button))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns or changes the alignment of the label within the `action-button`.

4.1.2 Callbacks

:press **Callback, action-button**

```
(defun press-function ())
```

Invoked when the user initiates the action represented by the `action-button`.

:release **Callback, action-button**

```
(defun release-function ())
```

Invoked when the user terminates the action represented by the `action-button`.

4.2 Action Item

An action-item is a menu item which is functionally equivalent to an action-button. However, an action-item is intended to be used as a member of a menu and therefore may have a different appearance and operation. Selecting an action-item in a menu allows a user to immediately invoke an action.

An action-item label may be either a text string or a pixmap. The action-item font is used to display a text label. The :press callback is invoked when a user initiates operation of the action-item. The :release callback is invoked when a user terminates operation of the action-item. Typically, only a :release callback needs to be defined. Both :press and :release callbacks may be used to control a continuous action.

4.2.1 Functional Definition

make-action-item **Function**

```
(defun    make-action-item
          (&rest initargs
          &key
          (border                *default-contact-border*)
          (font                  *default-display-text-font*)
          foreground
          (label                 "")
          (label-alignment       :left)
          &allow-other-keys)
(declare  (values action-item)))
```

Creates and returns a action-item contact. The resource specification list of the action-item class defines a resource for each of the initargs above.

button-font **Method, action-item**

```
(defmethod  button-font
            ((action-item action-item))
(declare    (values font)))
```

```
(defmethod  (setf button-font)
            (font
            (action-item action-item))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification for a text label. Together with the `button-label`, this determines the `font` object used to display label characters.

button-label **Method, action-item**

```
(defmethod  button-label
            ((action-item action-item))
(declare    (values (or string pixmap))))


(defmethod  (setf button-label)
            (label
            (action-item action-item))
(declare    (type (or stringable pixmap image) label))
(declare    (values (or string pixmap))))
```

Returns or changes the label contents. If a symbol is given for the label, it is converted to a string. If an `image` is given for the label, it is converted to a `pixmap`.

button-label-alignment **Method, action-item**

```
(defmethod  button-label-alignment
            ((action-item action-item))
(declare    (values (member :left :center :right))))


(defmethod  (setf button-label-alignment)
            (alignment
            (action-item action-item))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns or changes the alignment of the label within the `action-item`.

4.2.2 Action Item Choice Items

`action-item` contacts may be used as choice items. The `action-item` class implements the accessor methods and callbacks used in the choice item protocol (see Section 5.1).

Operating an `action-item` is intended to produce an immediate effect. Therefore, as a choice item, an `action-item` does not retain its "selected" state. When it is created, an `action-item` is unselected. After an `action-item` is selected and its `:on` callback is invoked, then it is immediately (and automatically) unselected and its `:off` callback is invoked. This means that it is seldom useful to define an `:off` callback for an `action-item` choice item. For this same reason, a choice contact containing only `action-item` choice items should not use the `:always-one` choice policy.

4.2.3 Callbacks

:press **Callback, action-item**

```
(defun press-function ())
```

Invoked when the user initiates the action represented by the `action-item`.

:release **Callback, action-item**

```
(defun release-function ())
```

Invoked when the user terminates the action represented by the `action-item`.

4.3 Dialog Button

A `dialog-button` allows a user to immediately present a dialog — for example, a `menu` or a `property-sheet`, etc. See Chapter 7 for a description of CLIO dialog classes.

A `dialog-button` is essentially a specialized type of `action-button`, for which `:press`/`:release` semantics (i.e. presenting a dialog) are defined automatically , not by the application programmer. Details of dialog presentation — for example, the position where the dialog appears — are thus implementation-dependent.

A `dialog-button` label may be either a text string or a `pixmap`. The `dialog-button` font is used to display a text label.

4.3.1 Functional Definition

make-dialog-button **Function**

```
(defun    make-dialog-button
          (&rest initargs
          &key
          (border                                       *default-contact-border*)
          dialog
          (font                                         *default-display-text-font*)
          foreground
          (label                                        "")
          (label-alignment                              :left)
          &allow-other-keys)
(declare  (type (or null contact function list) dialog))
(declare  (values dialog-button)))
```

Creates and returns a `dialog-button` contact. The resource specification list of the `dialog-button` class defines a resource for each of the initargs above.

The `dialog` argument specifies the dialog contact to be presented by the `dialog-button`. The value can be a `contact` instance for an existing dialog or (by default) `nil`, if the dialog will be defined later. Otherwise, the dialog contact is created automatically, according to the type of `dialog` argument given.

- A constructor function object. This function is called to create the dialog.
- A list of the form (*constructor* . *initargs*), where *constructor* is a function and *initargs* is a list of initargs used by the *constructor*. A dialog is created using the given constructor and initargs.

button-dialog Method, dialog-button

```
(defmethod  button-dialog
            ((dialog-button dialog-button))
(declare    (values contact)))
```

```
(defmethod  (setf button-dialog)
            (dialog
            (dialog-button dialog-button))
(declare    (type contact                   dialog))
(declare    (values contact)))
```

Returns the dialog presented by the `dialog-button`. Typically, this is an instance one of the CLIO dialog classes described in Chapter 7.

button-font Method, dialog-button

```
(defmethod  button-font
            ((dialog-button dialog-button))
(declare    (values font)))
```

```
(defmethod  (setf button-font)
            (font
            (dialog-button dialog-button))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification for a text label. Together with the `button-label`, this determines the `font` object used to display label characters.

button-label Method, dialog-button

```
(defmethod  button-label
            ((dialog-button dialog-button))
(declare    (values (or string pixmap))))
```

```
(defmethod  (setf button-label)
            (label
            (dialog-button dialog-button))
(declare    (type (or stringable pixmap image) label))
(declare    (values (or string pixmap))))
```

Returns or changes the label contents. If a symbol is given for the label, it is converted to a string. If an `image` is given for the label, it is converted to a `pixmap`.

button-label-alignment **Method, dialog-button**

```
(defmethod  button-label-alignment
            ((dialog-button dialog-button))
(declare    (values (member :left :center :right))))


(defmethod  (setf button-label-alignment)
            (alignment
            (dialog-button dialog-button))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns or changes the alignment of the label within the `dialog-button`.

4.4 Dialog Item

A `dialog-item` is a menu item used to present another dialog — for example, another `menu` or a `property-sheet`, etc. A `dialog-item` allows an application programmer to construct a multi-level hierarchy of menus. See Chapter 7 for a description of CLIO dialog classes.

A `dialog-item` is essentially a specialized type of `action-item`, for which `:press`/`:release` semantics (i.e. presenting a dialog) are defined automatically , not by the application programmer. Details of dialog presentation — for example, the position where the dialog appears — are thus implementation-dependent.

A `dialog-item` label may be either a text string or a `pixmap`. The `dialog-item` font is used to display a text label.

4.4.1 Functional Definition

make-dialog-item **Function**

```
(defun    make-dialog-item
          (&rest initargs
          &key
          (border                                    *default-contact-border*)
          dialog
          (font                                      *default-display-text-font*)
          foreground
          (label                                     "")
          (label-alignment                           :left)
          &allow-other-keys)
(declare  (type (or null contact function list) dialog))
(declare  (values dialog-item)))
```

Creates and returns a `dialog-item` contact. The resource specification list of the `dialog-item` class defines a resource for each of the initargs above.

The `dialog` argument specifies the dialog contact to be presented by the `dialog-item`. The value can be a `contact` instance for an existing dialog or (by default) `nil`, if the dialog will be defined later. Otherwise, the dialog contact is created automatically, according to the type of `dialog` argument given.

- A constructor function object. This function is called to create the dialog.
- A list of the form (*constructor* `.` *initargs*), where *constructor* is a function and *initargs* is a list of initargs used by the *constructor*. A dialog is created using the given constructor and initargs.

button-dialog — Method, dialog-item

```
(defmethod  button-dialog
            ((dialog-item dialog-item))
(declare    (values contact)))
```

```
(defmethod  (setf button-dialog)
            (dialog
            (dialog-item dialog-item))
(declare    (type contact              dialog))
(declare    (values contact)))
```

Returns the dialog presented by the `dialog-item`. Typically, this is an instance one of the CLIO dialog classes described in Chapter 7.

button-font — Method, dialog-item

```
(defmethod  button-font
            ((dialog-item dialog-item))
(declare    (values font)))
```

```
(defmethod  (setf button-font)
            (font
            (dialog-item dialog-item))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification for a text label. Together with the `button-label`, this determines the `font` object used to display label characters.

button-label Method, dialog-item

```
(defmethod  button-label
            ((dialog-item dialog-item))
(declare    (values (or string pixmap))))
```

```
(defmethod  (setf button-label)
            (label
            (dialog-item dialog-item))
(declare    (type (or stringable pixmap image) label))
(declare    (values (or string pixmap))))
```

Returns or changes the label contents. If a symbol is given for the label, it is converted to a string. If an `image` is given for the label, it is converted to a `pixmap`.

button-label-alignment Method, dialog-item

```
(defmethod  button-label-alignment
            ((dialog-item dialog-item))
(declare    (values (member :left :center :right))))
```

```
(defmethod  (setf button-label-alignment)
            (alignment
            (dialog-item dialog-item))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns or changes the alignment of the label within the `dialog-item`.

4.5 Scroller

The `scroller` class represents a particular form of a general type of user interface object known as a **scale**. A scale is used to present a numerical value for viewing and modification. A `scroller` is a scale which plays a specific user interface role — changing the viewing position of another user interface object — and therefore typically has a distinctive style of appearance and operation.

A `scroller` may have either a horizontal or a vertical orientation. The range of possible `scroller` values is given by its maximum and minimum. The current value of a `scroller` is always in this range of **value units**. The `:new-value` callback is invoked whenever a user changes the `scroller` value interactively. A `scroller` displays an indicator whose size is defined in the same value units as the minimum and maximum. Programmers typically change the indicator size to reflect the proportion of some other quantity to the total value range.

4.5.1 Functional Definition

make-scroller **Function**

```
(defun    make-scroller
          (&rest initargs
          &key
          (border              *default-contact-border*)
          foreground
          (increment           1)
          (indicator-size      0)
          (maximum             1)
          (minimum             0)
          (orientation         :vertical)
          (update-delay        0)
          (value               0)
          &allow-other-keys)
(declare  (values scroller)))
```

Creates and returns a `scroller` contact. The resource specification list of the `scroller` class defines a resource for each of the initargs above.

scale-increment **Method, scroller**

```
(defmethod  scale-increment
            ((scroller scroller))
(declare    (values number)))
```

```
(defmethod  (setf scale-increment)
            (increment
            (scroller scroller))
(declare    (type number increment))
(declare    (values number)))
```

Returns or changes the number of value units added to/subtracted from the current value when the user performs an increment/decrement operation.

scale-indicator-size — Method, scroller

```
(defmethod  scale-indicator-size
            ((scroller scroller))
(declare    (values (or number (member :off)))))
```

```
(defmethod  (setf scale-indicator-size)
            (indicator-size
            (scroller scroller))
(declare    (type number indicator-size))
(declare    (values (or number (member :off)))))
```

Returns or changes the indicator size in value units. The exact interpretation of the indicator size value is implementation-dependent.

A `scroller` typically displays the indicator size relative to total value range. A `scroll-frame` sets the indicator size to the number of value units currently visible in the scroll area. If the indicator size is `:off`, then no indicator is displayed.

scale-maximum — Method, scroller

```
(defmethod  scale-maximum
            ((scroller scroller))
(declare    (values number)))
```

```
(defmethod  (setf scale-maximum)
            (maximum
            (scroller scroller))
(declare    (type number maximum))
(declare    (values number)))
```

Returns or changes the maximum value.

scale-minimum **Method, scroller**

```
(defmethod  scale-minimum
            ((scroller scroller))
(declare    (values number)))
```

```
(defmethod  (setf scale-minimum)
            (minimum
            (scroller scroller))
(declare    (type number minimum))
(declare    (values number)))
```

Returns or changes the minimum value.

scale-orientation **Method, scroller**

```
(defmethod  scale-orientation
            ((scroller scroller))
(declare    (values (member :horizontal :vertical))))
```

```
(defmethod  (setf scale-orientation)
            (orientation
            (scroller scroller))
(declare    (type (member :horizontal :vertical) orientation))
(declare    (values (member :horizontal :vertical))))
```

Returns or changes the orientation used to display the value range.

scale-update **Method, scroller**

```
(defmethod  scale-update
            ((scroller scroller)
            &key
            increment
            indicator-size
            maximum
            minimum
            value)
(declare    (type number increment indicator-size maximum minimum value)))
```

Changes one or more `scroller` attributes simultaneously. This method causes the updated `scroller` to be redisplayed only once and thus is more efficient than changing each attribute individually.

scale-update-delay **Method, scroller**

```
(defmethod  scale-update-delay
            ((scroller scroller))
(declare    (values (or number (member :until-done)))))
```

```
(defmethod  (setf scale-update-delay)
            (update-delay
            (scroller scroller))
(declare    (type (or number (member :until-done)) update-delay))
(declare    (values (or number (member :until-done)))))
```

Returns or changes the current incremental update delay time interval. The update delay is meaningful only for `scroller` objects that present user controls for continuous update of the `scroller` value. The update delay specifies the time interval (in seconds) that must elapse during continuous updating before the `:new-value` callback is invoked to report a new value. If the update delay is `:until-done`, then the `:new-value` callback is invoked only when continuous updating ceases.

scale-value **Method, scroller**

```
(defmethod  scale-value
            ((scroller scroller))
(declare    (values number)))
```

```
(defmethod  (setf scale-value)
            (value
            (scroller scroller))
(declare    (type number value))
(declare    (values number)))
```

Returns or changes the current value.

4.5.2 Callbacks

:new-value **Callback, scroller**

```
(defun     new-value-function
           (value)
(declare   (type number value)))
```

Invoked when the value is changed by the user (but *not* when it is changed by the application program) in order to report the new value to the application.

:adjust-value **Callback, scroller**

```
(defun    adjust-value-function
          (value)
(declare  (type number value))
(declare  (values number)))
```

Invoked before `:new-value` each time the value is changed by the user. This callback allows an application to modify a user value before it is actually used.

:begin-continuous **Callback, scroller**

```
(defun  begin-continuous-function  ())
```

Invoked when the user begins continuous update of the `scroller` value. This callback is not used if the `scroller` does not present controls for continuous update.

An application `:new-value` callback may choose to respond differently to new values that occur during continuous update — that is, after the `:begin-continuous` callback is invoked and before the `:end-continuous` callback is invoked. For example, a faster method of displaying the new value might be used during continuous update.

:end-continuous **Callback, scroller**

```
(defun  end-continuous-function  ())
```

Invoked when the user ends continuous update of the `scroller` value. This callback is not used if the `scroller` does not present controls for continuous update.

4.6 Slider

The `slider` class represents a particular form of a general type of user interface object known as a **scale**. A scale is used to present a numerical value for viewing and modification. A `slider` has the same functional interface as a `scroller`, but it plays a different user interface role and typically has a different appearance and behavior.

A `slider` may have either a horizontal or a vertical orientation. The range of possible `slider` values is given by its maximum and minimum. The current value of a `slider` is always in this range of **value units**. The `:new-value` callback is invoked whenever a user changes the `slider` value interactively. The indicator size of a `slider` may specify the distance in value units between "ticks," or other fixed labels used to show the value at various indicator positions.

4.6.1 Functional Definition

make-slider **Function**

```
(defun    make-slider
          (&rest initargs
          &key
          (border               *default-contact-border*)
          foreground
          (increment            1)
          (indicator-size       0)
          (maximum              1)
          (minimum              0)
          (orientation          :horizontal)
          (update-delay         0)
          (value                0)
          &allow-other-keys)
(declare  (values slider)))
```

Creates and returns a `slider` contact. The resource specification list of the `slider` class defines a resource for each of the initargs above.

scale-increment **Method, slider**

```
(defmethod  scale-increment
            ((slider slider))
(declare    (values number)))
```

```
(defmethod  (setf scale-increment)
            (increment
            (slider slider))
(declare    (type number increment))
(declare    (values number)))
```

Returns or changes the number of value units added to/subtracted from the current value when the user performs an increment/decrement operation.

scale-indicator-size **Method, slider**

```
(defmethod  scale-indicator-size
            ((slider slider))
(declare    (values (or number (member :off)))))
```

```
(defmethod  (setf scale-indicator-size)
            (indicator-size
            (slider slider))
(declare    (type number indicator-size))
(declare    (values (or number (member :off)))))
```

Returns or changes the indicator size in value units. The exact interpretation of the indicator size value is implementation-dependent.

A `slider` typically interprets the indicator size as the distance in value units between "ticks," or other fixed labels used to show the value at various indicator positions. If the indicator size is zero, then the spacing of indicator labels is determined automatically and may change, depending on the size of the `slider` and its minimum/maximum value. If the indicator size is `:off`, then no indicator labels are displayed.

scale-maximum **Method, slider**

```
(defmethod  scale-maximum
            ((slider slider))
(declare    (values number)))
```

```
(defmethod  (setf scale-maximum)
            (maximum
            (slider slider))
(declare    (type number maximum))
(declare    (values number)))
```

Returns or changes the maximum value.

scale-minimum **Method, slider**

```
(defmethod  scale-minimum
            ((slider slider))
(declare    (values number)))
```

```
(defmethod  (setf scale-minimum)
            (minimum
            (slider slider))
(declare    (type number minimum))
(declare    (values number)))
```

Returns or changes the minimum value.

scale-orientation **Method, slider**

```
(defmethod  scale-orientation
            ((slider slider))
(declare    (values (member :horizontal :vertical))))
```

```
(defmethod  (setf scale-orientation)
            (orientation
            (slider slider))
(declare    (type (member :horizontal :vertical) orientation))
(declare    (values (member :horizontal :vertical))))
```

Returns or changes the orientation used to display the value range.

scale-update **Method, slider**

```
(defmethod  scale-update
            ((slider slider)
            &key
            increment
            indicator-size
            maximum
            minimum
            value)
(declare    (type number increment indicator-size maximum minimum value)))
```

Changes one or more `slider` attributes simultaneously. This method causes the updated `slider` to be redisplayed only once and thus is more efficient than changing each attribute individually.

scale-update-delay **Method, slider**

```
(defmethod  scale-update-delay
            ((slider slider))
(declare    (values (or number (member :until-done)))))
```

```
(defmethod  (setf scale-update-delay)
            (update-delay
            (slider slider))
(declare    (type (or number (member :until-done)) update-delay))
(declare    (values (or number (member :until-done)))))
```

Returns or changes the current incremental update delay time interval. The update delay is meaningful only for `slider` objects that present user controls for continuous update of the `slider` value. The update delay specifies the time interval (in seconds) that must elapse during continuous updating before the `:new-value` callback is invoked to report a new value. If the update delay is `:until-done`, then the `:new-value` callback is invoked only when continuous updating ceases.

scale-value **Method, slider**

```
(defmethod  scale-value
            ((slider slider))
(declare    (values number)))
```

```
(defmethod  (setf scale-value)
            (value
            (slider slider))
(declare    (type number value))
(declare    (values number)))
```

Returns or changes the current value.

4.6.2 Callbacks

:new-value **Callback, slider**

```
(defun    new-value-function
          (value)
(declare  (type number value)))
```

Invoked when the value is changed by the user (but *not* when it is changed by the application program) in order to report the new value to the application.

:adjust-value **Callback, slider**

```
(defun    adjust-value-function
          (value)
(declare  (type number value))
(declare  (values number)))
```

Invoked before `:new-value` each time the value is changed by the user. This callback allows an application to modify a user value before it is actually used.

:begin-continuous **Callback, slider**

```
(defun  begin-continuous-function  ())
```

Invoked when the user begins continuous update of the `slider` value. This callback is not used if the `slider` does not present controls for continuous update.

An application `:new-value` callback may choose to respond differently to new values that occur during continuous update — that is, after the `:begin-continuous` callback is invoked and before the `:end-continuous` callback is invoked. For example, a faster method of displaying the new value might be used during continuous update.

:end-continuous **Callback, slider**

```
(defun  end-continuous-function  ())
```

Invoked when the user ends continuous update of the `slider` value. This callback is not used if the `slider` does not present controls for continuous update.

4.7 Toggle Button

A `toggle-button` represents a two-state switch which a user may turn "on" or "off."

A `toggle-button` label may be either a text string or a `pixmap`. The `toggle-button` font is used to display a text label.

4.7.1 Functional Definition

make-toggle-button **Function**

```
(defun    make-toggle-button
          (&rest initargs
          &key
          (border                    *default-contact-border*)
          (font                      *default-display-text-font*)
          foreground
          (label                     "")
          (label-alignment           :center)
          (switch                    :off)
          &allow-other-keys)
(declare  (values toggle-button)))
```

Creates and returns a `toggle-button` contact. The resource specification list of the `toggle-button` class defines a resource for each of the initargs above.

button-font **Method, toggle-button**

```
(defmethod  button-font
            ((toggle-button toggle-button))
(declare    (values font)))
```

```
(defmethod  (setf button-font)
            (font
            (toggle-button toggle-button))
(declare    (type fontable font))
(declare    (values font)))
```

Returns or changes the font specification for a text label. Together with the `button-label`, this determines the `font` object used to display label characters.

button-label **Method, toggle-button**

```
(defmethod  button-label
            ((toggle-button toggle-button))
(declare    (values (or string pixmap))))
```

```
(defmethod  (setf button-label)
            (label
            (toggle-button toggle-button))
(declare    (type (or stringable pixmap image) label))
(declare    (values (or string pixmap))))
```

Returns or changes the label contents. If a symbol is given for the label, it is converted to a string. If an `image` is given for the label, it is converted to a `pixmap`.

button-label-alignment **Method, toggle-button**

```
(defmethod  button-label-alignment
            ((toggle-button toggle-button))
(declare    (values (member :left :center :right))))
```

```
(defmethod  (setf button-label-alignment)
            (alignment
            (toggle-button toggle-button))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns or changes the alignment of the label within the `toggle-button`.

button-switch **Method, toggle-button**

```
(defmethod  button-switch
            ((toggle-button toggle-button))
(declare    (values (member :on :off))))
```

```
(defmethod  (setf button-switch)
            (switch
            (toggle-button toggle-button))
(declare    (type (member :on :off) switch))
(declare    (values (member :on :off))))
```

Returns or changes the state of the `toggle-button` switch.

4.7.2 Toggle Button Choice Items

`toggle-button` contacts may be used as choice items. The `toggle-button` class implements the accessor methods and callbacks used in the choice item protocol (see Section 5.1).

4.7.3 Callbacks

The callbacks used by a `toggle-button` are defined by the choice item protocol. See Section 5.1.

Chapter 5

Choice

A **choice** contact is a composite contact used to contain a set of **choice items**. A choice contact allows a user to choose one or more of the choice items which are its children. In order to operate correctly as a choice item, a child contact need not belong to any specific class, but it must obey a certain **choice item protocol**.

5.1 Choice Item Protocol

The choice item protocol is a set of accessor functions and callbacks that are used by choice contacts to control the selection of choice items. The choice item protocol is *not* an application programmer interface. Rather, it is an interface used by a contact programmer to implement a choice contact. The choice item protocol allows any choice contact to accomodate a variety of choice item classes. In CLIO, `toggle-button` and `action-item` contacts are examples of choice item classes that implement this choice item protocol.

5.1.1 Methods

The following generic accessor functions, which accept a choice item argument, must have an applicable method for each choice item.

choice-item-font **Method, choice item protocol**

```
(defmethod  choice-item-font
            (choice-item)
(declare    (type contact     choice-item))
(declare    (values           font)))
```

```
(defmethod  (setf choice-item-font)
            (font
            choice-item)
(declare    (type fontable font)
            (type contact            choice-item))
(declare    (values font)))
```

Returns and changes the font specification of a choice item text label. Together with the `choice-item-label`, this determines the `font` object used to display label characters. `choice-item-font` should return `nil` if and only if the item label is a `pixmap`.

If a `choices` or `multiple-choices` parent has a `non-nil` font, then `(setf choice-item-font)` will be used to make all choice item labels use the parent font.

choice-item-highlight-default-p **Method, choice item protocol**

```
(defmethod  choice-item-highlight-default-p
            (choice-item)
(declare    (type contact                    choice-item))
(declare    (values boolean)))
```

```
(defmethod  (setf choice-item-highlight-default-p)
            (highlight-default-p
            choice-item)
(declare    (type boolean highlight-default-p)
            (type contact                             choice-item))
(declare    (values boolean)))
```

Returns and changes the visual state used by a choice item when it is a member of the default selection. When this state is true, a choice item should be displayed differently to indicate that it is a default selection; otherwise, a choice item should be displayed normally.

choice-item-highlight-selected-p **Method, choice item protocol**

```
(defmethod  choice-item-highlight-selected-p
            (choice-item)
(declare    (type contact                    choice-item))
(declare    (values boolean)))
```

```
(defmethod  (setf choice-item-highlight-selected-p)
            (highlight-selected-p
            choice-item)
(declare    (type boolean highlight-selected-p)
            (type contact                           choice-item))
(declare    (values boolean)))
```

Returns and changes the visual state used by a choice item when it is selected. When this state is true, a choice item should be displayed differently to indicate that it is selected; otherwise, a choice item should be displayed normally.

Note that these functions are related only to the visual highlighting of selection. It is possible for this state to be true, even if the choice item has not actually been selected.

choice-item-label **Method, choice item protocol**

```
(defmethod  choice-item-label
            (choice-item)
(declare    (type contact                choice-item))
(declare    (values(or string pixmap))))
```

Returns the `pixmap` or string object used to label the choice item.

choice-item-label-alignment **Method, choice item protocol**

```
(defmethod  choice-item-label-alignment
            (choice-item)
(declare    (type contact                           choice-item))
(declare    (values (member :left :center :right))))
```

```
(defmethod  (setf choice-item-label-alignment)
            (alignment
            choice-item)
(declare    (type (member :left :center :right) alignment)
            (type contact                                  choice-item))
(declare    (values (member :left :center :right))))
```

Returns or changes the alignment of the label within the `choice-item`.

choice-item-selected-p **Method, choice item protocol**

```
(defmethod  choice-item-selected-p
            (choice-item)
(declare    (type contact          choice-item))
(declare    (values boolean)))

(defmethod  (setf choice-item-selected-p)
            (selected-p
            choice-item)
(declare    (type boolean selected-p)
            (type contact                 choice-item))
(declare    (values boolean)))
```

Returns and changes the selected state of a choice item.

`(setf choice-item-selected-p)` is called when the current selection is changed by the application program, i.e. when the application program calls `(setf choice-selection)`. Calling this method should have the same effect as (un)selecting the choice item interactively by a user.

5.1.2 Application Callbacks

The application semantics of (un)selecting a choice item are implemented primarily by the callbacks of the choice item, *not* by callbacks of its parent. An application determines the semantics of (un)selecting a choice item by defining the following callbacks for it.

:off **Callback, choice item protocol**

```
(defun  off-function  ())
```

Invoked on the current selection item when the selection changes.

:on **Callback, choice item protocol**

```
(defun  on-function  ())
```

Invoked when the choice item is selected.

5.1.3 Control Callbacks

Depending on its behavior, a choice contact may need to respond to certain user operations on an individual choice item. For example, if a choice contact allows at most one choice item to be selected, then it should respond to a new user selection by unselecting the previously-selected choice item. Thus, a choice item must invoke certain control callbacks to allow its containing choices contact to control its selection interaction properly.

A choice item should invoke the following callbacks at the appropriate times in order inform its choice parent about important user actions. A choice contact will associate with these callback names the functions that implement its response to these user actions.

:change-allowed-p **Callback, choice item protocol**

```
(defun    change-allowed-p-function
          (to-selected-p)
(declare  (type boolean to-selected-p))
(declare  (values boolean)))
```

Invoked when the choice item is about to be selected or unselected. The `to-selected-p` argument is true if and only if the choice item is about to be selected. If no `:change-allowed-p` function is defined or if this callback function returns true, then choice item highlighting may change and the `:changing` callback may be invoked and, ultimately, the `:on`/`:off` callback may be invoked. However, if `:change-allowed-p` returns `nil`, then the choice item is not allowed to change state, and none of these operations should be performed.

:changing **Callback, choice item protocol**

```
(defun    changing-function
          (to-selected-p)
(declare  (type boolean to-selected-p)))
```

A sequence of user actions may be required to (un)select a choice item (for example, a `:button-press` event followed by a `:button-release` event). The `:changing` callback is invoked when the user begins to (un)select the choice item; the `to-selected-p` argument is true if and only if the user is beginning to select the choice item. In this case, the choice item may have changed its visual appearance to indicate the anticipated transition to the new state. A choice contact may respond to this callback by changing the highlighting of other choice items.

:canceling-change **Callback, choice item protocol**

```
(defun    canceling-change-function
          (to-selected-p)
(declare  (type boolean to-selected-p)))
```

A sequence of user actions may be required to (un)select a choice item (for example, a `:button-press` event followed by a `:button-release` event). The `:canceling-change` callback is invoked when the user cancels his (un)selection without completing the full sequence; the `to-selected-p` argument is true if the user is canceling a selection operation and `nil` otherwise. In this case, the choice item is no longer about to be (un)selected and may have removed any visual feedback of the previously-anticipated change in state. A choice contact may respond to this callback by changing the highlighting of other choice items.

5.2 Choices

A `choices` contact is a composite that allows a user to choose at most one of its children. The members of a `choices` children list are referred to as **choice items**. A `choices` composite uses the geometry management of a `table` to arrange its items in rows and columns (see Section 6.3).

The item of a `choices` chosen by a user is referred to as the **selection**[1]. An application program can initialize both the current selection and the default selection. `choices` behavior depends on its choice policy, which can be either `:always-one` (the current selection can be changed only by choosing another item) or `:one-or-none` (the current selection may be `nil`).

The following functions may be used to specify the layout of `choices` members.

- `table-column-alignment`
- `table-column-width`
- `table-columns`
- `table-row-alignment`
- `table-row-height`
- `table-same-height-in-row`
- `table-same-width-in-column`

The following functions may be used to set the margins surrounding the `choices`.

- `display-bottom-margin`
- `display-left-margin`
- `display-right-margin`
- `display-top-margin`

[1]Note that this meaning of "selection" is completely different from the concept of the selection mechanism for interclient communication discussed in Section 8.3.

The following functions may be used to set the spacing between `choices` rows and columns.

- `display-horizontal-space`
- `display-vertical-space`

5.2.1 Functional Definition

A `choices` composite uses the geometry management of a `table` to arrange its items in rows and columns. All accessors, initargs, and constraint resources defined in Section 6.3 are applicable to a `choices` composite.

make-choices **Function**

```
(defun    make-choices
          (&rest initargs
          &key
          (border               *default-contact-border*)
          (bottom-margin        :default)
          (choice-policy        :one-or-none)
          (column-alignment     :left)
          (column-width         :maximum)
          (columns              0)
          default-selection
          (font                 *default-choice-font*)
          foreground
          (horizontal-space     :default)
          (left-margin          :default)
          (right-margin         :default)
          (row-alignment        :bottom)
          (row-height           :maximum)
          (same-height-in-row   :off)
          (same-width-in-column :off)
          separators
          (top-margin           :default)
          (vertical-space       :default)
          &allow-other-keys)
 (declare (values choices)))
```

Creates and returns a `choices` contact. The resource specification list of the `choices` class defines a resource for each of the initargs above.

The `default-selection` initarg is the contact name symbol for the choice item that is the initial default selection.

choice-default **Method, choices**

```
(defmethod  choice-default
            ((choices choices))
(declare    (values (or null contact))))
```

```
(defmethod  (setf choice-default)
            (default
            (choices choices))
(declare    (type (or null contact) default))
(declare    (values (or null contact))))
```

Returns or changes the default selection.

choice-font **Method, choices**

```
(defmethod  choice-font
            ((choices choices))
(declare    (values (or null font))))
```

```
(defmethod  (setf choice-font)
            (font
            (choices choices))
(declare    (type (or null fontable) font))
(declare    (values (or null font))))
```

Returns or changes the font specification for all choice item text labels. Together with the `choice-item-label`, this determines the `font` object used to display label characters. If `nil`, then choice items are allowed to use different fonts.

choice-policy **Method, choices**

```
(defmethod  choice-policy
            ((choices choices))
(declare    (values (member :always-one :one-or-none))))
```

```
(defmethod  (setf choice-policy)
            (policy
            (choices choices))
(declare    (type (member :always-one :one-or-none) policy))
(declare    (values (member :always-one :one-or-none))))
```

Returns or changes the choice policy. If `:always-one`, then the current selection can be changed only by choosing another item. If `:one-or-none`, then the current selection may be `nil`.

choice-selection — Method, choices

```
(defmethod  choice-selection
            ((choices choices))
(declare    (values (or null contact))))


(defmethod  (setf choice-selection)
            (selection
            (choices choices))
(declare    (type (or null contact) selection))
(declare    (values (or null contact))))
```

Returns or changes the currently-selected choice item. In order to avoid surprising the user, an application program should change the selection only in response to some user action.

5.3 Multiple Choices

A `multiple-choices` contact is a composite that allows a user to choose any subset of its children. The members of a `multiple-choices` children list are referred to as **choice items**. A `multiple-choices` composite uses the geometry management of a `table` to arrange its items in rows and columns (see Section 6.3).

The items of a `multiple-choices` chosen by a user are referred to as the **selection**[2]. An application program can initialize both the current selection and the default selection.

The following functions may be used to specify the layout of `table` members.

- `table-column-alignment`
- `table-column-width`
- `table-columns`
- `table-row-alignment`
- `table-row-height`
- `table-same-height-in-row`
- `table-same-width-in-column`

The following functions may be used to set the margins surrounding the `multiple-choices`.

- `display-bottom-margin`
- `display-left-margin`
- `display-right-margin`
- `display-top-margin`

[2]Note that this meaning of "selection" is completely different from the concept of the selection mechanism for interclient communication discussed in Section 8.3.

The following functions may be used to set the spacing between `multiple-choices` rows and columns.

- `display-horizontal-space`
- `display-vertical-space`

5.3.1 Functional Definition

A `multiple-choices` composite uses the geometry management of a `table` to arrange its items in rows and columns. All accessors, initargs, and constraint resources defined in Section 6.3 are applicable to a `multiple-choices` composite.

make-multiple-choices **Function**

```
(defun    make-multiple-choices
          (&rest initargs
          &key
          (border                      *default-contact-border*)
          (bottom-margin               :default)
          (column-alignment            :left)
          (column-width                :maximum)
          (columns                     0)
          default-selection
          (font                        *default-choice-font*)
          foreground
          (horizontal-space            :default)
          (left-margin                 :default)
          (right-margin                :default)
          (row-alignment               :bottom)
          (row-height                  :maximum)
          (same-height-in-row          :off)
          (same-width-in-column        :off)
          separators
          (top-margin                  :default)
          (vertical-space              :default)
          &allow-other-keys)
(declare  (values multiple-choices)))
```

Creates and returns a `multiple-choices` contact. The resource specification list of the `multiple-choices` class defines a resource for each of the initargs above.

The `default-selection` initarg is a list of contact name symbols for the choice items that are the initial default selection.

choice-default **Method, multiple-choices**

```
(defmethod  choice-default
            ((multiple-choices multiple-choices))
(declare    (values list)))
```

```
(defmethod  (setf choice-default)
            (default
            (multiple-choices multiple-choices))
(declare    (type list default))
(declare    (values list)))
```

Returns or changes the default selection.

choice-font **Method, multiple-choices**

```
(defmethod  choice-font
            ((multiple-choices multiple-choices))
(declare    (values(or null font))))
```

```
(defmethod  (setf choice-font)
            (font
            (multiple-choices multiple-choices))
(declare    (type (or null fontable) font))
(declare    (values (or null font))))
```

Returns or changes the font specification for all choice item text labels. Together with the `choice-item-label`, this determines the `font` object used to display label characters. If `nil`, then choice items are allowed to use different fonts.

choice-selection **Method, multiple-choices**

```
(defmethod  choice-selection
            ((multiple-choices multiple-choices))
(declare    (values list)))
```

```
(defmethod  (setf choice-selection)
            (selection
            (multiple-choices multiple-choices))
(declare    (type list selection))
(declare    (values list)))
```

Returns or changes the currently-selected choice items. In order to avoid surprising the user, an application program should change the selection only in response to some user action.

Chapter 6

Containers

A **container** is a composite contact used to manage a set of child contacts. Some container classes are referred to as **layouts**. A layout is a type of container whose purpose is limited to providing a specific style of geometry management. Examples of CLIO layouts include `form` and `table`.

6.1 Form

A `form` is a layout contact which manages the geometry of a set of children (or **members**) according to a set of constraints. Geometrical constraints are defined by constraint resources[1] of individual members and by the **links** between members. Links are used to specify the ideal/minimum/maximum spacing between two members or between a member and the form itself. Member constraints are used to define the minimum/maximum for each member's size.

6.1.1 Form Layout Policy

A `form` is said to satisfy its layout constraints if the size of each member and the length of each link lies between its requested minimum and maximum. The basic `form` layout policy is to satisfy its constraints and keep all members completely visible within the current size of the form. Therefore, changing the size of a `form` will usually result in changes to the size or position of its members or to the length of the links that define the spaces between members. Similarly, any changes to the member constraints or to the links of a `form` may cause the form to change its layout to satisfy the new constraints. Note that members may be clipped by the edges of a `form`, if its constraints cannot be satisfied otherwise.

When a `form` is first created or when it is resized or when its set of members and links is changed, then the `form` must ensure that its constraints remain satisfied. In doing so, a `form` first determines

[1] See [2] for a complete description of constraint resources.

its own "ideal" size, defined by the current sizes of all members and links. A form will then negotiate with its geometry manager to change to the "best" approximation of the ideal size. If this best size differs from the current size of the form, then the resulting "stretch" or "shrink" is distributed over all members and links, in proportion to the "stretchability" or "shrinkability" of each individual member and link. The stretchability of a member or link is the difference between its current size and its maximum size. Similarly, the shrinkability of a member or link is the difference between its current size and its minimum size.

The maximum size of a member or the length of a link may be :infinite, indicating that the member or link can grow to any size. The form layout policy treats :infinite values specially. Within a chain of linked members, any extra "stretch" in the layout is distributed equally among the :infinite members and links, leaving other members and links unchanged.

6.1.2 Functional Definition

make-form **Function**

```
(defun    make-form
          (&rest initargs
          &key
          (border              *default-contact-border*)
          foreground
          horizontal-links
          vertical-links
          &allow-other-keys)
(declare  (type list           horizontal-links vertical-links))
(declare  (values form)))
```

Creates and returns a form contact. The resource specification list of the form class defines a resource for each of the initargs above.

The horizontal-links and vertical-links arguments initialize the links between form members (see Section 6.1.4). Each argument is a list of the form (*link-init-list**), where each *link-init-list* is a list of keyword/value pairs. The keywords allowed for horizontal-links and vertical-links are the same as those for make-horizontal-link and make-vertical-link, respectively. However, initial values of :from and :to keywords must be specified by contact name symbols, rather than contact instances.

6.1.3 Constraint Resources

The following functions may be used to return or change the constraint resources of form members.

form-max-height **Function**

```
(defun    form-max-height
          (member)
(declare  (type contact                                  member))
(declare  (values (or card16 (member :infinite)))))
```

Returns or (with `setf`) changes the maximum height allowed for the member. A `form` is allowed to change the height of the member to any value between its minimum/maximum height in order to satisfy its layout constraints.

The maximum height of a member may be initialized by specifying a `:max-height` initarg to the constructor function which creates the member.

form-max-width **Function**

```
(defun    form-max-width
          (member)
(declare  (type contact                                  member))
(declare  (values (or card16 (member :infinite)))))
```

Returns or (with `setf`) changes the maximum width allowed for the member. A `form` is allowed to change the width of the member to any value between its minimum/maximum width in order to satisfy its layout constraints.

The maximum width of a member may be initialized by specifying a `:max-width` initarg to the constructor function which creates the member.

form-min-height **Function**

```
(defun    form-min-height
          (member)
(declare  (type contact      member))
(declare  (values card16)))
```

Returns or (with `setf`) changes the minimum height allowed for the member. A `form` is allowed to change the height of the member to any value between its minimum/maximum height in order to satisfy its layout constraints.

The minimum height of a member may be initialized by specifying a `:min-height` initarg to the constructor function which creates the member.

form-min-width **Function**

```
(defun    form-min-width
          (member)
(declare  (type contact      member))
(declare  (values card16)))
```

Returns or (with `setf`) changes the minimum width allowed for the member. A `form` is allowed to change the width of the member to any value between its minimum/maximum width in order to satisfy its layout constraints.

The minimum width of a member may be initialized by specifying a `:min-width` initarg to the constructor function which creates the member.

6.1.4 Link Functions

A `link` represents the space between two members of a `form` layout. A `link` has a specific orientation, either horizontal or vertical. The **length** of a `link` specifies the distance in pixels from its **from-member** to its **to-member**,. The length of a `link` is also directional, where a positive length is to the right or downward, depending on the orientation of the `link`. The space represented by the length of a `link` is always measured between specific **attach points** on its to-member and from-member. An attach point can be either the center of the member or one of its edges — the left or right edge for horizontal links, the top or bottom edge for vertical links. Either the to-member or the from-member of a `link` can be the `form` itself; in this case, the other member must be a member of the `form`.

In general, between any two members, there can be at most one horizontal link and at most one vertical link. However, there can be multiple links between a `form` and one of its members. For a given orientation, a member can have a distinct link to each possible attach point on the `form`. For example, a single member could define horizontal links to both the left and right edges of its `form`.

The following functions are used to create, destroy, look up, and modify `link` objects.

make-horizontal-link **Function**

```
(defun    make-horizontal-link
          (&key
          (attach-from                        :right)
          (attach-to                          :left)
          from
          length
          (maximum                            :infinite)
          (minimum                            0)
          to)
(declare  (type contact                       from to)
          (type (member :left :center :right) attach-from attach-to)
          (type int16                         length minimum)
          (type (or int16 (member :infinite)) maximum))
(declare  (values link)))
```

Creates and returns a horizontal link between `from` and `to`. The `from` and `to` contacts may either be members of the same `form` or a member and its parent `form`. The `length` is measured from `from` to `to`, so a positive `length` indicates that `to` is to the right of `from`; by default, the `length` is equal to the `minimum`. `attach-from` and `attach-to` indicate where the link attaches to `from` and `to`, respectively.

make-vertical-link **Function**

```
(defun    make-vertical-link
          (&key
          (attach-from                        :bottom)
          (attach-to                          :top)
          from
          length
          (maximum                            :infinite)
          (minimum                            0)
          to)
(declare  (type contact                       from to)
          (type (member :top :center :bottom) attach-from attach-to)
          (type int16                         length minimum)
          (type (or int16 (member :infinite)) maximum))
(declare  (values link)))
```

Creates and returns a vertical link between `from` and `to`. The `from` and `to` contacts may either be members of the same `form` or a member and its parent `form`. The `length` is measured from `from` to `to`, so a positive `length` indicates that `to` is below `from`; by default, the `length` is equal to the `minimum`. `attach-from` and `attach-to` indicate where the link attaches to `from` and `to`, respectively.

destroy **Method, link**

```
(defmethod  destroy
            ((link   link)))
```

Destroys the `link`, removing its layout constraints between its `from` and `to` contacts.

find-link **Function**

```
(defun    find-link
          (contact-1
          contact-2
          orientation
          &optional
          form-attach-point)
(declare  (type contact                                  contact-1 contact-2)
          (type (member :horizontal :vertical)           orientation)
          (type (member :left :right :top :bottom :center)  form-attach-point))
(declare  (values (or link null))))
```

Returns the link of the given `orientation` between `contact-1` and `contact-2`. If either `contact-1` or `contact-2` is a `form`, then the `form-attach-point` argument may be given to specify the `form` link desired; the `form-attach-point` may be omitted if there is only one link of the given `orientation` between the `form` and the member.

link-attach-from **Method, link**

```
(defmethod  link-attach-from
            ((link link))
(declare    (values (member :left :right :top :bottom :center))))
```

```
(defmethod  (setf link-attach-from)
            (attach-from
            (link link))
(declare    (type (member :left :right :top :bottom :center)      attach-from))
(declare    (values (member :left :right :top :bottom :center))))
```

Returns and (with `setf`) changes the attach point of the `link` on its from-member. For a horizontal link, the attach point is one of `:left`, `:center`, or `:right`. For a vertical link, the attach point is one of `:top`, `:center`, or `:bottom`.

link-attach-to **Method, link**

```
(defmethod  link-attach-to
            ((link link))
(declare    (values (member :left :right :top :bottom :center))))
```

```
(defmethod  (setf link-attach-to)
            (attach-to
            (link link))
(declare    (type (member :left :right :top :bottom :center)      attach-to))
(declare    (values (member :left :right :top :bottom :center))))
```

Returns and (with `setf`) changes the attach point of the `link` on its to-member. For a horizontal link, the attach point is one of `:left`, `:center`, or `:right`. For a vertical link, the attach point is one of `:top`, `:center`, or `:bottom`.

link-from **Method, link**

```
(defmethod  link-from
            ((link link))
(declare    (values contact)))
```

Returns the from-member of the `link`.

link-length **Method, link**

```
(defmethod  link-length
            ((link link))
(declare    (values int16)))
```

```
(defmethod  (setf link-length)
            (length
            (link link))
(declare    (type int16         length))
(declare    (values int16)))
```

Returns and (with `setf`) changes the current length of the `link`.

link-maximum **Method, link**

```
(defmethod  link-maximum
            ((link link))
(declare    (values (or int16 (member :infinite)))))
```

```
(defmethod  (setf link-maximum)
            (maximum
            (link link))
(declare    (type (or int16 (member :infinite))       maximum))
(declare    (values (or int16 (member :infinite)))))
```

Returns and (with `setf`) changes the maximum length of the `link`.

link-minimum **Method, link**

```
(defmethod  link-minimum
            ((link link))
(declare    (values int16)))
```

```
(defmethod  (setf link-minimum)
            (minimum
            (link link))
(declare    (type int16          minimum))
(declare    (values int16)))
```

Returns and (with `setf`) changes the minimum length of the `link`.

link-orientation **Method, link**

```
(defmethod  link-orientation
            ((link link))
(declare    (values (member :horizontal :vertical))))
```

Returns the orientation of the `link`.

link-to **Method, link**

```
(defmethod  link-to
            ((link link))
(declare    (values contact)))
```

Returns the to-member of the `link`.

link-update Method, link

```
(defun    link-update
          ((link link)
          &key
          attach-from
          attach-to
          length
          minimum
          maximum)
(declare  (type (member :left :right :top :center :bottom)  attach-from attach-to)
          (type int16                                        length minimum)
          (type (or int16 (member :infinite))               maximum)))
```

Changes one or more `link` attributes simultaneously. This method causes form constraints to be reevaluated only once and thus is more efficient than changing each attribute individually.

6.2 Scroll Frame

A `scroll-frame` is a `composite` contact which contains a contact called the **content** and which allows a user to view different parts of the content by manipulating horizontal and/or vertical scrolling controls. The scrolling controls are implemented by `scroller` contacts and are created automatically.

The content is displayed in an area of the `scroll-frame` represented by a contact called the **scroll area**. The scroll area is the parent contact for the content. An application programmer initializes a `scroll-frame` by defining its content as a child of the scroll area. A `scroll-frame` can contain at most one contact as its content.

Scrolling is performed by callback functions defined on the content. An application programmer may define `:horizontal-calibrate` and `:vertical-calibrate` callbacks, which are called to initialize the `scroll-frame`. The content's `:scroll-to` callback is called to redisplay the content at a new position. The `:horizontal-calibrate` and `:vertical-calibrate` callbacks allow the application programmer to define the ranges and units for scroll position values. All scrolling is performed in these **content units**. A `scroll-frame` uses defaults for the `:horizontal-calibrate`, `:vertical-calibrate`, and `:scroll-to` callback functions, if they are not given by the application programmer.

A `scroll-frame` also defines `:horizontal-update` and `:vertical-update` callback functions on its content. These functions may be called to inform the `scroll-frame` about changes in content size, position, etc. caused by the application program.

6.2.1 Functional Definition

make-scroll-frame **Function**

```
(defun    make-scroll-frame
          (&rest initargs
          &key
          (border                *default-contact-border*)
          content
          foreground
          (horizontal            :on)
          (left                  0)
          (top                   0)
          (vertical              :on)
          &allow-other-keys)
(declare  (values scroll-frame)))
```

Creates and returns a `scroll-frame` contact. The resource specification list of the `scroll-frame` class defines a resource for each of the initargs above.

The `content` initarg, if given, causes the content contact to be created automatically with a specific constructor function and an optional list of initargs. The value of the `content` argument is either a constructor function or a list of the form (*constructor* . *content-initargs*), where *constructor* is a function that creates and returns the content and *content-initargs* is a list of keyword/value pairs used by the *constructor*.

scroll-frame-area — Method, scroll-frame

```
(defmethod  scroll-frame-area
            ((scroll-frame scroll-frame))
(declare    (values contact)))
```

Returns the scroll area of the `scroll-frame`. The content of the `scroll-frame` may be set by creating a contact whose parent is the scroll area.

scroll-frame-content — Method, scroll-frame

```
(defmethod  scroll-frame-content
            ((scroll-frame scroll-frame))
(declare    (values contact)))
```

Returns the content of the `scroll-frame`. The content of the `scroll-frame` is set by creating a contact whose parent is the scroll area, or by using the `content` initarg with `make-scroll-frame`.

scroll-frame-horizontal — Method, scroll-frame

```
(defmethod  scroll-frame-horizontal
            ((scroll-frame scroll-frame))
(declare    (values (member :on :off))))


(defmethod  (setf scroll-frame-horizontal)
            (switch
            (scroll-frame scroll-frame))
(declare    (type (member :on :off) switch))
(declare    (values (member :on :off))))
```

Enables or disables the user control for horizontal scrolling.

scroll-frame-vertical **Method, scroll-frame**

```
(defmethod  scroll-frame-vertical
            ((scroll-frame scroll-frame))
(declare    (values (member :on :off))))
```

```
(defmethod  (setf scroll-frame-vertical)
            (switch
            (scroll-frame scroll-frame))
(declare    (type (member :on :off) switch))
(declare    (values (member :on :off))))
```

Enables or disables the user control for vertical scrolling.

scroll-frame-position **Method, scroll-frame**

```
(defmethod  scroll-frame-position
            ((scroll-frame scroll-frame))
(declare    (values left top)))
```

Returns the current horizontal/vertical position of the content (in content units) which appears at the left/top edge of the scroll area.

scroll-frame-reposition **Method, scroll-frame**

```
(defmethod  scroll-frame-reposition
            ((scroll-frame scroll-frame)
            &key left top)
(declare    (type number left top))
(declare    (values left top)))
```

Changes the horizontal/vertical position of the content (in content units) which appears at the left/top edge of the scroll area. The (possibly adjusted) final content position (see `:horizontal-adjust` and `:vertical-adjust` callbacks, Section 6.2.2) is returned.

6.2.2 Content Callbacks

A `scroll-frame` does not use any callbacks of its own. Instead, an application programmer controls scrolling by defining the following callbacks for the content.

:horizontal-adjust **Callback**

```
(defun     horizontal-adjust-function
           (left)
(declare   (type number left))
(declare   (values number)))
```

Returns an adjusted new left position. Called when the horizontal position of the content is changed. The given left position is guaranteed to be a valid value. The default function for this callback returns the given position unchanged.

:horizontal-calibrate **Callback**

```
(defun     horizontal-calibrate-function ()
(declare   (values minimum maximum pixels-per-unit)))
```

Returns values needed to initialize horizontal scrolling for the `scroll-frame`. The current minimum and maximum for the horizontal position are returned in content units. The pixels-per-unit returned is used for two purposes. First, pixels-per-unit is used to set the indicator size of the horizontal `scroller` to show the number of units visible in the scroll area. Also, if the default `:scroll-to` callback is used, pixels-per-unit defines the size of the content units used for scrolling.

The default function for this callback returns the following values.

minimum:	0
maximum:	`(max 0 (- (contact-width content) (contact-width scroll-area)))`
pixels-per-unit:	1

:horizontal-update **Callback**

```
(defun     horizontal-update-function
           (&key
           position
           minimum
           maximum
           pixels-per-unit)
(declare   (type number position minimum maximum pixels-per-unit)))
```

This callback is added to the content automatically by the `scroll-frame`, not by the application programmer. May be called to inform the `scroll-frame` about changes made by the program in the content's horizontal calibration data. See `:horizontal-calibrate` callback.

:vertical-adjust **Callback**

```
(defun    vertical-adjust-function
          (top)
(declare  (type number top))
(declare  (values number)))
```

Returns an adjusted new top position. Called when the vertical position of the content is changed. The given top position is guaranteed to be a valid value. The default function for this callback returns the given position unchanged.

:vertical-calibrate **Callback**

```
(defun    vertical-calibrate-function ()
(declare  (values minimum maximum pixels-per-unit)))
```

Returns values needed to initialize vertical scrolling for the `scroll-frame`. The current minimum and maximum for the vertical position are returned in content units. The pixels-per-unit returned is used for two purposes. First, pixels-per-unit is used to set the indicator size of the vertical `scroller` to show the number of units visible in the scroll area. Also, if the default `:scroll-to` callback is used, pixels-per-unit defines the size of the content units used for scrolling.

The default function for this callback returns the following values.

minimum:	0
maximum:	`(max 0 (- (contact-height content) (contact-height scroll-area)))`
pixels-per-unit:	1

:vertical-update **Callback**

```
(defun    vertical-update-function
          (&key
          position
          minimum
          maximum
          pixels-per-unit)
(declare  (type number position minimum maximum pixels-per-unit)))
```

This callback is added to the content automatically by the `scroll-frame`, not by the application programmer. May be called to inform the `scroll-frame` about changes made by the program in the content's vertical calibration data. See `:vertical-calibrate` callback.

:scroll-to **Callback**

```
(defun    scroll-to-function
          (left top)
(declare  (type number left top)))
```

Redisplays the content so that the given position (in content units) appears at the upper-left of the scroll-frame. The default function for this callback assumes that content units are pixels and scrolls the content by moving it with respect to the `scroll-frame` parent.

6.3 Table

A `table` is a layout contact which arranges its children (or **members**) into an array of rows and columns. Row/column positions are defined as constraint resources of individual members (see [2] for a complete description of constraint resources).

6.3.1 Table Layout Policy

The layout of a `table` is governed by table constraints, spacing constraints, and member constraints. Table constraints are attributes which describe the rows and columns. The following functions may be used to return or change table constraints.

- `table-column-alignment`
- `table-column-width`
- `table-columns`
- `table-row-alignment`
- `table-row-height`
- `table-same-height-in-row`
- `table-same-width-in-column`

Spacing contraints control the amount of space surrounding rows and columns. The following functions may be used to return or change spacing constraints for a `table`.

- `display-bottom-margin`
- `display-left-margin`
- `display-right-margin`
- `display-top-margin`
- `display-horizontal-space`
- `display-vertical-space`
- `table-separator`

Member constraints may be used to specify the row/column position of an individual member. An application programmer must ensure that member constraints do not conflict with table constraints. In case of such a conflict, table constraints will override member constraints. For example, suppose table constraints specify that a `table` has two columns, but member constraints specify that a member should appear in the third column. In this case, the member column constraint will be ignored and the member will be placed in another column. The following functions may be used to return or change member constraints.

- `table-column`
- `table-row`

6.3.2 Functional Definition

make-table **Function**

```
(defun    make-table
          (&rest initargs
          &key
          (border                 *default-contact-border*)
          (bottom-margin          :default)
          (column-alignment       :left)
          (column-width           :maximum)
          (columns                :maximum)
          foreground
          (horizontal-space       :default)
          (left-margin            :default)
          (right-margin           :default)
          (row-alignment          :bottom)
          (row-height             :maximum)
          (same-height-in-row     :off)
          (same-width-in-column   :off)
          separators
          (top-margin             :default)
          (vertical-space         :default)
          &allow-other-keys)
 (declare (values table)))
```

Creates and returns a `table` contact. The resource specification list of the `table` class defines a resource for each of the initargs above.

The `separators` initarg is a list of row indexes indicating the rows of the `table` that are followed by separators. See `table-separator`, page 102.

display-bottom-margin **Method, table**

```
(defmethod  display-bottom-margin
            ((table table))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-bottom-margin)
            (bottom-margin
            (table table))
(declare    (type (or (integer 0 *) :default) bottom-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the bottom margin. The height of the contact minus the bottom margin size defines the bottom edge of the clipping rectangle used when displaying the source. Setting the bottom margin to `:default` causes the value of `*default-display-bottom-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-horizontal-space **Method, table**

```
(defmethod  display-horizontal-space
            ((table table))
(declare    (values integer)))
```

```
(defmethod  (setf display-horizontal-space)
            (horizontal-space
            (table table))
(declare    (type (or integer :default) horizontal-space))
(declare    (values integer)))
```

Returns or changes the pixel size of the space between columns in the `table`. Setting the horizontal space to `:default` causes the value of `*default-display-horizontal-space*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-left-margin **Method, table**

```
(defmethod  display-left-margin
            ((table table))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-left-margin)
            (left-margin
            (table table))
(declare    (type (or (integer 0 *) :default) left-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the left margin. The left margin size defines the left edge of the clipping rectangle used when displaying the source. Setting the left margin to `:default` causes the value of `*default-display-left-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-right-margin **Method, table**

```
(defmethod  display-right-margin
            ((table table))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-right-margin)
            (right-margin
            (table table))
(declare    (type (or (integer 0 *) :default) right-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the right margin. The width of the contact minus the right margin size defines the right edge of the clipping rectangle used when displaying the source. Setting the right margin to `:default` causes the value of `*default-display-right-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-top-margin **Method, table**

```
(defmethod  display-top-margin
            ((table table))
(declare    (values (integer 0 *))))
```

```
(defmethod  (setf display-top-margin)
            (top-margin
            (table table))
(declare    (type (or (integer 0 *) :default) top-margin))
(declare    (values (integer 0 *))))
```

Returns or changes the pixel size of the top margin. The top margin size defines the top edge of the clipping rectangle used when displaying the source. Setting the top margin to `:default` causes the value of `*default-display-top-margin*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

display-vertical-space **Method, table**

```
(defmethod  display-vertical-space
            ((table table))
(declare    (values integer)))
```

```
(defmethod  (setf display-vertical-space)
            (vertical-space
            (table table))
(declare    (type (or integer :default) vertical-space))
(declare    (values integer)))
```

Returns or changes the pixel size of the space between rows in the `table`. Setting the vertical space to `:default` causes the value of `*default-display-vertical-space*` (converted from points to the number of pixels appropriate for the contact screen) to be used.

table-column-alignment **Method, table**

```
(defmethod  table-column-alignment
            ((table table))
(declare    (values(member :left :center :right))))
```

```
(defmethod  (setf table-column-alignment)
            (alignment
            (table table))
(declare    (type (member :left :center :right) alignment))
(declare    (values (member :left :center :right))))
```

Returns and changes the horizontal alignment of members in each column.

table-column-width **Method, table**

```
(defmethod  table-column-width
            ((table table))
(declare    (values (or (member :maximum) (integer 1 *) list))))
```

```
(defmethod  (setf table-column-width)
            (width
            (table table))
(declare    (type (or (member :maximum) (integer 1 *) list) width))
(declare    (values (or (member :maximum) (integer 1 *) list))))
```

Returns and changes the width of columns in the `table`. An integer column width gives the pixel width used for each column. A `:maximum` column width means that each column will be as wide as its widest member, and thus may differ in width. Column widths may also be specified by a list of the form (*column-width**), where the `i`-th element of the list is the *column-width* for the `i`-th column. A *column-width* element may be either an integer pixel column width or `nil` (meaning `:maximum` column width).

table-columns **Method, table**

```
(defmethod  table-columns
            ((table table))
(declare    (values(or (integer 1 *) (member :none :maximum)))))
```

```
(defmethod  (setf table-columns)
            (columns
            (table table))
(declare    (type (or (integer 1 *) (member :none :maximum)) columns))
(declare    (values (or (integer 1 *) (member :none :maximum)))))
```

Returns and changes the number of columns in the `table`. A `:none` value means that rows are filled without aligning members into columns. A `:maximum` value means that as many columns as possible will be formed.

table-member **Method, table**

```
(defmethod  table-member
            ((table table)
            row
            column)
(declare    (type (integer 0 *) row column))
(declare    (values (or null contact))))
```

Returns the member, if any, at the given row/column position.

table-row-alignment **Method, table**

```
(defmethod  table-row-alignment
            ((table table))
(declare    (values(member :top :center :bottom))))

(defmethod  (setf table-row-alignment)
            (alignment
            (table table))
(declare    (type (member :top :center :bottom) alignment))
(declare    (values (member :top :center :bottom))))
```

Returns and changes the vertical alignment of members in each row.

table-row-height **Method, table**

```
(defmethod  table-row-height
            ((table table))
(declare    (values(or (member :maximum) (integer 1 *) list))))

(defmethod  (setf table-row-height)
            (height
            (table table))
(declare    (type (or (member :maximum) (integer 1 *) list) height))
(declare    (values (or (member :maximum) (integer 1 *) list))))
```

Returns and changes the height of rows in the `table`. An integer row height gives the pixel height used for each row. A `:maximum` row height means that each row will be as high as its highest member, and thus may differ in height. Row heights may also be specified by a list of the form (*row-height**), where the `i`-th element of the list is the *row-height* for the `i`-th row. A *row-height* element may be either an integer pixel row height or `nil` (meaning `:maximum` row height).

table-same-height-in-row **Method, table**

```
(defmethod  table-same-height-in-row
            ((table table))
(declare    (values(member :on :off))))

(defmethod  (setf table-same-height-in-row)
            (switch
            (table table))
(declare    (type (member :on :off) switch))
(declare    (values (member :on :off))))
```

Returns and changes whether the members in a row will be set to have the same height.

table-same-width-in-column Method, table

```
(defmethod  table-same-width-in-column
            ((table table))
(declare    (values(member :on :off))))
```

```
(defmethod  (setf table-same-width-in-column)
            (switch
            (table table))
(declare    (type (member :on :off) switch))
(declare    (values (member :on :off))))
```

Returns and changes whether the members in a column will be set to have the same width.

table-separator Method, table

```
(defmethod  table-separator
            ((table table)
            row)
(declare    (type (integer 0 *)            row))
(declare    (values (member :on :off))))
```

```
(defmethod  (setf table-separator)
            (switch
            (table table)
            row)
(declare    (type (member :on :off)        switch)
            (type (integer 0 *)            row))
(declare    (values (member :on :off))))
```

Returns and changes the presence of a separator after the given `row` of the `table`. For example, `(setf (table-separator table 0) :on)` causes a separator to appear between rows 0 and 1 of the `table`.

A separator is some kind of visual separation between adjacent table rows. For example, a separator could be represented by a thin line or by extra space. However, the exact visual form of a separator is implementation-dependent.

6.3.3 Constraint Resources

The following functions may be used to return or (with `setf`) to change the constraint resources of the members of a `table` contact.

table-column **Function**

```
(defun    table-column
          (member)
(declare  (type contact                    member))
(declare  (values (or null (integer 0 *)))))
```

Returns or (with `setf`) changes the column position of the table member. A `nil` value means that the `table` may choose any convenient column.

table-row **Function**

```
(defun    table-row
          (member)
(declare  (type contact                    member))
(declare  (values (or null (integer 0 *)))))
```

Returns or (with `setf`) changes the row position of the table member. A `nil` value means that the `table` may choose any convenient row.

Chapter 7

Dialogs

The term **dialog** refers generally to a type of composite which presents several application data items for interaction and reports a user's response. In some cases, a user can respond by modifying the presented data. In CLIO, dialogs are `shell` subclasses that represent top-level contacts.

CLIO defines dialogs for the following types of user interactions.

`command`	Presents a set of related value controls and a set of command controls which operate on the values. This is the most general type of dialog.
`confirm`	A simple dialog which presents a message and allows a user to enter a "yes or no" response.
`menu`	Allows a user to select from a set of choice items.
`property-sheet`	Presents a set of related values for editing and allows a user to accept or cancel any changes.

7.1 Accepting, Canceling, and Initializing Dialogs

When a user terminates interaction with a dialog, he is said to **exit** the dialog. In general, the application program determines the visual effect of exiting a dialog. Applications often use dialogs as "pop-ups," in which case exiting the dialog causes it to be "popped down" (i.e. its state becomes `:withdrawn`).

A dialog often contains one or more controls that allow a user to indicate a positive (or "accept") response and a negative (or "cancel") response. Most CLIO dialogs automatically create accept and cancel controls that use the generic functions `dialog-accept` and `dialog-cancel`. An accept response causes the `dialog-accept` function to be invoked and exits the dialog. A cancel response causes the `dialog-cancel` function to be invoked and exits the dialog.

In general, dialogs use the following callbacks for handling user responses and for initialization.

`:accept` Invoked by the `dialog-accept` function when a user accepts and exits the dialog.

`:cancel` Invoked by the `dialog-cancel` function when a user cancels and exits the dialog.

`:initialize` Invoked by the `shell-mapped` function when the dialog becomes `:mapped` (see [2], shell contacts).

`:verify` Invoked when a user accepts the dialog. This callback is used only when the dialog presents data which the user may change. This callback can be used to enforce validity constraints on user changes. By default, this callback is undefined.

For dialogs that contain values to be changed, an application programmer may also define the following callbacks for each value. These optional callbacks may be used to control the effect of user changes on individual values and are invoked only if the corresponding callback is not defined for the dialog itself.

`:accept` Invoked by the `dialog-accept` function if no `:accept` callback is defined for the dialog.

`:cancel` Invoked by the `dialog-cancel` function if no `:cancel` callback is defined for the dialog.

`:initialize` Invoked by the `shell-mapped` function if no `:initialize` callback is defined for the dialog.

Note that the application programmer has several options for controlling a dialog:

- Implement accept semantics via callbacks for individual members or via callbacks for the dialog or both (or neither).
- Implement cancel recovery semantics for edited values via "change-immediately-then-undo-later" or via "postpone-changes-until-accept."
- Implement the (re)initialization of a value via its `:initialize` callback or via its `:cancel` callback or not at all.

7.2 Presenting Dialogs

An application program should call the `present-dialog` function to present a `command` dialog at a specific location, in response to a user or program event. The `present-dialog` for each dialog class encapsulates all implementation-dependent rules for positioning the dialog and initializing its interaction.

7.3 Command

A `command` is a `transient-shell` which presents a set of related values to be viewed or changed by the user. Also presented are a set of controls which represent commands that operate on the values. In general, the application programmer is responsible for programming command controls to exit the dialog appropriately. For convenience, optional default accept and cancel controls can be created automatically, although their exact appearance and behavior are implementation-dependent.

The application programmer may also also identify a **default control**. A `command` may highlight the default control or otherwise expedite its selection by the user, although the exact treatment of the default control is implementation-dependent.

A `command` contains two children representing different regions. Values to be viewed or modified appear in the region called the **command area**. The command area is a layout contact, such as a `form`, which controls the layout of values. Command controls appear in the region called the **control area**. The control area is also a layout, such as a `table`. Command controls are presented by children of the control area. Typically, command controls are `action-button` objects.

A `command` uses the `:initialize` callback for initialization. If the default accept control is specified, then the `:accept` and `:verify` callbacks are used. If the default cancel control is specified, then the `:cancel` callback is used. See Section 7.1.

7.3.1 Functional Definition

The `command` class is a subclass of the `transient-shell` class. All `transient-shell` accessors and initargs may be used to operate on a `command`. See [2], `shell` contacts.

make-command **Function**

```
(defun    make-command
          (&rest initargs
          &key
          (border                    *default-contact-border*)
          (command-area              'make-table)
          (control-area              'make-table)
          (default-accept            :on)
          (default-cancel            :on)
          default-control
          foreground
          &allow-other-keys)
(declare  (type (or function list)  command-area control-area))
(declare  (values command)))
```

Creates and returns a `command` contact. The resource specification list of the `command` class defines a resource for each of the initargs above.

The `command-area` and `control-area` arguments specify the constructor function and (optionally) initial attributes for the command area and the control area, respectively. Each of these arguments may be either a function or a list of the form (*constructor* . *initargs*), where *initargs* is a list of keyword/value pairs allowed by the *constructor* function.

command-area **Method, command**

```
(defmethod  command-area
            ((command command))
(declare    (values composite)))
```

Returns the command area composite. Values to be presented or edited are represented by children of the command area.

command-control-area **Method, command**

```
(defmethod  command-control-area
            ((command command))
(declare    (values composite)))
```

Returns the control area composite. Command controls are represented by children of the control area.

command-default-accept **Method, command**

```
(defmethod  command-default-accept
            ((command command))
(declare    (values (or (member :on :off) string))))
```

```
(defmethod  (setf command-default-accept)
            (switch
            (command command))
(declare    (type (or (member :on :off) stringable)   switch))
(declare    (values (or (member :on :off) string))))
```

Returns (and with `setf`) changes whether the default accept control is used. If `:on`, then the default accept control is used with an implementation-dependent label. If a string, then the default accept control is used and labelled with the given string. If `:off`, then the default accept control is not used.

command-default-cancel **Method, command**

```
(defmethod  command-default-cancel
            ((command command))
(declare    (values (or (member :on :off) string))))
```

```
(defmethod  (setf command-default-cancel)
            (switch
            (command command))
(declare    (type (or (member :on :off) stringable)   switch))
(declare    (values (or (member :on :off) string))))
```

Returns and (with `setf`) changes whether the default cancel control is used. If `:on`, then the default cancel control is used with an implementation-dependent label. If a string, then the default cancel control is used and labelled with the given string. If `:off`, then the default cancel control is not used.

dialog-accept **Method, command**

```
(defmethod  dialog-accept
            ((command command)))
```

Called when the user accepts and exits the `command`, using the default accept control. The primary method invokes the `:accept` callback for the `command`, if defined; otherwise, the `:accept` callback is invoked for each member of the command area.

dialog-cancel **Method, command**

```
(defmethod  dialog-cancel
            ((command command)))
```

Called when the user cancels and exits the `command`, using the default cancel control. The primary method invokes the `:cancel` callback for the `command`, if defined; otherwise, the `:cancel` callback is invoked for each member of the command area.

dialog-default-control **Method, command**

```
(defmethod  dialog-default-control
            ((command command))
(declare    (values symbol)))
```

```
(defmethod  (setf dialog-default-control)
            (control
            (command command))
(declare    (type symbol control))
(declare    (values symbol)))
```

Returns and (with `setf`) changes the name of the default control. By default, the name of the default control is either `:accept` (if a default accept control exists) or the name of the first member of the control area.

present-dialog **Method, command**

```
(defmethod  present-dialog
            ((command command)
            &key
            x
            y
            button
            state)
(declare    (type (or int16 null)        x)
            (type (or int16 null)        y)
            (type (or button-name null)  button)
            (type (or mask16 null)       state)))
```

Presents the `command` at the position given by `x` and `y`. The values of `x` and `y` are treated as hints, and the exact position where the `command` will appear is implementation-dependent. By default, `x` and `y` are determined by the current pointer position.

If the `command` is presented in response to a pointer button event, then `button` should specify the button pressed or released. Valid button names are `:button-1`, `:button-2`, `:button-3`, `:button-4`, and `:button-5`. If given, `state` specifies the current state of the pointer buttons and modifier keys.

shell-mapped **Method, command**

```
(defmethod  shell-mapped
            ((command command)))
```

Called before `command` becomes `:mapped` (See [2], `shell` contacts). The primary method invokes the `:initialize` callback for the `command`, if defined; otherwise, the `:initialize` callback is invoked for each member of the command area.

7.3.2 Callbacks

An application programmer may define the following callbacks for a `command`.

:accept **Callback, command**

```
(defun  accept-function  () )
```

Invoked when a user accepts and exits the `command`, using the default accept control. This function should implement the application response to any user changes to the `command`. This callback is used if the default accept control is specified. However, if no default accept control is specified, then use of the `:accept` callback is implementation-dependent.

:cancel **Callback, command**

```
(defun  cancel-function  () )
```

Invoked when a user cancels and exits the `command`, using the default cancel control. This function should implement the application response to cancelling any user changes to the `command`. This callback is used if the default cancel control is specified. However, if no default cancel control is specified, then use of the `:cancel` callback is implementation-dependent.

:initialize **Callback, command**

```
(defun  initialize-function  () )
```

Invoked when the `command` becomes `:mapped`. This function should implement any initialization needed for the `command` before it becomes `:mapped`.

:verify **Callback, command**

```
(defun    verify-function
          (command)
(declare  (type command command))
(declare  (values boolean string (or null contact))))
```

If defined, this callback is invoked when a user accepts the `command`. This callback can be used to enforce validity constraints on user changes. If all user changes are valid, then the first return value is true and the `command` is accepted and exited. Otherwise, the first return value is `nil` and the `command` is not exited. If the first return value is `nil`, then two other values are returned. The second return value is an error message string to be displayed by the `command`. The third value is the member contact reporting the error, or `nil`.

If no `:verify` callback is defined, then the `command` is accepted and exited immediately.

This callback is used if the default accept control is specified. However, if no default accept control is specified, then use of the `:verify` callback is implementation-dependent.

7.3.3 Member Callbacks

An application programmer may define the following callbacks for members of the command area. These callbacks may or may not be used, depending on the functions for the `command` callbacks described in Section 7.3.2. See also the description for the `dialog-accept` and `dialog-cancel` methods.

:accept **Callback**

```
(defun  accept-function  () )
```

Invoked by the `dialog-accept` function if no `:accept` callback is defined for the `command`. This function should implement the application response to any user changes to the individual member.

:cancel **Callback**

```
(defun cancel-function () )
```

Invoked by the `dialog-cancel` function if no `:cancel` callback is defined for the `command`. This function should implement the application response to cancelling any user changes to the individual member.

:initialize **Callback**

```
(defun initialize-function () )
```

Invoked by the `shell-mapped` function if no `:initialize` callback is defined for the `command`. This function should implement any initialization needed for the individual member before the `command` becomes `:mapped`.

7.4 Confirm

A `confirm` is an `override-shell` which presents a message and allows a user to enter a "yes or no" response.

A `confirm` contains an accept control and, optionally, a cancel control. If only an accept control is specified, then a `confirm` accepts only a single response, indicating that a user has seen the message. The accept and cancel controls are created automatically, and their exact appearance and behavior are implementation-dependent. However, the application programmer can specify text labels to be displayed by the accept and cancel controls.

The application programmer may also also identify a **default control**. A `confirm` may highlight the default control or otherwise expedite its selection by the user, although the exact treatment of the default control is implementation-dependent.

A `confirm` uses the `:initialize`, `:accept`, and `:cancel` callbacks. See Section 7.1.

The `confirm-p` function is a simplified interface for presenting a `confirm` dialog and returning the user response as a boolean value.

7.4.1 Functional Definition

The `confirm` class is a subclass of the `override-shell` class. All `transient-shell` accessors and initargs may be used to operate on a `confirm`. See [2], `shell` contacts.

make-confirm **Function**

```
(defun    make-confirm
          (&rest initargs
          &key
          accept-label
          (accept-only        :off)
          (border             *default-contact-border*)
          cancel-label
          (default-control    :accept)
          foreground
          message
          near
          &allow-other-keys)
(declare  (values confirm)))
```

Creates and returns a `confirm` contact. The resource specification list of the `confirm` class defines a resource for each of the initargs above.

confirm-accept-label **Method, confirm**

```
(defmethod  confirm-accept-label
            ((confirm confirm))
(declare    (values string)))
```

```
(defmethod  (setf confirm-accept-label)
            (accept-label
            (confirm confirm))
(declare    (type stringable accept-label))
(declare    (values string)))
```

Returns or changes the label displayed by the accept control. The default accept label is implementation-dependent.

confirm-accept-only **Method, confirm**

```
(defmethod  confirm-accept-only
            ((confirm confirm))
(declare    (values (member :on :off))))
```

```
(defmethod  (setf confirm-accept-only)
            (accept-only
            (confirm confirm))
(declare    (type (member :on :off) accept-only))
(declare    (values (member :on :off))))
```

Returns or changes the presence of the cancel control. If :on, then no cancel control is presented.

confirm-cancel-label **Method, confirm**

```
(defmethod  confirm-cancel-label
            ((confirm confirm))
(declare    (values string)))
```

```
(defmethod  (setf confirm-cancel-label)
            (cancel-label
            (confirm confirm))
(declare    (type stringable cancel-label))
(declare    (values string)))
```

Returns or changes the label displayed by the cancel control. The default cancel label is implementation-dependent.

confirm-message — Method, confirm

```
(defmethod  confirm-message
            ((confirm confirm))
(declare    (values string)))
```

```
(defmethod  (setf confirm-message)
            (message
            (confirm confirm))
(declare    (type string message))
(declare    (values string)))
```

Returns or changes the message displayed by the `confirm`.

confirm-near — Method, confirm

```
(defmethod  confirm-near
            ((confirm confirm))
(declare    (values window)))
```

```
(defmethod  (setf confirm-near)
            (near
            (confirm confirm))
(declare    (type window near))
(declare    (values window)))
```

Returns or changes the position where the `confirm` is displayed. When it is `:mapped`, the `confirm` appears near the given `window`; the exact meaning of "near" is implementation-dependent. In general, the "near window" should be the one receiving the user input that caused the `confirm` to become `:mapped`. Typically, changing the "near window" changes the values of `x` and `y` for the `confirm`.

By default, the "near window" of a `confirm` is itself. This is a special case indicating that the position of the `confirm` is determined normally, by its `x` and `y` position

confirm-p — Function

```
(defun     confirm-p
           (&rest initargs
           &key near
           &allow-other-keys)
(declare   (type contact       near))
(declare   (values boolean)))
```

Presents a `confirm` and waits for a user to exit the dialog. Returns true if the user accepts and `nil` if the user cancels.

The attributes of the `confirm` are specified by the `initargs`, which can contain any initarg allowed by `make-confirm` (except `:callbacks`). The `:callbacks` initarg is not allowed because `:accept` and `:cancel` callbacks are defined by `confirm-p`. The `near` argument is required.

dialog-accept — Method, confirm

```
(defmethod  dialog-accept
            ((confirm confirm)))
```

Called when the user accepts and exits the `confirm`. The primary method invokes the `:accept` callback for the `confirm`.

dialog-cancel — Method, confirm

```
(defmethod  dialog-cancel
            ((confirm confirm)))
```

Called when the user cancels and exits the `confirm`. The primary method invokes the `:cancel` callback for the `confirm`.

dialog-default-control — Method, confirm

```
(defmethod  dialog-default-control
            ((confirm confirm))
(declare    (values (member :accept :cancel))))
```

```
(defmethod  (setf dialog-default-control)
            (control
            (confirm confirm))
(declare    (type (member :accept :cancel)      control))
(declare    (values (member :accept :cancel))))
```

Returns and (with `setf`) changes the name of the default control.

present-dialog **Method, confirm**

```
(defmethod  present-dialog
            ((confirm confirm)
            &key
            x
            y
            button
            state)
(declare    (type (or int16 null)        x)
            (type (or int16 null)        y)
            (type (or button-name null)  button)
            (type (or mask16 null)       state)))
```

Presents the `confirm` at the position given by `x` and `y`. The values of `x` and `y` are treated as hints, and the exact position where the `confirm` will appear is implementation-dependent. By default, `x` and `y` are determined by the current pointer position.

If the `confirm` is presented in response to a pointer button event, then `button` should specify the button pressed or released. Valid button names are `:button-1`, `:button-2`, `:button-3`, `:button-4`, and `:button-5`. If given, `state` specifies the current state of the pointer buttons and modifier keys.

shell-mapped **Method, confirm**

```
(defmethod  shell-mapped
            ((confirm confirm)))
```

Called before `confirm` becomes `:mapped` (See [2], `shell` contacts). The primary method invokes the `:initialize` callback for the `confirm`.

7.4.2 Callbacks

An application programmer may define the following callbacks for a `confirm`.

:accept **Callback, confirm**

```
(defun  accept-function  () )
```

Invoked when a user accepts and exits the `confirm`. This function should implement the application effect of a user's accept response.

:cancel **Callback, confirm**

```
(defun  cancel-function  () )
```

Invoked when a user cancels and exits the `confirm`. This function should implement the application effect of a user's cancel response.

:initialize **Callback, confirm**

```
(defun  initialize-function  () )
```

Invoked when the `confirm` becomes `:mapped`. This function should implement any initialization needed for the `confirm` before it becomes `:mapped`.

7.5 Menu

A menu is an override-shell which presents a choice contact and allows a user to select from a set of choice items. Menu items are added as choice items of this choice contact, which is created automatically. An application programmer can define the class and initial attributes for this choice contact when the menu is created. See Chapter 5 for a description of choice contact classes.

A menu has a title defined by a text string.

The choice item controls that appear in menus are called **menu items**. Menu items may exhibit a distinctive appearance and operation which reflect their special role as parts of menus. CLIO defines two classes of menu items — action-item and menu-item.

A menu uses the :initialize callback for initialization.

7.5.1 Functional Definition

The menu class is a subclass of the override-shell class. All override-shell accessors and initargs may be used to operate on a menu. See [2], shell contacts.

make-menu **Function**

```
(defun    make-menu
          (&rest initargs
          &key
          (border                      *default-contact-border*)
          (choice                      'make-choices)
          foreground
          (title                       "")
          &allow-other-keys)
(declare  (type (or function list)  choice))
(declare  (values menu)))
```

Creates and returns a menu contact. The resource specification list of the menu class defines a resource for each of the initargs above.

The choice argument specifies the constructor and (optionally) initial attributes for the choice contact of the new menu. This argument may be either a constructor function or a list of the form (*constructor* . *initargs*), where *initargs* is a list of keyword/value pairs allowed by the *constructor* function.

menu-choice — Method, menu

```
(defmethod  menu-choice
            ((menu menu))
(declare    (values choice-contact)))
```

Returns the choice contact for the `menu`. This choice contact is created automatically and its class can be set only by `make-menu`. Menu items are defined by creating choice items for this choice contact.

menu-title — Method, menu

```
(defmethod  menu-title
            ((menu menu))
(declare    (values title-contact)))
```

```
(defmethod  (setf menu-title)
            (title
            (menu menu))
(declare    (type stringable   title))
(declare    (values string)))
```

Returns or (with `setf`) changes the title string of the `menu`.

present-dialog — Method, menu

```
(defmethod  present-dialog
            ((menu menu)
            &key
            x
            y
            button
            state)
(declare    (type (or int16 null)        x)
            (type (or int16 null)        y)
            (type (or button-name null)  button)
            (type (or mask16 null)       state)))
```

Presents the `menu` at the position given by `x` and `y`. The values of `x` and `y` are treated as hints, and the exact position where the `menu` will appear is implementation-dependent. By default, `x` and `y` are determined by the current pointer position.

If the `menu` is presented in response to a pointer button event, then `button` should specify the button pressed or released. Valid button names are `:button-1`, `:button-2`, `:button-3`, `:button-4`, and `:button-5`. If given, `state` specifies the current state of the pointer buttons and modifier keys.

shell-mapped **Method, menu**

```
(defmethod  shell-mapped
            ((menu menu)))
```

Called before `menu` becomes `:mapped` (See [2], `shell` contacts). The primary method invokes the `:initialize` callback for the `menu`, if defined; otherwise, the `:initialize` callback is invoked for each menu item.

7.5.2 Callbacks

An application programmer may define the following callback for a `menu`.

:initialize **Callback**

```
(defun  initialize-function  () )
```

Invoked when the `menu` becomes `:mapped`. This function should implement any initialization needed for the `menu` before it becomes `:mapped`.

7.5.3 Item Callbacks

An application programmer may define the following callback for each item in a `menu`.

:initialize **Callback**

```
(defun  initialize-function  () )
```

Invoked by the `shell-mapped` function if no `:initialize` callback is defined for the `menu`. This function should implement any initialization needed for the individual item before the `menu` becomes `:mapped`.

7.6 Property Sheet

A property-sheet is a transient-shell which presents a set of related values that can be changed by a user. Typically, a property-sheet allows a user to modify the properties, or attributes, of a specific application object. A property-sheet contains controls which allow a user either to accept or to cancel any changes to the values. Accept and cancel controls are created automatically, and their exact appearance and behavior are implementation-dependent.

The application programmer may also also identify a **default control**. A property-sheet may highlight the default control or otherwise expedite its selection by the user, although the exact treatment of the default control is implementation-dependent.

The content of a property-sheet is called the **property area**. The property area is a layout contact, such as a form. Property values are presented by contacts which are children (or **members**) of the property area.

A property-sheet uses the :initialize, :accept, :cancel, and :verify callbacks. See Section 7.1.

7.6.1 Functional Definition

The property-sheet class is a subclass of the transient-shell class. All transient-shell accessors and initargs may be used to operate on a property-sheet. See [2], shell contacts.

make-property-sheet **Function**

```
(defun    make-property-sheet
          (&rest initargs
          &key
          (border                    *default-contact-border*)
          (default-control           :accept)
          foreground
          (property-area             'make-table)
          &allow-other-keys)
(declare  (type (or symbol list)     property-area))
(declare  (values property-sheet)))
```

Creates and returns a property-sheet contact. The resource specification list of the property-sheet class defines a resource for each of the initargs above.

The property-area argument specifies the constructor and (optionally) initial attributes for the property area. This argument may be either a constructor function or a list of the form (*constructor* . *initargs*), where *initargs* is a list of keyword/value pairs allowed by the *constructor* function.

dialog-accept **Method, property-sheet**

```
(defmethod  dialog-accept
            ((property-sheet property-sheet)))
```

Called when the user accepts and exits the `property-sheet`. The primary method invokes the `:accept` callback for the `property-sheet`, if defined; otherwise, the `:accept` callback is invoked for each member of the property area.

dialog-cancel **Method, property-sheet**

```
(defmethod  dialog-cancel
            ((property-sheet property-sheet)))
```

Called when the user cancels and exits the `property-sheet`. The primary method invokes the `:cancel` callback for the `property-sheet`, if defined; otherwise, the `:cancel` callback is invoked for each member of the property area.

dialog-default-control **Method, property-sheet**

```
(defmethod  dialog-default-control
            ((property-sheet property-sheet))
(declare    (values (member :accept :cancel))))
```

```
(defmethod  (setf dialog-default-control)
            (control
            (property-sheet property-sheet))
(declare    (type (member :accept :cancel)       control))
(declare    (values (member :accept :cancel))))
```

Returns and (with `setf`) changes the name of the default control. By default, the name of the default control is either `:accept` (if a default accept control exists) or the name of the first member of the control area.

present-dialog **Method, property-sheet**

```
(defmethod  present-dialog
            ((property-sheet property-sheet)
            &key
            x
            y
            button
            state)
(declare    (type (or int16 null)          x)
            (type (or int16 null)          y)
            (type (or button-name null)    button)
            (type (or mask16 null)         state)))
```

Presents the `property-sheet` at the position given by `x` and `y`. The values of `x` and `y` are treated as hints, and the exact position where the `property-sheet` will appear is implementation-dependent. By default, `x` and `y` are determined by the current pointer position.

If the `property-sheet` is presented in response to a pointer button event, then `button` should specify the button pressed or released. Valid button names are `:button-1`, `:button-2`, `:button-3`, `:button-4`, and `:button-5`. If given, `state` specifies the current state of the pointer buttons and modifier keys.

shell-mapped **Method, property-sheet**

```
(defmethod  shell-mapped
            ((property-sheet property-sheet)))
```

Called before `property-sheet` becomes `:mapped` (See [2], `shell` contacts). The primary method invokes the `:initialize` callback for the `property-sheet`, if defined; otherwise, the `:initialize` callback is invoked for each member of the property area.

property-sheet-area **Method, property-sheet**

```
(defmethod  property-sheet-area
            ((property-sheet property-sheet))
(declare    (values contact)))
```

Returns the property area contact.

7.6.2 Callbacks

An application programmer may define the following callbacks for a `property-sheet`.

:accept **Callback, property-sheet**

```
(defun  accept-function  () )
```

Invoked when a user accepts and exits the `property-sheet`. This function should implement the application response to any user changes to the `property-sheet`.

:cancel **Callback, property-sheet**

```
(defun  cancel-function  () )
```

Invoked when a user cancels and exits the `property-sheet`. This function should implement the application response to cancelling any user changes to the `property-sheet`.

:initialize **Callback, property-sheet**

```
(defun  initialize-function  () )
```

Invoked when the `property-sheet` becomes `:mapped`. This function should implement any initialization needed for the `property-sheet` before it becomes `:mapped`.

:verify **Callback, property-sheet**

```
(defun     verify-function
           (property-sheet)
(declare   (type property-sheet property-sheet))
(declare   (values boolean string (or null contact))))
```

If defined, this callback is invoked when a user accepts the `property-sheet`. This callback can be used to enforce validity constraints on user changes. If all user changes are valid, then the first return value is true and the `property-sheet` is accepted and exited. Otherwise, the first return value is `nil` and the `property-sheet` is not exited. If the first return value is `nil`, then two other values are returned. The second return value is an error message string to be displayed by the `property-sheet`. The third value is the member contact reporting the error, or `nil`.

If no `:verify` callback is defined, then the `property-sheet` is accepted and exited immediately.

7.6.3 Member Callbacks

An application programmer may define the following callbacks for members of the property area. These callbacks may or may not be used, depending on the functions for the `property-sheet` callbacks described in Section 7.6.2.

:accept **Callback**

```
(defun accept-function () )
```

Invoked by the `dialog-accept` function if no `:accept` callback is defined for the `property-sheet`. This function should implement the application response to any user changes to the individual member.

:cancel **Callback**

```
(defun cancel-function () )
```

Invoked by the `dialog-cancel` function if no `:cancel` callback is defined for the `property-sheet`. This function should implement the application response to cancelling any user changes to the individual member.

:initialize **Callback**

```
(defun initialize-function () )
```

Invoked by the `shell-mapped` function if no `:initialize` callback is defined for the `property-sheet`. This function should implement any initialization needed for the individual member before the `property-sheet` becomes `:mapped`.

Chapter 8

General Features

8.1 Utilities

This section describes various utility functions defined by CLIO.

point-pixels **Function**

```
(defun    point-pixels
          (screen
          &optional
          (number 1)
          (dimension :vertical))
(declare  (type screen screen)
          (type number number)
          (type (member :horizontal :vertical) dimension))
(declare  (values (integer 0 *))))
```

Returns the number of pixels represented by the given `number` of points, in either the `:vertical` or `:horizontal` dimension of the `screen`.

pixel-points **Function**

```
(defun    pixel-points
          (screen
          &optional
          (number 1)
          (dimension :vertical))
(declare  (type screen screen)
          (type number number)
          (type (member :horizontal :vertical) dimension))
(declare  (values number)))
```

Returns the number of points represented by the given `number` of pixels, in either the `:vertical` or `:horizontal` dimension of the `screen`.

inch-pixels **Function**

```
(defun    inch-pixels
          (screen
          &optional
          (number 1)
          (dimension :vertical))
(declare  (type screen screen)
          (type number number)
          (type (member :horizontal :vertical) dimension))
(declare  (values (integer 0 *))))
```

Returns the number of pixels represented by the given `number` of inches, in either the `:vertical` or `:horizontal` dimension of the `screen`.

pixel-inches **Function**

```
(defun    pixel-inches
          (screen
          &optional
          (number 1)
          (dimension :vertical))
(declare  (type screen screen)
          (type number number)
          (type (member :horizontal :vertical) dimension))
(declare  (values number)))
```

Returns the number of inches represented by the given `number` of pixels, in either the `:vertical` or `:horizontal` dimension of the `screen`.

millimeter-pixels **Function**

```
(defun    millimeter-pixels
          (screen
          &optional
          (number 1)
          (dimension :vertical))
(declare  (type screen screen)
          (type number number)
          (type (member :horizontal :vertical) dimension))
(declare  (values (integer 0 *))))
```

Returns the number of pixels represented by the given `number` of millimeters, in either the `:vertical` or `:horizontal` dimension of the `screen`.

pixel-millimeters **Function**

```
(defun    pixel-millimeters
          (screen
          &optional
          (number 1)
          (dimension :vertical))
(declare  (type screen screen)
          (type number number)
          (type (member :horizontal :vertical) dimension))
(declare  (values number)))
```

Returns the number of millimeters represented by the given `number` of pixels, in either the `:vertical` or `:horizontal` dimension of the `screen`.

8.2 Global Variables and Type Specifiers

gravity **Type**

```
(deftype  gravity   ()
          '(member  :north-west :north :north-east
                    :west :center :east
                    :south-west :south :south-east))
```

Describes an alignment position used to display contact contents.

default-choice-font **Variable**

```
(defparameter  *default-choice-font*
               "*-*-*-*-r-*--12-*-*-*-p-*-iso8859-1")
```

The default font used for text labels in choice items.

default-contact-border **Variable**

```
(defparameter  *default-contact-border*  :black)
```

The default border color for CLIO contacts.

default-contact-foreground **Variable**

```
(defparameter  *default-contact-foreground*  :black)
```

The default initial foreground color for CLIO contacts.

default-display-bottom-margin **Variable**

```
(defparameter  *default-display-bottom-margin*  0)
```

The default bottom margin for CLIO contacts, given in points. This value must be converted into pixel units appropriate for the given display.

default-display-left-margin **Variable**

```
(defparameter  *default-display-left-margin*  0)
```

The default left margin for CLIO contacts, given in points. This value must be converted into pixel units appropriate for the given display.

default-display-right-margin **Variable**

```
(defparameter  *default-display-right-margin*  0)
```

The default right margin for CLIO contacts, given in points. This value must be converted into pixel units appropriate for the given display.

default-display-top-margin **Variable**

```
(defparameter  *default-display-top-margin*  0)
```

The default top margin for CLIO contacts, given in points. This value must be converted into pixel units appropriate for the given display.

default-display-horizontal-space **Variable**

```
(defparameter  *default-display-horizontal-space*  0)
```

The default horizontal spacing for CLIO layout contacts, given in points. This value must be converted into pixel units appropriate for the given display.

default-display-vertical-space **Variable**

```
(defparameter  *default-display-vertical-space*  0)
```

The default vertical spacing for CLIO layout contacts, given in points. This value must be converted into pixel units appropriate for the given display.

default-display-text-font **Variable**

```
(defparameter  *default-display-text-font*
               "*-*-*-*-r-*--12-*-*-*-p-*-iso8859-1")
```

The default font used by CLIO contacts.

8.3 Selections for Interclient Communication

Certain CLIO contacts for display and editing support the interchange of data among different clients via **selections**. The X selection mechanism is defined by the X Window System Protocol[4].

The use of selections by CLIO contacts conforms to the conventions described by the X Window System Inter-Client Communications Convention Manual (ICCCM)[3].

In general, display and editing contacts supply user operations which set the value of certain standard selections to contact data. That is, a user can make a display/editing contact the owner of a standard selection and can cause the contact then to return selected contact data in response to SelectionRequest events. The specific user operations which control selections depend on the contact class.

In order to conform to ICCCM, display and editing contacts support the following targets for all supported selections (see [3] for a complete description of these required targets).

`:multiple`	Return a list containing the selection value in multiple target formats.
`:targets`	Return a list of supported target formats.
`:timestamp`	Return a timestamp giving the time when selection ownership was acquired.

Chapter 9

Acknowledgements

Major contributions to the CLIO design came from the other members of the team responsible for its initial implementation:

Javier Arellano	Texas Instruments
William Cohagan	William Cohagan Inc.
Paul Fuqua	Texas Instruments
Eric Mielke	Texas Instruments
Mark Young	Texas Instruments

In addition, we wish to thank the following individuals, who were among our first users and who suggested many significant improvements.

Patrick Hogan	Texas Instruments
Aaron Larson	Honywell Systems Research Center
Jill Nicola	Texas Instruments

Bibliography

[1] Bobrow, Daniel G., et al. The Common Lisp Object System Specification (X3J13-88-002). American National Standards Institute, June, 1988.

[2] Kimbrough, Kerry and Oren, LaMott. Common Lisp User Interface Environment, Version 7.1 (November, 1989).

[3] Rosenthal, David S. H. X11 Inter-Client Communication Conventions Manual, Version 1 (January, 1990).

[4] Scheifler, Robert W. The X Window System Protocol, Version 11, Revision 3.

[5] Scheifler, Robert W., et al. CLX — Common Lisp X Interface, Release 4 (January 1990).

Index

A

B

C

F

G

I

L

M

O

P

R

S

T

V