Digitool

For Macintosh Common Lisp versions 3.1 & 4.0

Macintosh Common Lisp Reference

Digitool

030-1959-C
Developer Technical Publications

Digitool, Inc.

Printed in the United States of America.

One Main Street,
Cambridge, MA 02142
617-441-5000

Simultaneously published in the United States and Canada.

Contents

Appendix A: Implementation Notes / 617

Figures and tables

Introduction:

About This Book

Contents

This introduction describes the syntax and notational conventions used in this reference.

Documentation conventions

This manual follows specific conventions for fonts, notation, Lisp syntax, and definition formats.

Courier font

In this manual, all MCL code appears in Courier font. When an MCL interaction is shown, what you type appears in boldface Courier and what MCL responds with is shown in regular Courier.

Courier font always represents exactly what is typed into and returned by the program, with one exception. In the syntax of definitions, words in Courier beginning with an ampersand (lambda list keywords) indicate certain standard parts of the body of a definition. For example, `&key` indicates that the items following it are keywords, `&optional` indicates that all arguments past that point are optional, and so on.

See *Common Lisp: The Language* for a full description of this syntax.

Italics

Italics indicate parameter names and place holders (words that you replace on the screen with an actual value). For example, when using the function `my-function`, you see the definition

`my-function` *my-arg* `&optional` *more-info* `&key :test`

Type the words `my-function` and `:test` as they appear, but substitute some value for *my-arg* and *more-info*.

Definition formats

The same definition format is used for functions, methods, variables, named constants, classes, macros, and special forms.

The header indicates the name and type of the definition. In the case of a function, for example, the first line indicates the name of the function and the fact that it is a function. Its syntax appears below its name and type; it is described; its parameters are defined; finally, in many cases, it is used in an example.

A definition format always includes a description of the item being defined; where appropriate, it also shows its syntax, includes a description of its arguments, and gives an example of its use. Here are some abridged examples of definition formats.

`pop-up-menu` [*Class name*]

Description This is the class of pop-up menus, built on the classes `menu` and `dialog-item`.

`*fred-default-font-spec*` [*Variable*]

Description The `*fred-default-font-spec*` variable specifies which font is used when new Fred (editor) windows are opened. The initial value is `("Monaco" 9 :PLAIN)`.

`with-focused-view` [*Macro*]

Syntax `with-focused-view` *view* {*form*}*

Description The `with-focused-view` macro executes *forms* with the current GrafPort set for drawing into *view.*

Arguments

view A view installed in a window, or `nil`. If `nil`, the current GrafPort is set to an invisible GrafPort.

form Zero or more forms to be executed with the current view set.

Example

Here is an example of using `with-focused-view` to paint a round-cornered rectangle within a window `window1`, using the Macintosh trap `#_PaintRoundRect`:

```
(rlet ((r :rect :top 20 :left 20 :bottom 80 :right 60))
  (with-focused-view window1
    (#_paintroundrect r 30 30)))
```

`find-window` [*Function*]

Syntax `find-window` *title* `&optional` *class*

Description The `find-window` function returns the frontmost window of the *class* class for which a prefix of the window's title is `string-equal` to *title*. If no window has *title* as its title, `nil` is returned.

Arguments

title	A string specifying the title of the window to search for.
class	A class used to filter the result. (The `&optional` in the syntax means that this argument is optional.)

Definition formats of CLOS generic functions

Like a function, a CLOS generic function specifies a procedure, but the generic function is specialized on the class of the instance to which it is applied. Thus a generic function may have more than one primary method. The provided methods of generic functions are listed in the "Syntax" section of the definition. Their syntax includes a procedure for matching the instance to a class.

`set-view-position` [*Generic function*]

Syntax `set-view-position (`*view* `simple-view)` *h* `&optional` *v*

Description The `set-view-position` generic function sets the position of the view in its container.

The positions are given in the container's coordinate system.

Arguments

view	A view or simple view, but not a window.
h	The horizontal coordinate of the new position, or the complete position (encoded as an integer) if *v* is nil or not supplied.
v	The vertical coordinate of the new position, or `nil` if the complete position is given by *h*.

Example

This code sets the position of `checkbox`, a checkbox dialog item, in the view `ed`.

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (set-view-position checkbox #@(20 20))
```

`1310740`

The generic function `initialize-instance`

The generic function `initialize-instance`, which is called by the function that creates an instance, also typically has a number of initialization arguments, which specify properties of the object instance and their initial values. These are documented among the arguments.

(Note that the function you call to create an instance is `make-instance`; `make-instance` calls `initialize-instance`.)

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*dialog-item* `dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `dialog-item` initializes a dialog item.

Arguments

dialog-item A dialog item.

initargs A list of keywords and default values used to initialize a dialog item. The *initargs* keywords for all dialog items are:

`:view-size`
The size of the dialog item.

`:view-position`
The position in the dialog box where the item will be placed, in the coordinate system of its container.

Argument list punctuation

Macintosh Common Lisp follows the notational conventions of Common Lisp. Argument lists use punctuation, such as parentheses, braces, and brackets, in special ways:

- Brackets `[]` indicate that anything they enclose is optional. This means that anything within them may appear once or not at all.
- Braces `{}` followed by an asterisk * mean that whatever they enclose may appear any number of times or not at all; everything within the braces is interpreted as a group.

- Braces`{}` followed by a plus sign + mean that whatever they enclose may appear multiple times but must appear at least once.
- A vertical bar `|` inside braces or brackets separates mutually exclusive choices. The group may be composed of a set from one side of the bar or from the other.
- Double brackets [[ ]] indicate that any number of the enclosed alternatives may appear, and in any order, but that each alternative may be used at most once unless followed by an asterisk.
- A downward arrow ↓ precedes a syntactic variable that will be subsequently defined.

Lisp syntax

Macintosh Common Lisp follows the syntactic conventions of Common Lisp; the complete Common Lisp syntax is described in Chapter 22 of the second edition of *Common Lisp: The Language.*

The following are some general characteristics of Lisp syntax:

- An open parenthesis (also called left parenthesis) begins a list of items.
- A close parenthesis (also called right parenthesis) ends a list of items.

  Nested lists are enclosed in nested parentheses:

  ```
  (like (these))
  ```
- A single quote (also called acute accent or apostrophe) followed by an expression *form* is an abbreviation for `(quote` *form*`)`.

  The expression `'foo` means `(quote foo)` and the expression `'(cons 'a 'b)` means `(quote (cons (quote a) (quote b)))`.
- A semicolon signals a comment. The semicolon and all characters following it up to the end of the line are ignored. A newline signals the end of the comment:

  ```
  (Here is Lisp code) ;Here is a comment,
                      ;which continues here.
    (and here is Lisp code again)
  ```
- Quotation marks, also called double quotes, surround character strings:

  ```
  "like this"
  ```
- A backslash `\`, the escape character, causes the next character to be treated as a letter rather than syntactically. For example, `\{` indicates the character for a left brace.
- Vertical bars in pairs `||` surround the name of a symbol with many special characters. Surrounding some characters with vertical bars is roughly equivalent to putting a backslash before each of the characters.

- A number sign `#`, also called a hash mark, signals the beginning of a complicated syntactic structure. The next character designates the syntactic structure to follow. For example, `#b1001` means `1001` in binary notation; `#(foo bar baz)` denotes a vector of three elements, `foo`, `bar`, and `baz`; `#\A` denotes the character object `A`; `#P"foo:bah"` indicates the pathname `"foo:bah"`; and `#'`*function* means `(function` *function*`)`.
- A grave accent ` (also called a backquote) is used together with commas to describe templates. The backquote syntax represents a program that will construct a data structure; commas are used within backquote syntax.
- A colon is used to indicate the package of a symbol. For instance, `lisp:dialog-item-size` denotes the symbol `dialog-item-size` in the package named `lisp`.

Chapter 1:

Editing in Macintosh Common Lisp

Contents

This chapter describes tools available for editing in MCL. It discusses Fred, the MCL text editor, as well as a number of additional tools which are helpful in editing text.

Fred combines the standard Macintosh multiple-window text editor with Emacs, the fully programmable editor that is a feature of most Lisp implementations. “Fred” is an acronym for “Fred Resembles Emacs Deliberately.”

If you are familiar with other Macintosh editors, you can begin editing in Macintosh Common Lisp immediately. However, Fred is much more powerful than most Macintosh editors. This chapter describes basic Fred concepts and keyboard editing shortcuts.

Since Fred is written in Macintosh Common Lisp, it is completely programmable. If you wish to change or extend it, you should read Chapter 14: Programming the Editor.

The MCL editor

Fred combines a standard Macintosh editor with Emacs, the fully programmable editor, optimized for Lisp programming, that is a feature of most Lisp implementations.

If you are familiar with other Macintosh editors, you can begin editing in Macintosh Common Lisp immediately. However, Fred has many more, and more powerful, features than the general run of Macintosh editors, and it has special features for programming Lisp.

- Fred includes many specialized Lisp manipulation commands. For example, you can select complete or partial symbolic expressions, move from level to level of a symbolic expression, reindent them, get their documentation and argument list, and inspect them, all with simple keyboard commands.

  Lisp expressions can be executed from Fred windows by pressing Enter (that is, the Enter key on the numeric keypad, not the Return key) with the cursor position at either end of a top-level expression. You can also highlight an expression and press Command-E.

  Placing the insertion point after a close parenthesis, or before an open parenthesis, causes the matching parenthesis to blink. For example, placing the insertion point after a close parenthesis causes the matching open parenthesis to blink. This feature is very helpful in balancing parentheses.

  Double-clicking after a close parenthesis, or before an open parenthesis, highlights to the matching parenthesis. For example, double-clicking before an open parenthesis highlights forward to the matching close parenthesis. This is another quick way to check the balance of your parentheses.

  Pressing Tab after a Return indents the new line appropriately.

  Pressing Control-Meta-Q reindents the current expression in a readable way.

  Pressing Meta–close parenthesis moves the cursor into position for typing the next expression.

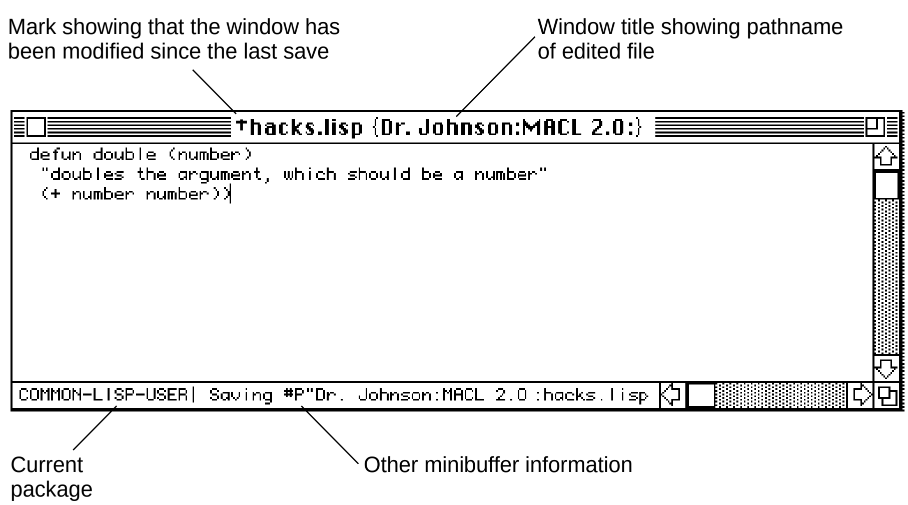
  Other Fred commands get information on the argument list of a function or its documentation, inspect it, and edit its source file. Most of these are available both on the “Tools” menu and as keyboard commands.
- Fred has online documentation of its own commands. Choose Fred Commands from the “Tools” menu to see a window of all Fred commands.
- Fred supports right-to-left as well as left-to-right editing. For information on using this feature, see the `:line-right-p` initialization arguments of the `fred-window` and `fred-dialog-item` classes.

- Since Fred is written in Macintosh Common Lisp, it is completely programmable. For example, the file `escape-key.lisp` in the MCL Examples folder binds the Macintosh Escape key to Meta.

The editing window

Figure 1-1 describes the parts of an editor window. At the top, in the title bar, is the pathname of the file contained in the window. The main body of the window contains the text of the file which is being edited. At the bottom of the window, the minibuffer displays the name of the window's package and other information.

■ **Figure 1-1** A Fred window



Working with the editor

This section gives general information on using the editor.

Creating new windows and opening files

To create a new file, press Command-N or choose “New” from the “File” menu. To open an already existing file, press Command-O or choose “Open” from the “File” menu.

Adding text to a file

Fred works with a mouse and the keyboard, just like other Macintosh text editors such as BBEdit. However, it understands Lisp and Lisp formatting better than those text editors. Specific editing instructions are given in the sections “Editing in Emacs style” on page 42 and “Fred commands” on page 43.

Saving text to files

To save the contents of a window, you can use the Command-S command or choose “Save” from the “File” menu. To save the contents under another name, choose the “Save As…” command from the “File” menu.

A small cross to the left of the filename in the title bar of a Fred window indicates that the contents of the window have been altered since the window was last saved. (See Figure 1-1.)

The “Windows” menu also displays a small cross to the left of the name of any window whose contents have been modified and not saved.

- *Note:* Fred stores files as text files, so they can be edited with other text editors. However, if you use another editor on a Fred file containing multiple fonts, the Fred font information will be corrupted.

Multiple Panes

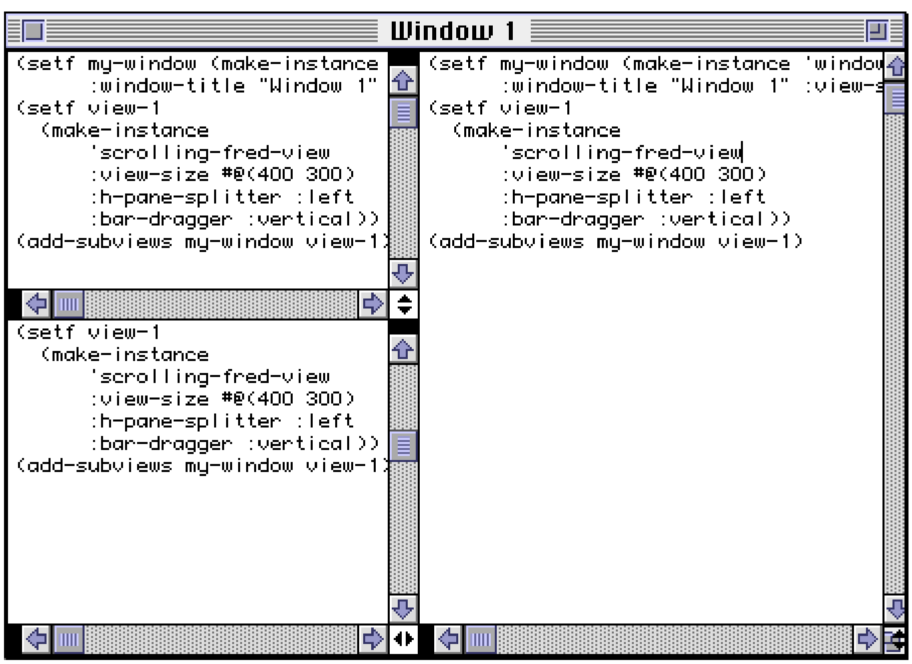
Fred windows can be split into multiple panes. Each pane can show a different portion of the text file being edited in the window.

The scrollbars in a Fred window have “pane-splitters” next to them. The pane splitter is a small black box. To create a new pane, click and drag on one of the pane splitters. A single vertical line appears in the window above the cursor. Drag it into the window while holding the mouse button down. When the window roughly in half, release the mouse button. The window now contains two individually scrollable panes. You can also double-click on the pane-splitter to create two panes of the same size.

When you have multiple-panes, the scroll-bar separating the panes will be abutted by a control containing two black triangles. This control is used to resize the panes. Click and drag the control to make the panes the size that you want. If you make a pane so small that it would be impractical to use it, the pane is removed. You can also remove a pane by double-clicking on the pane resizing control.

Each of the panes in a Fred window is provides a different view into a single file. To view more than one file using Fred, open each file into a different Fred window.

■ **Figure 1-2** A Fred window with multiple panes



The minibuffer

Each Fred window contains a minibuffer for conveying current information to the user. The minibuffer is at the bottom of the window, to the left of the horizontal scroll bar. (See Figures 2-1 and 2-2.) Information displayed in the minibuffer includes the package information for the window. This is the name of the window's package, if the window has one, or the value of the variable `*package*`.

Other information displayed in the minibuffer depends on what Macintosh Common Lisp is doing. Many system commands cause information to appear in the minibuffer.

In addition, you can set the text of the minibuffer yourself, as described in Chapter 14: Programming the Editor.

The minibuffer is actually a separate pane in the window, and so it can be resized. The up-and-down control in the horizontal (bottom) scrollbar allows you to reshape the window's minibuffer, to have more space to view messages there. This is particularly useful if you normally have `*arglist-on-space*` set to true, since it allows you to view long argument lists.

The kill ring and the Macintosh Clipboard

Macintosh Common Lisp supports both the standard Macintosh Clipboard and an Emacs-style **kill ring.**

Only the traditional Macintosh commands Cut and Copy move text to the Clipboard. Only the Macintosh command Paste moves text from the Clipboard. The Clipboard contains only one edit at a time.

In contrast, the Fred kill ring is a circular list that stores and retrieves multiple pieces of text. Fred's kill-ring mechanism guarantees that important text is not permanently lost through accidental deletion. It is a far more powerful mechanism than the Clipboard.

Any command that deletes or copies text moves the text to the kill ring. The Macintosh commands Cut, Copy, and Clear, as well as various Fred commands, add text to the kill ring. In addition, any text deleted by a side effect (that is, by typing or pasting when text in the window is selected) is also moved to the kill ring. Successive deletions with no intervening commands are concatenated into a single string in the kill ring. Only white space and single characters deleted by a side effect are not copied to the kill ring.

The kill ring is stored as a circular list in the variable `*killed-strings*`. You can retrieve any item from this list using the Fred command keystrokes Control-Y and Meta-Y, described among the Insertion commands in this chapter.

Fred commands that delete text do not place the text in the Clipboard, and Fred text retrieval commands do not retrieve text from the Clipboard. When you are cutting and pasting between Macintosh Common Lisp and another Macintosh application, you should use the Clipboard editing commands—Command-X, Command-C, and Command-V—rather than the Fred commands.

Multiple fonts

Fred has a standard Macintosh multiple-font capability. Runs of characters may be in different fonts, and the insertion font can be set and changed.

Fred window fonts can be set programmatically, as described in Chapter 14: Programming the Editor.They can also be set through commands on the “Edit” menu.

Font information is retained during cut, copy, and paste operations. You can disable this feature by setting the variable `*paste-with-styles*` to `nil`.

◆ *Note:* If you use an editor other than Fred on a Fred file containing multiple fonts, the Fred font information is corrupted.

Packages

Any Fred window can have an associated package. Expressions read from the window are read in the window’s package. If the window doesn’t have a package, then the value of the variable `*package*` is used.

A new, empty Fred window has no associated package.

The package may be set in three ways: through a mode line at the start of the text in the window, through an `in-package` statement, and through the generic function `set-window-package`. These three methods are not interchangeable. The circumstances under which each method can be used are described in this section.

Mode lines

To give a new, empty Fred window a package, you can add a prototype mode line by giving the Fred command Control-Meta-M. Then edit it to suit and use the Fred command Control-Meta-Shift-M to reparse the mode line and set the window package.

If present, the mode line must be the first nonempty line in the window's contents. It begins with one or more semicolons, followed by `-*-` (and often by `Mode: LISP` and a semicolon), followed by the package declaration.

For example, the following mode line causes expressions in the window to be read in the CCL package:

```
;-*- Mode: Lisp; Package: CCL -*-
```

Here are possible package specifications and the forms to which they are equivalent.

- `Package: FOO` is equivalent to `(in-package "FOO")`.
- `Package: (FOO)` is equivalent to `(make-package "FOO")`.
- `Package: (FOO (bar baz))` is equivalent to `(make-package "FOO" :USE '("BAR" "BAZ")))`.
- `Package: (FOO &rest` *x*`)` is equivalent to `(apply #'make-package "FOO"` *x*`)`.

If the package specified in the mode line exists, the window's package is set to that package. If it does not, the minibuffer indicates a new package:

```
(New package FOO)
```

The first time the package is needed to read an expression in the buffer, the package is created from the mode line specification, and the window's package is set to the created package.

An `in-package` expression

If there is no mode line, Fred looks for an `in-package` form at the beginning of the file. This form must be either the first form in the file or the second form when the first form defines a package with `defpackage`.

If there is an `in-package` form but the package does not exist, the window's package is set to `nil` and expressions read from the contents of the window are read in the package that is the value of `*package*`. If the package is being created with `defpackage`, you must make sure that the value of `*package*` either is or uses the package `"COMMON-LISP"`.

Once the package exists, use the Fred command Control-Meta-Shift-M to parse the mode line and set the window package.

◆ *Note:* The search for the `in-package` form ignores the read-time conditionals `#+` and `#-`.

A `set-window-package` expression

If you don't use either of the above methods, you can use the generic function `set-window-package`. The method for Fred windows takes two arguments, a Fred window and a package or a symbol that names a package.

Finding a window's package

You can find the package associated with a Fred window by calling the generic function `window-package`, with the window as the argument, or by looking in the minibuffer.

Fred parameters

The parameters in Table 1-1 can be used to control some of the behavior of Fred.

■ **Table 1-1** Fred parameters

Variable	Purpose
`*arglist-on-space*`	Displays in the minibuffer the argument list of a function when a user types a space following an open parenthesis and function name. (Does not parse; displays argument list of *any* symbol name that follows an open parenthesis.) *Default is true;* displays argument list for functions. If `nil`, does not display.
`*clear-mini-buffer*`	Specifies whether to clear the minibuffer after each Fred command. If you operate with `*arglist-on-space*` true, you may wish to set this to `nil` so that argument lists persist long enough to use. *Default is true;* text is cleared from minibuffer after any Fred command is run. If `nil`, text is cleared from minibuffer only when being replaced by other text.

`*control-key-mapping*`	Allows the command or command-shift key to be used as a Control key. This option is most useful for Macintosh keyboards with no Control key, determines which key combination specifies MCL Control key. The variable should have one of the following values:
	`nil`
	give no special meaning to command or command-shift.
	`:command-shift`
	command-shift maps to control and command is command.
	`:command`
	command maps to control and command-shift maps to command.
	Default is `nil`.

`*fred-default-font-spec*`	Specifies which font is used when new Fred windows are opened. The initial value is `("Monaco" 9 :PLAIN)`.
`*mini-buffer-font-spec*`	Specifies the font used in minibuffers. The default value is `("Monaco" 9)`. Note that the size of minibuffers does not increase even when a large point size is used. .

(continued)

■ **Table 1-1** Fred parameters (continued)

Variable	Purpose
`*next-screen-context-lines*`	This variable must be either an integer or a floating-point number. When it is an integer, it determines the number of context lines to retain when Fred scrolls to the previous or next screen. (Context lines are the lines from the previous screen that are retained on the new screen.) This value is used by various commands that scroll Fred windows. The default value is 2. When this variable is a floating-point number, it represents the percentage of context lines to retain. The value must be between 0.0 and 100.0.
`*paste-with-styles*`	Affects all commands that cause text to be pasted into a window. *Default is true;* style information is retained when text is copied and pasted. If `nil`, style information is discarded.
`*save-fred-window-positions*`	Affects whether window size, position, and current selection of Fred windows are retained when files are saved and later reopened. *Default is true;* information is retained. If `nil`, information is discarded.
`*save-position-on-window-close*`	Determines when the editor saves information about the size, position, beginning line, cursor position, and selection of the Fred window. *Default is* `nil`; when `*save-fred-window-positions*` is true, information is saved in the file's resource fork when the file is saved. If true, information is saved whenever the window is closed.

Normalizing `*next-screen-context-lines*`

The `next-screen-context-lines` function is used to normalize the Fred parameter `*next-screen-context-lines*` for a particular screen height.

`next-screen-context-lines` [*Function*]

Syntax `next-screen-context-lines` *screen-height*

Description The `next-screen-context-lines` function returns the number of lines of context to leave when scrolling a window.

Argument *screen-height* The window height in text lines.

Example

This function could be defined as follows.

```
? (defun next-screen-context-lines (screen-height)
    (let ((context *next-screen-context-lines*))
    (if (floatp context)
      (round (* context screen-height))
      (if (and (fixnump context)
               (< 0 context screen-height))
        context
        0))))
```

Editing in Macintosh style

Fred supports the standard set of Macintosh editing features and conforms to Macintosh standards. The basic Macintosh editing commands are available on the “Edit” menu, and their keyboard equivalents are supported.

You can cut, copy, and paste text between different windows (including the Listener) using Macintosh commands.

You can use almost any combination of MCL editing commands and Macintosh commands. You do not have to worry about how you combine them.

Editing in Emacs style

Fred supports a full suite of keyboard commands for manipulating text. Fred commands have been defined with care to conform to Emacs conventions. The exceptions are primarily due to the Macintosh standards and keyboard limitations.

The Control and Meta modifier keys

Emacs relies on two modifier keys to indicate command keystrokes. In Emacs, these modifiers are called Control and Meta. In Macintosh Common Lisp, various keystrokes may be used to invoke Control and Meta sequences.

- The Emacs Control modifier is accessible through the Macintosh Control key or through the Command key (on Macintosh keyboards that don't have a Control key). In all MCL documentation, whichever key you are using to indicate Control is referred to as the Control key. See the description of the variable `*control-key-mapping*` in the preceding table for instructions on using the Command key to indicate control.

  To issue a Control command, hold down the Control key while you press the letter of the command. For example, to enter Control-X, hold down the Control key and press X. To enter Control-X Control-S (the Emacs Save command), hold down the Control key and press X, then continue to hold down the Control key and press S. To enter Control-X H (the Emacs Select Entire Buffer command), hold down the Control key and press X, then release the Control key and press H.

- The Emacs Meta modifier is accessed through the Macintosh Option key.

  To issue a Meta command, hold down the Meta key while you press the letter of the command. For example, to enter Meta-X, hold down the Meta key and press X. This differs from some other implementations of Emacs, in which you press and release the Meta key before pressing the command letter.

  If you would prefer to use the Escape key as a Meta key, load the file `escape-key.lisp` in the Examples folder. To issue a meta command, press and release the escape key before you press the command letter. The Option key remains a Meta key and works as it did before.

To insert a Macintosh Option character into Macintosh Common Lisp, quote it: press Control-Q, then the character. For instance, you can insert the bullet sign, normally the Option-8 keystroke, by pressing Control-Q, then Option-8.

Control-Q works only on the next character typed; if you want to type a second Option character, press Control-Q again.

Disabling dead keys

The Macintosh keyboard supports **dead keys.** These are certain Option keystrokes used to prefix other keystrokes. The initial keystroke does not generate a character, but the second keystroke does. For example, no character appears when you press Option-N on a Macintosh English-language keyboard, but if you press A subsequently, you generate the character ã.

The dead key mechanism can interfere with the use of the Option key as the Meta key modifier. You can get around this in one of two ways:

- You can install a second keyboard layout that does not support dead keys. A number of freeware and shareware keyboard layouts are available for this purpose. You can also make your own keyboard layout by copying and editing the 'KCHR' resource. This resource type is documented in *Inside Macintosh*.

  If you install a keyboard layout that does not support dead keys, you can insert a dead-key keystroke in Macintosh Common Lisp by quoting it. For example, you can generate the character ã by pressing Control-Q Control-N A.
- You can use the Escape key as a Meta key, as described in the previous section. If you do this regularly, load `escape-key.lisp` as part of your `init` file.

Fred commands

The following Fred commands are defined in the initial MCL environment. Files in the Examples folder include additional Fred commands, and you can also write your own (as described in "Defining Fred commands" on page 516). Many commands are case insensitive; that is, you can press either Control-D or Control-Shift-D.

On the Apple Extended Keyboard, MCL editing uses the six named keys—Help, Forward Delete, Home, End, Page Up, and Page Down—in addition to the commands listed here.

Macintosh Common Lisp also uses the mouse for editing, both in the standard Macintosh way and in a few extended commands. For example, Macintosh Common Lisp recognizes up to a quadruple mouse click; it also recognizes mouse clicks in combination with Control and Meta keys. These commands are documented below.

The term **current expression,** used in the following documentation, denotes the text currently selected, if any. If no text is selected and the insertion point is next to a parenthesis, the current expression is between that parenthesis and the matching parenthesis—for example, between a close parenthesis and the matching open parenthesis, or between an open parenthesis and the matching close parenthesis. If no text is selected and the insertion point is inside a symbol, the symbol is the current expression. In other cases, there is no current expression.

Help, documentation, and inspection functions

The keystrokes and functions in Table 1-2 give information about Macintosh Common Lisp and its components.

■ **Table 1-2** Fred commands for help, documentation, and inspection

Keystroke	Function invoked	Purpose
Control-?	`ed-help`	Brings up the Fred Commands window. This window contains a list of all Fred keyboard commands available in the global command table. The list is regenerated each time the window is created. The Fred Commands window may be searched, saved, and printed.
Control-=	`ed-what-cursor-position`	Prints information about the current editor window to `*standard-output*`.
Meta-period	`ed-edit-definition`	Attempts to bring up the source code definition for the symbol surrounding the insertion point. If the symbol is defined from more than one source file, the user is given a choice of definitions. If the symbol is defined as a slot in a `defclass`, Meta-period finds the *approximate* location of the symbol. Search backward with Control-R to find the location at which the symbol is defined. This function works for most forms that are defined with `*record-source-file*` set to `t`.

(continued)

■ **Table 1-2** Fred commands for help, documentation, and inspection (continued)

Keystroke	Function invoked	Purpose
Command-Meta-click	`edit-definition`	Attempts to bring up the source code definition for the symbol on which the mouse clicks; works like `ed-edit-definition`.
Control-X Control-A	`ed-arglist`	Prints the argument list of the function bound to the symbol surrounding the insertion point. Argument list is displayed in the minibuffer if the value of `*mini-buffer-help-output*` is `t`; otherwise, it is displayed in the `*standard-output*` stream. The `ed-arglist` function works for built-in functions and macros, and for most functions and macros defined with `*save-local-symbols*` or `*fasl-save-local-symbols*` set to `t`.
Control-X Control-D	`ed-get-documentation`	Opens a dialog box displaying the symbol surrounding the insertion point and the documentation string of the function bound to that symbol. If no documentation string is available, displays "No documentation available." This function works for built-in functions and macros and for most forms defined with `*save-doc-strings*` set to true.
Control-X Control-I	`ed-inspect-current-sexp`	Inspects the current symbolic expression.

Movement

During editing, use the functions and keystrokes in Table 1-3 to move the insertion point. Most of these movement commands can be modified by the Shift key to establish or extend a selection; see Table 1-4.

■ **Table 1-3** Fred commands for movement

Keystroke	Function invoked	Purpose
Control-B, ←	`ed-backward-char`	Moves the insertion point back one character..
Control-F, →	`ed-forward-char`	Moves the insertion point forward one character.
Meta-B, Meta-←	`ed-backward-word`	Moves the insertion point back one word.
Meta-F, Meta-→	`ed-forward-word`	Moves the insertion point forward one word.
Control-Meta-B, Control-←	`ed-backward-sexp`	Moves the insertion point back one s-expression.
Control-Meta-F, Control-→	`ed-forward-sexp`	Moves the insertion point forward one s-expression.
Control-A	`ed-beginning-of-line`	Moves the insertion point to the beginning of the line.
Control-E	`ed-end-of-line`	Moves the insertion point to the end of the line.
Control-Meta-A	`ed-start-top-level-sexp`	Moves the insertion point to the beginning of the current top-level s-expression. Top-level expressions are signaled by an open parenthesis flush with the left margin.
Control-Meta-E	`ed-end-top-level-sexp`	Moves the insertion point to the end of the current top-level s-expression. Top-level expressions are recognized by having an open parenthesis flush with the left margin.
Control-P	`ed-previous-line`	Moves the insertion point up one line.
Control-N	`ed-next-line`	Moves the insertion point down one line.
Meta-V	`ed-previous-screen`	Scrolls upward through the text by a windowful and moves the insertion point to the upper-left corner of the window. The number of lines to be retained from the previous screen after scrolling is determined by `*next-screen-context-lines*`.

(continued)

■ **Table 1-3** Fred commands for movement (continued)

Keystroke	Function invokedPurpose	
Control-V	`ed-next-screen`	Scrolls downward through the text by a windowful and moves the insertion point to the upper-left corner of the window. The number of lines to be retained is determined by `*next-screen-context-lines*`.
Meta-<	`ed-beginning-of-buffer`	Moves the insertion point to the beginning of the buffer.
Meta->	`ed-end-of-buffer`	Moves the insertion point to the end of the buffer.
Meta-)	`ed-move-over-close-and-reindent`	Moves the insertion point over the next close parenthesis and into position for typing the next Lisp expression.
Control-Tab	`ed-indent-differently`	Reindents the line containing the insertion point to an alternate indentation.
Control-Meta-)	`ed-fwd-up-list`	Moves the insertion point past the end of the current s-expression. Used again, it moves the insertion point up one level of the expression, that is, past the close parenthesis at the next higher level of the expression.;
Control-Meta-(	`ed-bwd-up-list`	Moves the insertion point to before the beginning of the current s-expression. Used again, it moves the insertion point up one level of the expression, that is, to before the open parenthesis at the next higher level of the expression.;
Control-Meta-N, Control-Meta-↓	`ed-next-list`	Moves the insertion point in window past the end parenthesis of the next s-expression at the same level.;
Control-Meta-P, Control-Meta-↑	`ed-previous-list`	Moves the insertion point to before the opening parenthesis of the previous s-expression at the same level.;
Meta-M	`ed-back-to-indentation`	Moves the insertion point to the first non-white-space character in its current line.

Selection

The keystrokes in Table 1-4 are used to select text. You can modify most motion commands with the Shift key to select the region between the original insertion point and the new insertion point.

In addition, you can use the mouse to select text, either through multiple-clicks, or by clicking and dragging.

- Two clicks selects a word or parenthesized expression.
- Three clicks selects a line.
- Four clicks selects the entire window contents.

■ **Table 1-4** Fred commands for selection

Keystroke	Function invoked	Purpose
Shift-←	`ed-backward-select-char`	Selects one character backward from the insertion point and moves the insertion point to the left of that character.
Shift-→	`ed-forward-select-char`	Selects one character forward from the insertion point and moves the insertion point to the right of that character.
Meta-Shift-←	`ed-backward-select-word`	Selects one word backward from the insertion point and moves the insertion point to the left of that word. If the insertion point is in the middle of a word, selects the word.
Meta-Shift-→	`ed-forward-select-word`	Selects one word forward from the insertion point and moves the insertion point to the right of that word. If the insertion point is in the middle of a word, selects the word.
Control-Shift-←	`ed-backward-select-sexp`	Selects one symbolic expression backward from the insertion point and moves the insertion point to the left of that symbolic expression. If the insertion point is in the middle of a word, selects to the beginning of the word.

(continued)

■ **Table 1-4** Fred commands for selection (continued)

Keystroke	Function invoked	Purpose
Control-Shift-→	`ed-forward-select-sexp`	Selects one symbolic expression forward from the insertion point and moves the insertion point to the right of that symbolic expression. If the insertion point is in the middle of a word, selects to the end of the word.
Control-Shift-A	`ed-select-beginning-of-line`	Selects to the beginning of the line and moves the insertion point to the beginning of the selection.
Control-Shift-E	`ed-select-end-of-line`	Selects to the end of the line and moves the insertion point to the end of the selection.
Control-Meta-H	`ed-select-top-level-sexp`	Selects the current top-level s-expression. Top-level expressions are signaled by an open parenthesis flush with the left margin.
Control–Meta–Space bar	`ed-select-current-sexp`	Selects the current s-expression.
Control-X H	`select-all`	Selects the entire buffer and scrolls to the beginning of the buffer.
Shift-↑, Control-Shift-P	`ed-select-previous-line`	Selects to the same point of the previous line and moves the insertion point to before the beginning of the selection. If it is not possible to move the insertion point to the same column in the previous line, it moves the insertion point to the end of the previous line.
Shift-↓, Control-Shift-N	`ed-select-next-line`	Selects to the same point of the next line and moves the insertion point past the end of the selection. If it is not possible to move the insertion point to the same column in the next line, Macintosh Common Lisp moves the insertion point to the end of the next line.

(continued)

■ **Table 1-4** Fred commands for selection (continued)

Keystroke	Function invoked	Purpose
Shift–Page Up, Meta-Shift-V	`ed-select-previous-screen`	Selects from the insertion point to the corresponding line and column in the previous screen, or, if this is not possible, to the end of the corresponding line on the previous screen. It moves the insertion point to before the beginning of the selection.
Shift–Page Down, Control-Shift-V	`ed-select-next-screen`	Selects from the insertion point to the corresponding line and column in the next screen, or, if this is not possible, to the end of the corresponding line on the next screen. It moves the insertion point past the end of the selection.
Control-Meta-Shift-P, Control-Meta-Shift-↑	`ed-select-previous-list`	Selects to the beginning of the previous list at the same level and moves the insertion point to before the open parenthesis of that list.
Control-Meta-Shift-N, Control-Meta-Shift-↓	`ed-select-next-list`	Selects to the end of the next list at the same level and moves the insertion point past the close parenthesis of that list.
Control-X Control-X	`ed-exchange-point-and-mark`	Exchanges the positions of the insertion point and the top mark. With an argument, the range between the two is selected. For example, Control-X Control-X exchanges the position of the point and the mark; Control-1 Control-X Control-X exchanges them and selects the range between.

Insertion

The keystrokes in Table 1-5 are used to insert text and space.

■ **Table 1-5** Fred commands for insertion

Keystroke	Function invoked	Purpose
Control-O	`ed-open-line`	Inserts a new line without moving the insertion point.
Control-Meta-O	`ed-split-line`	Splits the line in which the insertion point is located, indenting so that the column in which the characters are located does not change.
Tab	`ed-indent-for-lisp`	Reindents the current line. (To insert a tab, press Control-Q followed by Tab.) If there is a selection, the entire selection is reindented.
Control-Meta-Q	`ed-indent-sexp`	Reindents the current expression.
Control-Return	`ed-newline-and-indent`	Inserts Return followed by Tab.
Control-Y	`ed-yank`	Inserts (yanks) the current kill ring string into the buffer at the insertion point. If text is selected, it is replaced with the inserted text. This command keystroke is often used after Cut or Copy (Control-W or Meta-W).
Meta-Y	`ed-yank-pop`	Performs a “rotating yank.” When Meta-Y is first pressed, the first item in the kill ring is inserted (yanked). If pressed immediately again, Meta-Y removes the old insertion, rotates the kill ring, and inserts the next item in the kill ring. Repeatedly pressing Meta-Y shows each item in the kill ring (you rotate through the kill ring and eventually return to the beginning). The kill ring remains rotated until you perform another kill.

(continued)

■ **Table 1-5** Fred commands for insertion (continued)

Keystroke	Function invoked	Purpose
`Control-Q`		Inserts the next keystroke quoted, allowing access to the Macintosh optional character set and other special characters. That is, for a single keystroke following the pressing of Control-Q, the Option key is not interpreted as a Meta keystroke. For example, you insert the bullet sign (normally the Option-8 keystroke) by pressing the Control-Q and Meta-8. Pressing only Meta-8 would cause Fred to look for a command. Control-Q can also be used to insert control characters such as tabs into buffers.
`Meta-"`	`ed-insert-double-quotes`	Inserts the characters " " and puts the insertion point between them.
`Meta-#`	`ed-insert-sharp-comment`	Inserts the characters `#||#` and puts the insertion point between the vertical bars.
`Meta-(`	`ed-insert-parens`	Inserts a set of parentheses and puts the insertion point between them.
`Meta-U`	`ed-upcase-word`	Converts the rest of the current word or each word in a selection to uppercase. For example, if the insertion point is between the `y` and the `u` of the word `giddyup`, pressing Meta-U produces `giddyUP`. Repeatedly typing Meta-U converts successive words to uppercase. Note that Option-U is a dead key on English-language keyboards; see “Disabling dead keys” on page 43.
`Meta-L`	`ed-downcase-word`	Converts the rest of the current word or each word in a selection to lowercase. For example, if the insertion point is between the `E` and the `M` of the word `EMACS`, pressing Meta-L produces `Emacs`. Repeatedly typing Meta-L converts successive words to lowercase.

(continued)

■ **Table 1-5** Fred commands for insertion (continued)

Keystroke	Function invoked	Purpose Deletion
Meta-C	`ed-capitalize-word`	Capitalizes the first letter of the rest of the current word or the first letter of each word in a selection. For example, if the insertion point is between the first and second `c` of the word `Hiccup`, typing Meta-C produces `HicCup`. Repeatedly typing Meta-C capitalizes successive words.
Control-T	`ed-transpose-chars`	Transposes the two characters surrounding the insertion point unless the insertion point is at the end of a line, in which case it transposes the two characters to the left of the insertion point. If there is a selection, the first character in the selection is transposed with the character before the selection.
Meta-T	`ed-transpose-words`	Transposes the two words surrounding the insertion point.
Control-Meta-T	`ed-transpose-sexps`	Transposes the two symbolic expressions surrounding the insertion point.
Control–Space bar	`ed-push/pop-mark-ring`	Pushes the position of a mark onto the mark ring. With an argument *n*, it moves to the *n*th mark position in the mark ring. If the mark ring is empty, the function signals an error.
Control-X Control-X	`ed-exchange-point-and-mark`	Exchanges the positions of the insertion point and the top mark. With an argument, the range between the two is selected. For example, Control-X Control-X exchanges the position of the point and the mark; Control-1 Control-X Control-X exchanges them and selects the range between.

Deletion

The keystrokes and functions in Table 1-6 are used to delete text and spaces.

◆ *Note:* The key in Delete, Meta-Delete, and Control-Meta-Delete is the Delete key, not the Forward Delete key on the Apple Extended Keyboard.

■ **Table 1-6** Fred commands for deletion

Keystroke	Function invoked	Purpose
Delete	`ed-rubout-char`	Deletes the character to the left of the insertion point.
Meta-Delete	`ed-rubout-word`	Deletes the word to the left of the insertion point. If the insertion point is inside a word, only the portion of the word to the left of the insertion point is deleted.
Control-Meta-Delete	`ed-kill-backward-sexp`	Deletes the expression to the left of the insertion point.
Control-D, Forward Delete (extended keyboard)	`ed-delete-char`	Deletes the character to the right of the insertion point. (This is the Forward Delete key on the Apple Extended Keyboard, not the Delete key over the Return key.)
Meta-D	`ed-delete-word`	Deletes the word to the right of the insertion point. If the insertion point is inside a word, only the portion of the word to the right of the insertion point is deleted.
Control-K	`ed-kill-line`	Deletes the remainder of the line containing the insertion point, adding it to the kill ring. If the insertion point is at the end of a line, the following carriage return is deleted.
Control-Meta-K	`ed-kill-forward-sexp`	Deletes the expression to the right of the insertion point, adding it to the kill ring.

(continued)

■ **Table 1-6** Fred commands for deletion (continued)

Keystroke	Function invoked	Purpose
Control-W	`ed-kill-region`	Deletes the current selection, adding it to the kill ring.
Meta-W	`ed-copy-region-as-kill`	Adds the current selection (or current expression) to the kill ring without deleting it from the buffer.
Control-X Control–Space bar	`ed-delete-forward-whitespace`	Deletes all white-space from the insertion point to the next non-white-space character.
Meta–Space bar	`ed-delete-whitespace`	Replaces all spaces and tabs surrounding the insertion point by a single space.
Meta-\	`ed-delete-horizontal-whitespace`	Deletes all white space characters to the left and right of the insertion point.
Control-Meta-;	`ed-kill-comment`	Kills only the comment in the line containing the insertion point. The insertion point may be located anywhere in the line.

Lisp operations

The functions and keystrokes in Table 1-7 perform Lisp operations on the current expression.

■ **Table 1-7** Fred commands for Lisp operations

Keystroke	Function invoked	Purpose
Enter	`ed-eval-or-compile-current-sexp`	Executes or compiles the current expression. This key is not the Return key (which inserts a carriage return and may cause an execution in the Listener) but the key marked Enter in the numeric keypad.
Control-X Control-C	`ed-eval-or-compile-top-level-sexp`	Executes or compiles the current selection or the current top-level Lisp expression, whichever is appropriate. The current top-level Lisp expression is determined heuristically by searching backward for an open parenthesis at the start of a line.
Control-X Control-E	`ed-eval-current-sexp`	Executes the current expression.
Control-M	`ed-macroexpand-1-current-sexp`	Macroexpands the current expression with `macroexpand—1`, repeatedly if necessary, until the expression is no longer a macro. The result of each call to `macroexpand-1` is printed in the Listener.
Control-X Control-M	`ed-macroexpand-current-sexp`	Macroexpands the current expression and pretty-prints the result into the Listener. The expansion is done as if by a call to `macroexpand`.
Control-Meta-Shift-M	`add-modeline`	Adds a mode line.
Control-X Control-R	`ed-read-current-sexp`	Reads the current expression and pretty-prints the result into the Listener. This command is useful for checking read-time bugs, especially for those expressions containing backquotes.

(continued)

■ **Table 1-7** Fred commands for Lisp operations (continued)

Keystroke	Function invoked	Purpose
Meta-;	`ed-indent-comment`	Inserts or aligns comments. If the line that contains the insertion point of window or item starts with one or more semicolons (which indicate comments in Lisp), aligns the line with the comment column (by default, column 40). If there is no comment on the line containing the insertion point, the function inserts a semicolon at the comment column, followed by a space, and moves the insertion point to the comment column +2.
Control-X ;	`ed-set-comment-column`	Sets the comment column to that of the current insertion point.
Control-Meta-;	`ed-kill-comment`	Kills only the comment in the line containing the insertion point. The insertion point may be located anywhere in the line.

Window and file operations

The functions and keystrokes in Table 1-8 are used to save and select text manipulated in windows.

■ **Table 1-8** Fred commands for window and file operations

Keystroke	Function invoked	Purpose
Control-X Control-S	`window-save`	Saves the contents of the active Fred window to its associated disk file. If no file is associated with the window, the user is requested to supply a filename.
Control-X Control-W	`window-save-as`	Saves the contents of the active Fred window to a file specified by the user.
Control-X Control-V	`edit-select-file`	Allows the user to select a text file and opens a Fred window for editing that file. .
Control-Meta-L	`ed-last-buffer`	Switches the positions of the first and second windows on the list of windows, so that the second window becomes the active window. Called again, it toggles their positions again. (It switches away from Apropos, Inspector Central, Search Files, and String Search, but not back.)

Undo commands

The Undo command undoes the effect of previous commands. Functions and keystrokes associated with Undo are listed in Table 1-9. Successive insertions or deletions, or multiple replacements via the Search dialog, are considered a single command.

Each window has its own Undo history list.

■ **Table 1-9** Fred commands for undoing commands

Keystroke	Function invoked	Purpose
Control-_	`ed-history-undo`	Undoes a previous Fred command.
Control-Meta-_	`ed-print-history`	Displays the Undo history list in the Listener.

Numeric arguments

The keystrokes in Table 1-10 multiply the effect of any command to which they can be applied. (They can always be applied to motion and selection commands.)

■ **Table 1-10** Fred commands for giving numeric arguments

Keystroke	Function invoked	Purpose
Control-U *n*	`ed-universal-argument`	The universal argument multiplies any Fred keystroke command *n* number of times. The argument *n* is optional. If a keystroke command is entered instead of a number, *n* is taken to be 4. For example, to move down four lines, you give the command Control-U Control-N. To move down three lines, you give the command Control-U 3 Control-N. (Entering a very large number may result in an error.)
Control-*n*, Meta-*n*, Control-Meta-*n*	`ed-numeric-argument`	Turns a digit *n* into a numeric argument for the subsequent command. For example, pressing Meta-5 Control-N moves the insertion point down five lines; pressing Meta-1 Meta-2 Control-N moves it down twelve lines. (Entering a very large number may result in an error.)

Incremental searching in Fred

Fred supports an Emacs-style incremental search. The incremental search is invoked through the keystrokes Control-S (incremental search forward) and Control-R (incremental search reverse).

The mechanism of incremental search is fairly complicated. However, this complexity is necessary to make the incremental search easy to perform (as well as powerful). If you have trouble following this description, experiment with incremental searching. You should get the hang of it easily.

Performing an incremental search

When you first press Control-S in a Fred window, Fred displays the prompt `i-search` in the window's minibuffer (or `i-search reverse` for Control-R). At this point, you can start typing the characters in your search string.

If you type `f`, Fred immediately searches for the next occurrence of `f` after the insertion point and selects it, scrolling through the text if necessary to make it visible.

If you then type `o`, Fred starts with the currently selected `f` and searches for `fo`. You can continue typing characters to add to the search string. With each addition, Fred immediately searches for the next occurrence of the string and selects the found text. The next occurrence may be a simple extension of the previously found text, or it may occur later in the buffer.

Making additional searches

Suppose you type `foo` and Fred finds the string in the buffer, but it is not the right one. You want a later occurrence of `foo`. Press Control-S a second time to search for the next occurrence of `foo`. You can continue pressing Control-S to search for subsequent occurrences of the string.

When a search fails, you hear a beep, and the `i-search` prompt changes to `Failing i-search.` A search may fail because there are no more occurrences of the string or because you add a character to the search string and that new string cannot be found in the buffer. In either case, pressing Control-S again at this point causes the search to begin again at the beginning of the buffer.

The behavior of Control-R is identical to that of Control-S except that the search proceeds backward from the insertion point to the beginning. When no further occurrences are found and you press Control-R again, the search begins anew from the end of the buffer.

Backing up with the Delete key

Sometimes you may want to change the search string. For example, you may mistakenly type `foot` instead of `fool`. Pressing the Delete key has the effect of undoing the last keystroke in the search string. This deletes the last character of the search string and, if necessary, resumes the search at the buffer location where the insertion point was when you typed the last character of the search string. Pressing Delete several times removes additional characters from the search string and "moves back" the search further.

You can use the Delete key to undo the effects of Control-S and Control-R in addition to the effects of adding characters to the search string. For example, suppose you type `foo` and then press Control-S twice. At this point, the insertion point will be located at the third occurrence of `foo` in the window (assuming there are three occurrences of `foo`). If you then press Delete, Fred reverses the effects of the last keystroke (Control-S), returning to the second occurrence of `foo`. Pressing Delete again undoes the first Control-S, and the insertion point moves to the first occurrence. Pressing Delete yet again undoes the letter `o`, and Fred shows you the first occurrence of `fo` in the window.

If the last keystroke added a block of characters to the search string, pressing Delete removes the entire block. (See Control-Q, Control-W, and Control-Y in "Special incremental search keystrokes" on page 64.)

Terminating an incremental search

There are a number of ways to terminate an incremental search:

- Clicking the mouse button performs the indicated action and terminates the incremental search.
- Pressing Escape terminates the incremental search, moving the insertion point to the end of the selection (on incremental search) or beginning of the selection (on incremental search reverse).
- Pressing Control-G terminates the search if there were no unfound characters and returns the insertion point to its location before the search began. If there were some unfound characters, these are deleted from the search string, and the search can continue.
- Choosing a Fred command causes the command to be executed and terminates the search.
- Choosing a menu command causes the command to be executed and terminates the search.

Doing another incremental search

Fred keeps track of the last string used in an incremental search. When you do another incremental search, this string appears in the minibuffer as the default search string. This feature makes it easy to search several windows for the same string.

When the default string appears, immediately press Control-S or Control-R to search for the string. If you type anything else before typing Control-S or Control-R, Fred deletes the default string and starts a search for the new string.

Special incremental search keystrokes

These keystrokes in Table 1-11 have special meanings in the context of an incremental search.

■ **Table 1-11** Fred commands for searching

Keystroke	Function invoked	Purpose
Control-S	`ed-i-search-forward`	Initiates a forward incremental search..
Control-R	`ed-i-search-reverse`	Initiates a reverse incremental search.
Delete		Deletes the last character typed and backs up the search.
Control-G		During a search that has found nothing, deletes all unfound characters from the search string. The search can then continue. During a successful search, ends the incremental search and returns the insertion point to its original position.
Control-Q		Gets a character and inserts it quoted into the search string. This is used to search for special characters, such as Control-S, Return, or Tab.
Control-W		Copies the word or selection following the insertion point into the search string.
Control-Y		Copies the line following the insertion point into the search string.
Control-S Control-Y		Appends the selection or rest of line to the search string.
Control-S Control-W		Appends the string from the insertion point to the end of the current word to the search string.
Control-S Control-Meta-W		Appends the string from the insertion point to the end of the current s-expression to the search string.
Control-S Meta-W		Ends the search.

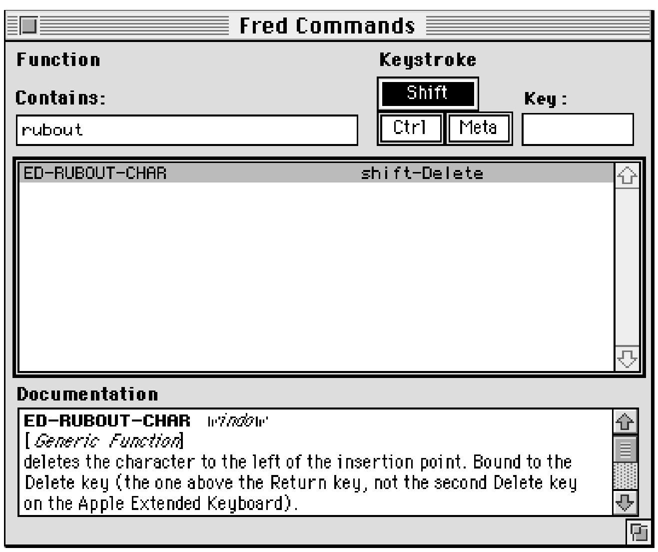
The Fred Commands tool

The Fred Commands tool is accessed through the “Fred Commands” command on the “Tools” menu. It lists all the Fred commands bound to keys. (These commands are those in the command table stored in the parameter `*comtab*`).

The following figure shows the Fred Commands dialog box. The Contains text edit field specifies a string contained by commands in the scrolling-list. The Keystroke buttons and Key box are live controls that specify a key sequence to show in the list.

The Keystroke button specifies a keystroke when you press the button with your mouse or press a corresponding key on your keyboard. To enter a character in the Key box, you must first click in the box, then type a character from your keyboard.

■ **Figure 1-3** The Fred Commands dialog box

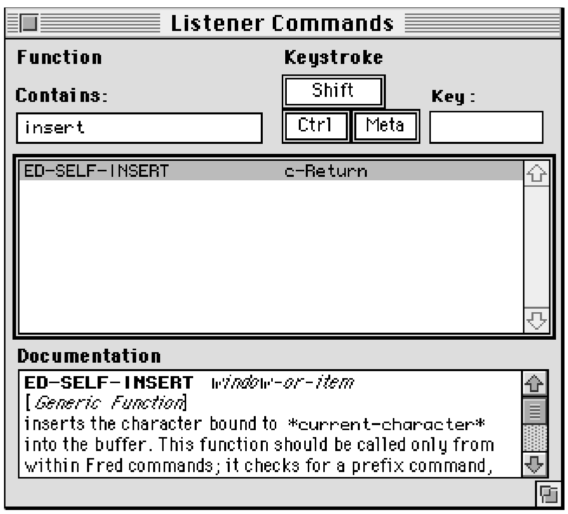


The Listener Commands tool

The editing commands available in the Listener are slightly different from those available in other Fred windows. The Listener Commands tool, accessed through the "Tools" menu, lists these differences. (The commands it shows are those stored in `*listener-comtab*`).

The operation of this tool is the same as that of the Fred Commands tool.

■ **Figure 1-4** The Listener Commands dialog box



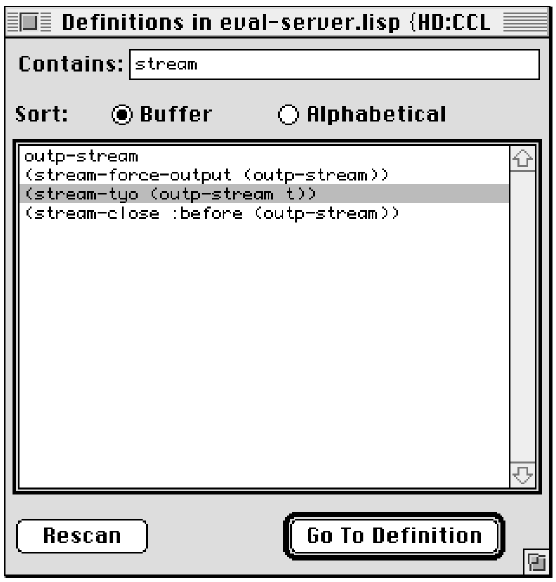
The List Definitions tool

The List Definitions tool is accessed through the "List Definitions" command on the "Tools" menu. It displays a modeless dialog box (see the following figure), that lists all the definitions in the top editor window. Double-clicking on a selection or pressing the Go To Definition button, scrolls the window to that definition. The Contains text edit field in the dialog box acts as a filter for selecting only those definitions containing a particular string.

This tool sorts definitions in the order that they appear in the buffer or alphabetically, depending on the setting of the Sort buttons. The buttons at the bottom of the dialog box rescan the buffer and find the highlighted definition.

When the active window is not an editor window, this command is dimmed.

■ **Figure 1-5** The List Definitions dialog box



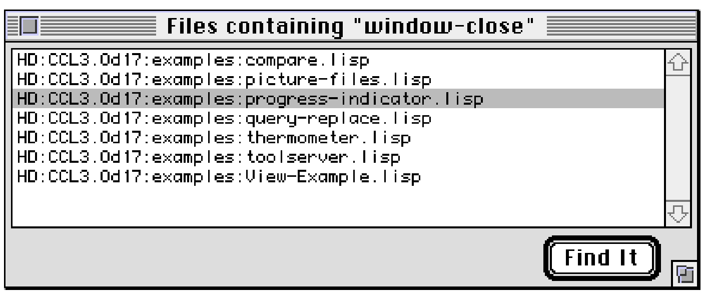
The Search Files tool

The Search Files tool is accessed through the “Search Files” command on the “Tools” menu. It searches a set of files for a given string. This tool displays the dialog box shown in the following figure. The In Pathname text edit field specifies the set of files for the search. The Search For text edit field specifies a string to locate in the files.

■ **Figure 1-6** The Search Files dialog box

If the Search Files tool finds a file containing the specified string, the tool displays the dialog box in the following figure. If you press the Find It button in that dialog box, this tool opens the file in a Fred window.

■ **Figure 1-7** Dialog box after a successful search



This tool accepts wildcard characters in the pathname specification. Macintosh Common Lisp supports Common Lisp extended wildcards, which are documented in Steele, pages 623–627, and has a wildcard specification system described in “Wildcards” on page 298 .

The Search Files tool spawns a process, so it is possible to have multiple searches running at the same time. Because it uses Boyer-Moore search the Boyer-Moore search algorithm, it is quite fast.

Chapter 2:

Points and Fonts

Contents

This chapter describes the MCL implementation of points and fonts. Points are used for drawing into views; font specifications and font codes describe fonts.

Some allied MCL functions give useful data about your Macintosh system. They are also described in this chapter.

You should read this if you are not already familiar with the MCL and Macintosh implementations of these concepts.

Points

Points are used throughout Macintosh Common Lisp to represent two-dimensional data. The most common use of points is in graphics operations that require you to specify a width and a height (for example, specifying the size of a window) or horizontal and vertical coordinates (for instance, specifying the position of an item in a dialog box).

How Macintosh Common Lisp encodes points

Points are graphics coordinates with an x component and a y component. To save space, Macintosh Common Lisp encodes the x and y components of a point into a single integer, known as the "encoded form." The low-order 16 bits hold the x coordinate and the high-order 16 bits hold the y coordinate. Both dimensions are signed.

Many Lisp functions that take a point as an argument can accept it as two coordinates (*h* and *v*) or as a single integer holding both coordinates. If a function takes more than one point, or has optional arguments, the points must all be passed in encoded form.

Points are always returned as a single encoded integer.

The reader macro `#@` converts the subsequent list of two integers into a point. This can be used for clarity in source code. For example, `#@(30 -100)` expands into `-6553570`, an integer that represents the point with a horizontal coordinate of 30 and a vertical coordinate of –100.

The integer that encodes the x and y coordinates of a point is automatically converted to a bignum if a fixnum cannot accommodate it. (For definitions of bignum and fixnum, see *Common Lisp: The Language*.)

Except in cases where efficiency is paramount and the range is guaranteed to be below 4096, you should always assume that graphics points may be bignums. Because of this, `eq` can't safely be used to compare points; you must use `eql` as follows:

```
? (eq #@(1800 7496) #@(1800 7496))
NIL
? (eql #@(1800 7496) #@(1800 7496))
T
```

MCL functions relating to points

The following functions relate to points.

point-string [*Function*]

Syntax `point-string` *point*

Description The `point-string` function returns a string representation of *point*.

Argument *point* A point.

Example

```
? (point-string 4194336)
"#@(32 64)"
? (view-position (front-window))
14417924
? (point-string (view-position (front-window)))
"#@(4 220)"
```

point-h [*Function*]

Syntax `point-h` *point*

Description The `point-h` function returns the horizontal coordinate of *point*.

Argument *point* A point.

Example

```
? (point-h 4194336)
32
```

point-v [*Function*]

Syntax `point-v` *point*

Description The `point-v` function returns the vertical coordinate of *point*.

Argument *point* A point.

Example

```
? (point-v 4194336)
64
```

---

point<= [*Function*]

Syntax `point<=` *point* `&rest` *other-points*

Description The `point<=` function checks to see whether *point* and *other-points* are ordered by nondecreasing size in both coordinates. If they are, or if there is only one point, the function returns `t`; otherwise, it returns `nil`.

Arguments

point	A point, expressed as an integer.
other-points	Zero or more other points, expressed as fixnums.

---

make-point [*Function*]

Syntax `make-point` *h* `&optional` *v*

Description The `make-point` function returns a point constructed from horizontal and vertical coordinates *h* and *v*.

Arguments

h	The horizontal coordinate of the point, or the complete point (encoded as an integer) if *v* is `nil` or not supplied.
v	The vertical coordinate of the point. If *v* is `nil` (the default), *h* is assumed to be an entire point in encoded form and is returned unchanged.

Examples

```
? (make-point 32 64)
4194336
? (make-point 32 nil)
32
? (make-point 32)
32
```

You can pass `make-point` the two coordinates of a point, or you can pass it a point as a single argument. In either case, it returns a point. This makes `make-point` very useful in processing optional argument sets.

```
? (make-point 40 50)
3276840
? (make-point 3276840)
3276840
? (point-string 3276840)
"#@(40 50) "
? (defun show-point
    (h &optional v)
    (point-string (make-point h v)))
show-point
? (show-point 32 32)
"#@(32 32)"
? (show-point 3276840)
"#@(40 50)"
```

add-points [*Function*]

Syntax `add-points` *point1 point2*

Description The `add-points` function returns a point that is the result of adding *point-1* and *point-2*.

Points cannot be added with the standard addition function because of possible overflow between the x and y components of the encoded form.

Arguments

point-1	A point.
point-2	A point.

Example

```
? (point-string (add-points #@(10 10) #@(50 100)))
"#@(60 110)"
```

subtract-points [*Function*]

Syntax `subtract-points` *point1 point2*

Description The `subtract-points` function returns a point that is the result of subtracting *point-2* from *point-1*.

Points cannot be subtracted with the standard subtraction function because of possible overflow between the x and y components of the encoded form.

Arguments

point-1	A point.
point-2	A point.

Example

```
? (point-string (subtract-points #@(10 10)
                                 #@(3 4)))
"#@(7 6)"
```

Fonts

There are two ways of representing fonts in Macintosh Common Lisp, font specifications and font codes.

A font specification (font spec) is an atom or list of atoms specifying one or more of the following: the font name, font size, font styles, font color and transfer mode. They are more humanly readable than font codes. They can be translated into font codes through the function `font-codes`.

Font codes represent font information in a way that accesses the Macintosh Font Manager directly. Since they don't need to be interpreted, they are significantly faster than font specifications. They can be translated into font specifications explicitly through the function `font-spec`.

The manner in which font information is encoded in font-codes is described fully in *Inside Macintosh*.

Implementation of font specifications

The font name should be a string. It should correspond to a font available in the System file. You can find out which fonts are available by examining the `*font-list*` variable, described in the section "System data" on page 87. Font names are not case sensitive.

The font size should be an integer, which is always in the range from 1 to 127. Because of an idiosyncrasy in the Macintosh Operating System, a point size of 0 may appear to be the same value as a point size of 12.

The font style should be one or more of the following style keywords. Multiple font styles are allowed. A `:plain` font style implies the absence of other font styles.

```
:plain      :bold       :condense   :extend
:italic     :outline    :shadow     :underline
```

The transfer mode should be one of the following transfer-mode keywords. These transfer modes are described in Appendix D: QuickDraw Graphics.

```
:srcCopy    :srcOr      :srcXor     :srcBic
:srcPatCopy :srcPatOr   :srcPatXor  :srcPatBic
```

A font specification can have one of 256 colors. The colors are represented by their index into the operating system's 8-bit color table, with the exception that color index 0 indicates the default foreground color. The color in a font spec should be a list of the form `(:color` *x*`)` or `(:color-index` *y*`)`, where *x* is a 24-bit MCL color as returned by `make-color` and *y* is an integer between 0 and 255 inclusive.

An error is signaled if more than one name, size, color, or transfer mode is given in a single font specification.

The following are examples of legal font specifications:

```
"New York"
"nEw YOrk"
("Monaco" 9)
("Monaco" :extend :shadow 57 :srcPatCopy)
:srcCopy
:outline
(12 :srcCopy)
("Monaco" 12 :bold (:color #.*red-color*))
("Chicago" 9 (:color-index 5))
```

Implementation of font codes

Font codes are the four numbers used by the Macintosh computer to represent fonts. These numbers are stored in GrafPort, CGrafPort, and TERec records.

```
(defrecord grafport
  ...
  (txFont integer)
  (txFace unsigned-byte)
  (txMode integer)
  (txSize integer)
  ...)
```

Macintosh Common Lisp encodes these 64 bits of information as two fixnums, the font-face code (ff) and the mode-size code (ms). (The field `txFace` is only 8 bits, but an alignment byte follows it in the record.)

The font-face layout looks like this:

```
 ---------------------------------
|    txFont     | txFace | unused |
 ---------------------------------
 31           16 15     8 7      0
```

The mode-size layout looks like this:

```
 --------------------------------
|    txMode     |     txSize      |
 --------------------------------
 31           16 15             0
```

Note that since MCL fixnums use only 29 bits, you get only 13 bits of the 16-bit `txFont` and `txMode` fields.

You can find more about the meaning of these codes in the QuickDraw information in *Inside Macintosh*. Macintosh Common Lisp provides high-level functions to manipulate them for you.

Functions related to font specifications

The following functions implement and use font specifications.

`real-font` [*Function*]

Syntax `real-font &optional` *font-spec*

Description The `real-font` function returns `t` if *font-spec* corresponds to a font or font size that actually exists in the system (in other words, that is not a calculated font). Otherwise, the function returns `nil`.

The font style and transfer mode are ignored by `real-font`. If *font-spec* is not supplied, the font specification of the current GrafPort is used.

Argument *font-spec* A font specification.

`font-spec` [*Function*]

Syntax `font-spec` *ff-code ms-code*

Description The `font-spec` function creates a font specification from font codes.

Arguments *ff-code* The font-face code. A font-face code is a 32-bit integer that combines the encoded name of the font and its face (plain, bold, italic, and so on).

ms-code	The mode-size code. A mode-size code is a 32-bit integer that indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size.

Example

Here is an example of translating between font codes and font specifications.

The font-face and mode-size codes are the first two values returned by `font-codes`:

```
? (font-codes '("Monaco" 9 :srcor :plain))
262144
65545
-256
-1
```

The function `font-spec` can regenerate the font specification from them:

```
? (font-spec 262144 65545)
("Monaco" 9 :SRCOR :PLAIN)
```

`string-width` [*Function*]

Syntax `string-width` *string* `&optional` *font-spec*

Description The `string-width` function returns the width in pixels of *string*, as if it were displayed in the font, size, and style of *font-spec*.

If *font-spec* is not supplied, the font specification of the current GrafPort is used.

See also `font-codes-string-width` on page 82.

Arguments

font-spec	A font specification.
string	A string.

Example

```
? (string-width "Hi there" '("Monaco" 9 :PLAIN))
48
```

`grafport-write-string` [*Macro*]

Syntax `grafport-write-string` *string start end*

Description The `grafport-write-string` macro draws the portion of *string* between *start* and *end* to the current GrafPort, which is usually set up by `with-focused-view` or `with-port`. Drawing begins at the pen position. The macro expands into a call to the `#_DrawString` trap.

Arguments

string	A string.
start	The beginning of the string to write.
end	The end of the string to write.

Example

The generic function `stream-write-string` could be written as follows. (This version does not handle strings that contain newlines.)

```
? (defmethod stream-write-string
       ((stream simple-view) string start end)
  (with-font-focused-view stream
    (grafport-write-string string start end)))
STREAM-WRITE-STRING
```

---

font-info [*Function*]

Syntax `font-info &optional` *font-spec*

Description The `font-info` function returns four values that represent (in pixels) the ascent, descent, maximum width, and leading of *font-spec*.

The ascent is the distance from the baseline to the highest ascender of the font, the descent is the distance from the baseline to the lowest descender of the font, the maximum width is that of the widest character in the font, and the leading is the suggested spacing between lines. Only the font and font-size aspects of *font-spec* are used in the calculation. The font styles and transfer mode are not significant.

If *font-spec* is `nil` or not supplied, the font specification of the current GrafPort is used.

Argument *font-spec* A font specification.

Example

```
? (defun line-height (font-name font-size)
   (multiple-value-bind (ascent descent maxwidth leading)
                        (font-info (list font-name
                                         font-size))
    (declare (ignore maxwidth))   ;We don't use this value.
    (+ ascent descent leading)))
LINE-HEIGHT
? (line-height "new york" 12)
16
? (line-height "new york" 24)
32
? (line-height "times" 10)
12
```

---

view-font

[*Generic function*]

Syntax

`view-font` (*view* `simple-view`)
`view-font` (*window* `window`)
`view-font` (*window* `fred-window`)
`view-font` (*window* `listener`)

Description

The `view-font` generic function returns the font specification used for drawing text in the window. Due to an idiosyncrasy of the Macintosh computer, a font size of 0 points may appear as a font size of 12 points.

In the Listener, `view-font` removes boldface text, then calls the method of `window`.

In Fred windows, `view-font` returns three values: the current font, the font at the insertion point, and a Boolean value specifying whether all the selected text is in the same font as the current font.

You should not write methods for this function; use `view-font-codes` instead.

Arguments

view	A view or simple view.
window	A window, Fred window, or Listener window.

---

set-view-font

[*Generic function*]

Syntax

`set-view-font` (*view* `simple-view`) *font-spec*

Description

The generic function `set-view-font` sets the font of *view* to *font-spec*.You should not write methods for this function; use `set-view-font-codes` instead.

Arguments

view	A simple view.
font-spec	A font specification.

Functions related to font codes

The following functions implement and use font codes.

`font-codes` [*Function*]

Syntax `font-codes` *font-spec* `&optional` *old-ff old-ms*

Description The `font-codes` function creates font codes from a font specification. It returns four values: the font-face code, the mode-size code, the ff-mask, and the ms-mask. The two latter values are masks that tell which bits were specified in the font-face and mode-size codes, respectively.

Arguments

font-spec	A font specification.
old-ff	The old font/face code. A font/face code is a 32-bit integer that combines the encoded name of the font and its face (plain, bold, italic, and so on). If there is an *old-ff*, its values are used if the new font specification specifies no value for either the font name or its face. If *old-ff* is `nil` or unspecified, it defaults to 0.
old-ms	The old mode-size code. A mode-size code is a 32-bit integer that indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size. If there is an *old-ms*, its values are used if the new font specification specifies no value for either the font mode or its size. If *old-ms* is `nil` or unspecified, it defaults to 65536 (the code for a mode of `:SRCOR` and a size of 0).

Examples

Here is an example of getting and reading font codes.

```
? (setq *print-base* 16)
10
? (font-codes '("Geneva" 9 :plain))
30000
10009
-100
FFFF
```

The `txFont` value for Geneva is 3, the `txFace` value for `:plain` is 0, the `txSize` value is 9, and the `txMode` value was not specified (hence the *ms-mask* is `#xFFFF`) but defaults to 1.

Here is an example of using old font codes to modify the returned font code:

```
? (font-codes '("Monaco" 12 :BOLD))
262400
65548
-65280
65535
? (font-codes '("Times" 15))
1310720
65551
-65536
65535
? (font-codes '("Times" 15) 262400 65548)
1310976
65551
-65536
65535
? (font-spec 1310976 65551)
("Times" 15 :SRCOR :BOLD)
```

---

font-codes-info [*Function*]

Syntax `font-codes-info` *ff ms*

Description The `font-codes-info` function returns four values that represent (in pixels) the ascent, descent, maximum width, and leading of the font specified by *ff* and *ms*.

The ascent is the distance from the baseline to the highest ascender of the font, the descent is the distance from the baseline to the lowest descender of the font, the maximum width is that of the widest character in the font, and the leading is the suggested spacing between lines. Only the font and font-size aspects of *font-spec* are used in the calculation. The font styles and transfer mode are not significant.

Arguments

ff	The font/face code.
ms	The mode/size code.

Example

```
? (setq *print-base* 10.)
10
? (multiple-value-bind (ff ms) (font-codes '("Geneva" 9))
    (font-codes-info ff ms))
10
2
10
0
? (font-info '("Geneva" 9))
10
2
10
0
```

font-codes-line-height [*Function*]

Syntax `font-codes-line-height` *ff ms*

Description The function `font-codes-line-height` returns the line height for the font specified by *ff* and *ms*.

Arguments

ff A font/face code. A font/face code is a 32-bit integer that combines the name of the font and its face (e.g., plain, bold, italic). For more information see “Functions related to font codes” on page 80.

ms A mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (e.g., inclusive-or, exclusive-or, complemented) and the font size.

Example

```
? (multiple-value-bind (ff ms) (font-codes '("courier" 12 :plain))
    (font-codes-line-height ff ms))
12
```

font-codes-string-width [*Function*]

Syntax `font-codes-string-width` *string ff ms*

Description The function `font-codes-string-width` returns the width in pixels of *string* using the font specified by *ff* and *ms*.

Arguments

string A character string.

ff A font/face code. A font/face code is a 32-bit integer that combines the name of the font and its face (e.g., plain, bold, italic). For more information, see "Functions related to font codes" on page 80.

ms A mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (e.g., inclusive-or, exclusive-or, complemented) and the font size.

Example

```
? (multiple-value-bind (ff ms) (font-codes '("courier" 12
:plain))
(font-codes-string-width "hello there" ff ms))
77
```

---

view-font-codes [*Generic function*]

Syntax

`view-font-codes` (*view* `simple-view`)
`view-font-codes` (*item* `dialog-item`)
`view-font-codes` (*window* `window`)

Description The `view-font-codes` generic function returns two values, the font/face code and mode/size code for *view*'s font.

Arguments

view	A simple view.
item	A dialog item.
window	A window.

Example

```
? (setq w (make-instance 'window
            :view-font '("New York" 10 :bold)))
#<WINDOW "Untitled" #xDB5B39>
? (view-font w)
("New York" 10 :SRCOR :BOLD)
? (view-font-codes w)
131328
65546
? (font-spec 131328 65546)
("New York" 10 :SRCOR :BOLD)
```

`set-view-font-codes` [*Generic function*]

Syntax

`set-view-font-codes` (*view* `simple-view`) *ff ms* `&optional`
ff-mask ms-mask
set-view-font-codes (*item* dialog-item) *ff ms* &optional
ff-mask ms-mask
set-view-font-codes (*window* window) *ff ms* &optional
ff-mask ms-mask

Description

The generic function `set-view-font-codes` changes the view font codes of *view*. The font/face code is changed only in the bits that are set in *ff-mask*. The mode/size code is changed only in the bits that are set in *ms-mask*. These masks default to passing all bits of *ff* and *ms*.

Arguments

view	A simple view.
item	A dialog item.
window	A window.
ff	The font/face code. A font/face code is a 32-bit integer that stores the encoded name of the font and its face (plain, bold, italic, and so on). If there is no *ff*, the value of *ff* is set to 0.
ms	The mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size. If there is no *ms,* the value of *ms* is set to 0.
ff-mask	A mask that allows `set-view-font-codes` to look only at certain bits of the font/face integer. Fred dialog items and Fred windows ignore this parameter; other views and windows use it as a mask.
ms-mask	A mask that allows `set-view-font-codes` to look only at certain bits of the mode/size integer. Fred dialog items and Fred windows ignore this parameter; other views and windows use it as a mask.

Example

```
? (font-codes '("Geneva" 9))
196608
65545
-65536
65535
? (font-spec 196608 65545)
("Geneva" 9 :SRCOR :PLAIN)
? (set-view-font-codes w 196608 65545 -65536 65535)
196864
65545
? (view-font w)
```

```
("Geneva" 9 :SRCOR :BOLD)
? (set-view-font-codes w 196608 65545)
196608
65545
? (view-font w)
("Geneva" 9 :SRCOR :PLAIN)
```

`grafport-font-codes` [*Function*]

Syntax `grafport-font-codes`

Description The `grafport-font-codes` function returns two values, the font codes of the current GrafPort.

`set-grafport-font-codes` [*Function*]

Syntax `set-grafport-font-codes` *ff ms* `&optional` *ff-mask ms-mask*

Description The `set-grafport-font-codes` function sets the font codes of the current GrafPort.

Arguments

ff	The new font/face code, expressed as a fixnum.
ms	The new mode/size code, expressed as a fixnum.
ff-mask	A mask that allows `set-grafport-font-codes` to look only at certain bits of the font/face integer.
ms-mask	A mask that allows `set-grafport-font-codes` to look only at certain bits of the mode/size integer.

`wptr-font-codes` [*Function*]

Syntax `wptr-font-codes` *wptr*

Description The `wptr-font-codes` function returns the font codes of *wptr*.

Argument *wptr* A window pointer.

`set-wptr-font-codes` [*Function*]

Syntax `set-wptr-font-codes` *wptr ff ms* `&optional` *ff-mask ms-mask*

Description The `set-wptr-font-codes` function sets the font codes of *wptr* to the new font codes indicated by *ff* and *ms*.

Arguments

wptr	A window pointer.
ff	The new font/face code, expressed as a fixnum.
ms	The new mode/size code, expressed as a fixnum.
ff-mask	A mask that allows `set-wptr-font-codes` to look only at certain bits of the font/face integer.
ms-mask	A mask that allows `set-wptr-font-codes` to look only at certain bits of the mode/size integer.

`merge-font-codes` [*Function*]

Syntax `merge-font-codes` old-ff old-ms ff ms `&optional` ff-mask ms-mask

Description The `merge-font-codes` function merges two font codes.

Arguments

old-ff	The old font/face code, expressed as a fixnum. A font/face code stores the encoded name of the font and its face (plain, bold, italic, and so on). If there is no *old-ff*, the value of *old-ff* is set to 0.
old-ms	The old mode/size code, expressed as a fixnum. A mode/size code indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size. If there is no *old-ms*, the value of *old-ms* is set to 0.
ff	The new font/face code, expressed as a fixnum
ms	The new mode/size code, expressed as a fixnum.
ff-mask	A mask that allows `merge-font-codes` to look only at certain bits of the font/face integer.
ms-mask	A mask that allows `merge-font-codes` to look only at certain bits of the mode/size integer.

Examples

The function `merge-font-codes` could be written as follows:

```
(defun merge-font-codes (old-ff-code old-ms-code ff-code ms-
code
                                  &optional ff-mask ms-mask)
  (values
   (if ff-mask
     (logior (logand ff-code ff-mask)
             (logand old-ff-code (lognot ff-mask)))
     ff-code)
   (if ms-mask
```

```
    (logior (logand ms-code ms-mask)
            (logand old-ms-code (lognot ms-mask)))
    ms-code)))
```

Here is an example of merging font codes. This example is in hexadecimal.

```
(setf *print-base* 16)
10
? (font-codes '("Geneva" 9 :plain))
30000
10009
-100
FFFF
? (font-codes '(:bold :italic :notpatxor))
300
E0000
300
-10000
? (merge-font-codes #x30000 #x10009 #x300 #xe0000 #x300 #x-10000)
30300
E0009
? (font-spec #x30300 #xe0009)
("Geneva" 9 :NOTPATXOR :ITALIC :BOLD)
```

Here is a more condensed version of the same merging.

```
? (multiple-value-bind (ff ms)
        (font-codes '("Geneva" 9 :plain))
    (multiple-value-bind (bin-ff bin-ms ff-mask ms-mask)
        (font-codes '(:bold :italic :notpatxor))
      (multiple-value-bind
        (merged-ff merged-ms)
        (merge-font-codes ff ms bin-ff bin-ms
                          ff-mask ms-mask)
        (font-spec merged-ff merged-ms))))
("Geneva" 9 :NOTPATXOR :ITALIC :BOLD)
```

System data

The following symbols are bound to useful Macintosh system data.

font-list [*Variable*]

Description The `*font-list*` variable contains a list of all the fonts installed in the current Macintosh Operating System, sorted alphabetically.

pen-modes [*Variable*]

Description The `*pen-modes*` variable contains a list of pen-mode keywords.

Example

Macintosh traps (and pen-state records) encode pen modes as integers. These integers match the zero-based numeric position of the keyword in the `*pen-modes*` list. So, for example, the number of `:srcor` pen mode could be coded as `(position :srcor *pen-modes*)`. The inverse operation (turning a pen-mode integer into a keyword) can be performed with the Common Lisp function `elt`.

```
? *pen-modes*
(:srccopy :srcor :srcxor :srcbic :notsrccopy :notsrcor
:notsrcxor :notsrcbic :patcopy :pator :patxor :patbic
:notpatcopy :notpator :notpatxor :notpatbic)
? (position :srcor *pen-modes*)
1
```

style-alist [*Variable*]

Description The `*style-alist*` variable contains an association list of font-style keywords and numbers that the Macintosh computer uses to encode these styles.

The Macintosh Operating System encodes styles as a byte, with each style represented by a bit (this encoding allows multiple styles). You can derive a byte to pass to the Macintosh computer by adding the numbers corresponding to the styles listed here.

Example

```
? *style-alist*
((:plain . 0)(:bold . 1)
 (:italic . 2)(:underline . 4)
 (:outline . 8)(:shadow . 16)
 (:condense . 32)(:extend . 64))
```

`*white-pattern*` [*Variable*]

`*black-pattern*` [*Variable*]

`*gray-pattern*` [*Variable*]

`*light-gray-pattern*` [*Variable*]

`*dark-gray-pattern*` [*Variable*]

Description These variables hold Macintosh pen patterns. The patterns may be passed to traps or used with QuickDraw calls.

`*screen-width*` [*Variable*]

`*screen-height*` [*Variable*]

Description These variables contain the width and height, in pixels, of the current screen. On a Macintosh Plus or Macintosh SE computer, the width is 512 pixels and the height is 342 pixels. On a Macintosh II computer with multiple screens, the values refer to the main screen.

`*pixels-per-inch-x*` [*Variable*]

`*pixels-per-inch-y*` [*Variable*]

Description These variables contain the number of pixels per inch on the Macintosh computer screen in the horizontal and vertical directions. On a Macintosh Plus or Macintosh SE computer, both values are 72. On other Macintosh computers, the values vary according to the screen used. On a computer with multiple screens, the values refer to the main screen.

`*menubar-bottom*` [*Variable*]

Description The `*menubar-bottom*` variable holds the vertical coordinate of the first QuickDraw point below the menu bar. It is provided so that windows do not draw themselves in the area taken up by the menu bar, but use only the area below the bottom of the menu bar.

In Macintosh Common Lisp version 2, this variable is defined as `(+ (%get-word (%int-to-ptr $MBarHeight)) 18)`. Since 18 is the height of the title bar of a window with the standard window definition function, this variable has questionable utility for setting the position of any other type of window.

Chapter 3:

Menus

Contents

This chapter discusses how menus and menu items are created in Macintosh Common Lisp, how they are installed, and how you can customize them.

This chapter first discusses the class structure of menus and menu items, then discusses the associated MCL functions in detail. It describes how to add colors to menus and menu items, and discusses a specialized class, window menu items.

If you are creating your own menus or customizing the MCL menus, you should read this chapter.

A simple MCL application for editing menus is documented in Chapter 7: The Interface Toolkit.

How menus are created

In Macintosh Common Lisp, menus and menu items are instances of CLOS classes. A menu is created from the class `menu`. A menu item is created from the class `menu-item`. Both menus and menu items inherit from a direct superclass, `menu-element`, which is an abstract class; it isn't instantiated directly.

Menus appear in the menubar, the list of menus visible at the top of the screen. A menu is not visible until you use `menu–install` to add it to the menubar.

A menu is a list of menu items (which may themselves be menus). Menus can be installed at the top level of the menubar or as items on other menus, for implementing hierarchical menus.

Menus and menu items can be created at any time. They can exist, and you can perform operations on them, without being installed on the menubar. For example, menu items can be added to and removed from menus, whether or not the menus are installed in the menubar.

Because of the requirements of the Macintosh Operating System, the Apple menu is a special case; not all items can be removed from it, and it cannot be removed from the menubar.

It is often desirable to separate items in a menu into groups by placing a dotted line between the groups. A menu item whose title is the string `"-"` appears as a dotted line and cannot be selected.

A sample menu file

In the Examples folder distributed with your copy of Macintosh Common Lisp, look at `font-menus.lisp` for an annotated example of how a typical menu is created. Load `font-menus.lisp` to see the font menu in action.

The `menu-element` class

The general behavior of menus and menu items is defined by the class `menu-element`. Both `menu` and `menu-item` inherit from `menu-element`, so any method defined for `menu-element` is applicable to menus and menu items.

`menu-element` [*Class name*]

Description This is the class of menu elements, on which menus and menu items are built. This class is not instantiated directly.

The menubar

At any given point, a set of menu titles is displayed across the top of the screen. This group forms the **menubar.**

At any time, only one menubar can be displayed. Other menubars can be defined, however, and you can rotate among them.

You can use the generic function `menu-install` to install a menu in the menubar and the function `set-menubar` to change the entire menubar.

Menubar forms

The following MCL forms control menubars.

`menubar` [*Class name*]

Description The `menubar` class is built on `standard-object`. Its single instance is used to set the colors of parts of the menubar. It is not currently used for any other purpose.

menubar [*Variable*]

Description The value of the `*menubar*` variable is the single instance of the `menubar` class.

menubar [*Function*]

Syntax `menubar`

Description The `menubar` function returns a list of the menus currently installed in the menubar.

Example

```
? (menubar)
(#<APPLE-MENU ""> #<MENU "File"> #<MENU "Edit"> #<MENU
"Lisp"> #<MENU "Tools"> #<MENU "Windows">)
```

set-menubar [*Function*]

Syntax `set-menubar` *new-menubar-list*

Description The `set-menubar` function installs a new set of menus in the current menubar.

First the `menu-deinstall` function is applied to each installed menu except the Apple menu, and then the `menu-install` function is applied to each menu in *new-menubar-list*. The *new-menubar-list* may be empty, in which case the menubar is simply cleared. The function returns *new-menubar-list*.

You can never remove the Apple menu. Even if you call `(set-menubar nil)`, the Apple menu remains in the menubar.

Argument *new-menubar-list*A list of menus.

Example

```
? (setq foo (menubar))
(#<Apple-Menu "">  ;No Apple character in this font.
 #<Menu "File">
 #<Menu "Edit">
 #<Menu "Lisp">
 #<Menu "Tools">
 #<Menu "Windows">)
```

```
;Assume a menu, MY-FROGS-MENU, whose title is "Tree Frogs":
? (set-menubar (list (car foo) my-frogs-menu))
(#<Apple-Menu "">
 #<Menu "Tree Frogs">)
? (menubar)
(#<Apple-Menu "">
 #<Menu "Tree Frogs">)
```

---

find-menu [*Function*]

Syntax `find-menu` *string*

Description The `find-menu` function returns the first menu in the menubar that has *string* as its title. If no matching menu is found, it returns `nil`.

Argument *string* A string giving the title of the menu to find.

Example

```
? (find-menu "Edit")
#<MENU "Edit">
```

The built-in menus

---

default-menubar [*Variable*]

Description The variable `*default-menubar*` contains a list of the menus that are installed when you first start Macintosh Common Lisp. You may use `set-menubar` to restore the original menus after installing your own set of menus.

Note that `*default-menubar*` is simply a list of the menus present when Macintosh Common Lisp starts up. It does not contain any code for initializing these menus. If you destructively change the startup menus, then `*default-menubar*` will contain the changed menus. Calling `(set-menubar *default-menubar*)` will not undo those modifications.

Example

Here is an example of using `*default-menubar*`.

```
? (setq frogs (menubar))
(#<Apple-Menu "">
 #<Menu "Tree Frogs">)
```

```
? (set-menubar *default-menubar*)
(#<Apple-Menu "">
 #<Menu "File">
 #<Menu "Edit">
 #<Menu "Lisp">
 #<Menu "Tools">
 #<Menu "Windows">)
? (set-menubar frogs)
(#<Apple-Menu "">
 #<Menu "Tree Frogs">)
```

`*apple-menu*`, `*edit-menu*`, `*eval-menu*`, `*file-menu*`, `*lisp-menu*`, `*tools-menu*`, and `*windows-menu*`

`*apple-menu*` [*Variable*]

Description The variable `*apple-menu*` contains the Apple menu from the initial menubar. Because of the special handling of this menu by the Macintosh OS, you should be very careful adding or removing commands from it.

`*file-menu*` [*Variable*]

Description The variable `*file-menu*` contains the File menu from the initial menubar.

`*edit-menu*` [*Variable*]

Description The variable `*edit-menu*` contains the Edit menu from the initial menubar.

`*lisp-menu*` [*Variable*]

Description The variable `*lisp-menu*` contains the Lisp menu from the initial menubar.

`*tools-menu*` [*Variable*]

Description The variable `*tools-menu*` contains the Tools menu from the initial menubar.

`*windows-menu*` [*Variable*]

Description The variable `*windows-menu*` contains the Windows menu from the initial menubar. Because of MCL's special handling of this menu, you should take care adding and removing menu-items from it.

Menubar colors

Menu titles in the menubar can be colored. You can set the background color of the menubar, give menus and menu items a default color, and specify a default background color for pull-down menus.

The following functions, defined on the class `menubar`, operate on colors.

`part-color` [*Generic function*]

Syntax `part-color` (*menubar* `menubar`) *part*

Description The `part-color` generic function returns the color of *part*, a part of the menubar. See Chapter 6: Color for a description of color encoding.

Arguments

menubar The current menubar, the only instance of the class `menubar`.

part A keyword specifying which part of the menubar should be set. The four possible keywords have the following effects:

`:default-menu-title`
The default color used for the titles of menus in the menubar.

`:default-menu-background`
The default color used for the background of the pull-down menus accessed from the menubar.

`:default-menu-item-title`
The default color used for the titles of menu items.

`:menubar` The background color of the menubar itself.

Example

```
? (part-color *menubar* :menubar)
16777215
```

---

set-part-color *[Generic function]*

Syntax `set-part-color :after` (*menubar* `menubar`) *part color*

Description The `set-part-color` generic function sets the color of *part*, a part of the menubar, to *color*.

Arguments

menubar	The current menubar, the only instance of the class `menubar`.
part	A keyword specifying which part of the menubar should be set. The keywords are the same as for `part-color`.
color	The new color, encoded as an integer. (See Chapter 6: Color)

Example

```
? (set-part-color *menubar* :menubar *red-color*)
14485510
```

---

part-color-list *[Generic function]*

Syntax `part-color-list` (*menubar* `menubar`)

Description The `part-color-list` generic function returns a property list of keywords and colors for all the parts of the menubar.

Argument

menubar	The current menubar, the only instance of the class `menubar`.

Example

```
? (part-color-list *menubar*)
(:MENUBAR 14485510 :DEFAULT-ITEM-TITLE 0 :DEFAULT-MENU-
BACKGROUND 16777215 :DEFAULT-MENU-TITLE 0)
```

Menus

Menus contain sets of menu items. Menus can be added to the menubar, or they can be added to other menus. When they are added to other menus, they are treated as menu items; hierarchical menus are implemented in this way.

MCL forms relating to menus

The following MCL forms control menus.

`menu` [*Class name*]

Description The class of menus, built on `menu-element`. All menus are instantiated on the class `menu` or one of its subclasses. There are no built-in subclasses of `menu`, but you can define subclasses.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance (`*menu* `menu) &rest` *initargs*

Description This generic function initializes the menu so that you can add menu items to it and install it. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`; see the example that follows.)

The `initialize-instance` function initializes the menu but does not add it to the menubar. To add the menu, use the function `menu-install`.

Arguments

menu A menu.

initargs A set of initialization arguments and values used for initializing the menu:

`:menu-title`
A string giving the title of the menu. The default is `"Untitled"`.

`:menu-items`
A list of items to be added to the newly created menu.

`:menu-colors`
A property list of menu parts and colors. The allowable parts are given in the definition of `set-part-color`.

`:update-function`
: A function to be run when the menu item is updated. The default is `nil`.

`:help-spec`
: A value describing the Balloon Help for the menu. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

Example

```
? (setq food-menu (make-instance 'menu
              :menu-title "Food"
              :menu-colors '(:menu-title #.*red-color*)))
#<MENU "Food">
? (setq bar-menu (make-instance 'menu
              :menu-title "Bar"
              :menu-colors '(:menu-title #.*blue-color*)))
? (menu-title food-menu)
"Food"
? (menu-installed-p food-menu)
NIL           ;Not yet installed in the menubar
```

`menu-title` [*Generic function*]

Syntax `menu-title` (*menu* `menu`)

Description The `menu-title` generic function returns the title of the menu as a string.

Argument *menu* A menu.

`set-menu-title` [*Generic function*]

Syntax `menu-title` (*menu* `menu`) *new-title*

Description The `set-menu-title` generic function sets the menu title to *new-title* and returns *new-title.*

If the menu is installed, the change in title is immediately reflected in the menubar.

Arguments *menu* A menu.
new-title A string.

Example

```
? (menu-title food-menu)
"Food"
? (set-menu-title food-menu "Chinese Menu")
"Chinese Menu"
? (menu-title food-menu)
"Chinese Menu"
```

---

menu-install [*Generic function*]

Syntax `menu-install` (*menu* `menu`)

Description The `menu-install` generic function adds the menu to the menubar at the rightmost position. It returns `t`.

Argument *menu* A menu.

Example

```
? (menu-install food-menu)
T
```

---

menu-deinstall [*Generic function*]

Syntax `menu-deinstall` (*menu* `menu`)

Description The `menu-deinstall` generic function removes a menu from the menubar. It returns `nil`.

You may reinstall the menu at a later time.

Argument *menu* A menu.

Example

```
? (menu-deinstall food-menu)
NIL
```

---

menu-installed-p [*Generic function*]

Syntax `menu-installed-p` (*menu* `menu`)

Description The `menu-installed-p` generic function returns `t` if the menu is installed and `nil` if the menu is not installed.

Argument *menu* A menu.

Example

```
? (menu-installed-p food-menu)
NIL
```

`menu-disable` [*Generic function*]

Syntax `menu-disable` (*menu* `menu`)

Description The `menu-disable` generic function disables a menu. Its items may still be viewed, but they cannot be chosen. The menu and its items appear dimmed. This function has no effect if the menu is already disabled.

Menus can be enabled and disabled at any time. The effects are visible only when the menu is installed in the current menubar.

Argument *menu* A menu.

`menu-enable` [*Generic function*]

Syntax `menu-enable` (*menu* `menu`)

Description The `menu-enable` generic function enables a menu, making it possible to choose its items. This function has no effect if the menu is already enabled.

Menus can be enabled and disabled at any time. The effects are visible only when the menu is installed in the current menubar.

Argument *menu* A menu.

`menu-enabled-p` [*Generic function*]

Syntax `menu-enabled-p` (*menu* `menu`)

Description The `menu-enabled-p` generic function returns `t` if the menu is enabled and `nil` if the menu is disabled.

Argument *menu* A menu.

`menu-style` [*Generic function*]

Syntax `menu-style` (*menu* `menu`)

Description The `menu-style` generic function returns the font style in which the menu appears.

Styles are `:plain`, `:bold`, `:italic`, `:shadow`, `:outline`, `:underline`, `:condense`, and `:extend`. The keyword `:plain` indicates the absence of other styles.

Argument *menu* A menu.

`menu-update-function` [*Generic function*]

Syntax `menu-update-function` (*menu* `menu`)

Description The `menu-update-function` generic function returns the function that is run when the menu is updated.

Argument *menu* A menu.

MCL forms relating to elements in menus

The following generic functions are used to add elements to menus, remove elements from menus, find an element in a menu, and return the elements in a menu. The element may be either a menu or a menu item.

`add-menu-items` [*Generic function*]

Syntax `add-menu-items` (*menu* `menu`) `&rest` *menu-items*

Description The `add-menu-items` generic function appends *menu-items* to the menu. The new items are added to the bottom of the menu in the order specified. The function returns `nil`.

Arguments *menu* A menu.

menu-items Any number of menus and menu items to be added to the menu.

Example

```
?  (add-menu-items food-menu
    (make-instance 'menu-item
      :menu-item-title "Stir-Fried Beep"
      :menu-item-action  #'(lambda ()
                             (ed-beep)))
    (make-instance 'menu-item
      :menu-item-title "Egg Foo Bar"
      :menu-item-action
            #'(lambda ()
                (get-string-from-user
                  "How would you like your eggs?"))))
```

`remove-menu-items` [*Generic function*]

Syntax `remove-menu-items` (*menu* `menu`) `&rest` *menu-items*

Description The `remove-menu-items` generic function removes *menu-items* from the menu. The removed *menu-items* may be reinstalled later or installed in other menus. It is not an error to attempt to remove an item that is not in the menu. The `remove-menu-items` function returns `nil`.

Arguments

menu	A menu.
menu-items	Any number of menus and menu items to be removed from the menu.

Example

```
? (apply #'remove-menu-items food-menu
      (menu-items food-menu))
NIL
```

`menu-items` [*Generic function*]

Syntax `menu-items` (*menu* `menu`) `&optional` *menu-item-class*

Description The `menu-items` generic function returns a list of the menu items installed in the menu.

The menu items are listed in the order in which they appear in the menu.

Arguments

menu	A menu.

menu-item-class
: The class from which the returned menu items inherit. The default value is `menu-element`. Only those menu items that inherit from *menu-item-class* are included in the list that is returned.

Example

```
? (menu-items food-menu)
(#<MENU-ITEM "Stir-Fried Beep">
 #<MENU-ITEM "Egg Foo Bar">)
```

`find-menu-item` [*Generic function*]

Syntax `find-menu-item` (*menu* `menu`) *title*

Description The `find-menu-item` generic function returns the first menu item in the menu whose name is *title*, which should be a string. If no menu item is titled *title*, `nil` is returned.

Arguments

menu	A menu.
title	A string giving the name of the menu item to find.

Example

```
? (find-menu-item food-menu "Beep")
NIL
? (find-menu-item food-menu "Stir-Fried Beep")
#<MENU-ITEM "Stir-Fried Beep">
```

`help-spec` [*Generic function*]

Syntax `help-spec` (*menu-element* `menu-element`)

Description The `help-spec` generic function returns the text of the Balloon Help associated with *menu-element*. If it has none, `nil` is returned.

Argument *menu-element* A menu or menu item.

MCL forms relating to colors of menu elements

Like the menubar, menus and parts of menus can be colored.

`part-color` [*Generic function*]

Syntax `part-color` (*menu* `menu`) *part*

Description The `part-color` generic function returns the color of *part*, a part of the menu. See Chapter 6: Color for a description of color encoding.

Arguments

menu A menu.

part A keyword specifying a part of the menu. The three possible keywords have the following meanings:

`:menu-title`
The color in which the title of the menu is displayed in the menubar.

`:menu-background`
The color used for the background of the pull-down menu.

`:default-menu-item-title`
The default color used for the titles of items in the menu.

Example

```
? (part-color food-menu :menu-title)
14485510
```

`set-part-color` [*Generic function*]

Syntax `set-part-color` (*menu* `menu`) *part color*

Description The `set-part-color` generic function sets the color of *part*, a part of the menu specified by the arguments, and returns *color*.

Arguments

menu A menu.

part A keyword specifying which part of the menu should be set. The keywords are the same as for `part-color`.

color The new color, encoded as an integer. (See Chapter 6: Color.)

Example

```
? (set-part-color food-menu :menu-title #.*orange-color*)
16737282
```

`part-color-list` [*Generic function*]

Syntax `part-color-list` (*menu* `menu`)

Description The `part-color-list` generic function returns a property list of part keywords and colors for all the parts of the menu.

Argument *menu* A menu.

Example

```
? (part-color-list food-menu)
(:MENU-TITLE 17630104)
```

Advanced menu features

The advanced menu programmer may find the following MCL forms useful.

menu-update [*Generic function*]

Syntax `menu-update` (*menu* `menu`)

Description The `menu-update` generic function is called whenever the user clicks in the menubar or presses a command-key equivalent. The `menu-update` method for menus calls the menu's `menu-update-function` on *menu* if it has one; otherwise it calls `menu-item-update` on each item in the menu. This facility is provided so that menus and menu items can be adjusted to the current program context before they are displayed. (For example, an item may be checked or unchecked, enabled or disabled, added, removed, or reordered.)

You can specialize `menu-update`, but you normally do not need to call it. (It is called by the MCL run-time system.)

Argument *menu* A menu.

menu-handle [*Generic function*]

Syntax `menu-handle` (*menu* `menu`)

Description If the menu is installed, the `menu-handle` generic function returns the handle to the menu's menu record on the Macintosh heap. If the menu is not installed, `menu-handle` returns `nil`.

The menu handle can be useful when low-level operations are performed with the Macintosh ROM. You should not modify this value.

Argument *menu* A menu.

Example

```
? (menu-handle food-menu)
#<A Mac Handle, Unlocked, Size 34 #x6118EC>
```

---

menu-id [*Generic function*]

Syntax `menu-id` (*menu* `menu`)

Description If the menu is installed, the `menu-id` generic function returns the unique numeric ID of the menu, used by the Macintosh Operating System. If the menu is not installed, this function returns `nil`. If a menu is removed from the menubar and later reinstalled, it may be given a different ID.

Argument *menu* A menu.

Example

```
? (menu-id food-menu)
12
```

---

menu-id-object-alist [*Variable*]

Description The `*menu-id-object-alist*` variable contains an association list mapping menu ID numbers (used by the Macintosh Operating System) to MCL menu objects. You may wish to look at this list, but you should not modify it.

---

menubar-frozen [*Variable*]

Description The `*menubar-frozen*` variable is typically bound to `t` while several menu changes are made. Once the changes are complete, a call to `draw-menubar-if` draws the new menubar all at once. This mechanism can prevent undue flickering of the menubar.

If the value of this variable is true, no menubar redrawing will occur.

If the value of this variable is `nil`, the menubar will be redrawn.

If you use `*menubar-frozen*`, it is up to you to later call `draw-menubar-if`. The menubar is not redrawn automatically.

`draw-menubar-if` [*Function*]

Syntax `draw-menubar-if`

Description The `draw-menubar-if` function redraws the menubar (by calling the trap `#_DrawMenuBar`) if the value of `*menubar-frozen*` is `nil`. If the value of `*menubar-frozen*` is not `nil`, no action is taken.

Menu items

Menu items form the bodies of menus. They are instances of the class `menu-item`, which is a subclass of `menu-element`. Every menu item is associated with some action, or occasionally with `nil`, which means the menu item does nothing.

When you create an instance of a menu item, you include a value for the `:menu-item-action` initialization argument; that value should be a function of no arguments. You can get that value with the accessor function `menu-item-action-function` and change it with `set-menu-item-action-function`.

Whenever the user chooses a menu item (by either clicking it or pressing a key equivalent), the current program is interrupted and the menu item's definition of the generic function `menu-item-action` is run. The default `menu-item-action` calls `(menu-item-action-function` *menu-item*`)` and applies the result to no arguments.

You can specialize this behavior for your own menu items.

When `menu-item-action` returns, execution of the previous program resumes. (The value returned by the call to `menu-item-action` is not used.)

Here is an example of a menu item definition with a simple value for the `:menu-item-action` initialization argument.

```
(MAKE-INSTANCE 'MENU-ITEM
                :MENU-ITEM-TITLE "Beep three times"
                :MENU-ITEM-ACTION
                     #'(LAMBDA NIL
                         (ED-BEEP)
                         (ED-BEEP)
                         (ED-BEEP)))
```

The `menu-item-action-function` method is executed at interrupt level, and further event processing is disabled while it is executed. Therefore, if a menu item initiates a lengthy process, the process shouldn't be executed directly as a `menu-item-action`; instead, it should be inserted into the normal read-eval-print loop using the function `eval-enqueue`. For a complete description of `eval-enqueue`, see Chapter 10: Events.

MCL forms relating to menu items

The following MCL forms are provided for programming menu items.

The forms specialized on `menu-element` can also be applied to menus installed as hierarchical menus.

`menu-item` [*Class name*]

Description The `menu-item` class, built on the class `menu-element`, is used to create menu items.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*menu-item* `menu-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `menu-item` initializes a menu item so that it can be installed in a menu. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

menu-item A menu item.

initargs The initialization arguments for the menu item and their initial values, if any:

`:owner` The menu in which the menu item is installed. The default value is `nil`.

`:menu-item-title`
The title of the menu item. The default value is `"Untitled"`.

`:command-key`
If the value of `:command-key` is `nil`, then the menu item has no keyboard equivalent. If the value of `:command-key` is a character, then that character key is the equivalent.

`:menu-item-action`
: The action performed when the menu item is selected. This may be a function or a symbol with a function binding. The accessors for this initialization argument are `menu-item-action-function` and `set-menu-item-action-function`.

`:disabled`
: If the value of `:disabled` is true, the menu item is disabled.

`:menu-item-colors`
: A property list of part keywords and colors. See the `set-part-color` method for menu items, described in "MCL forms relating to menu item colors" on page 118.

`:menu-item-checked`
: The value of this keyword may be `t`, `nil`, a character, or a number indicating the check mark of the menu item. The values have the same meanings as for the function `set-menu-item-check-mark.`

`:style`
: A keyword or list of keywords indicating the font style of the menu item. See the description of the function `menu-item-style` later in this section.

`:update-function`
: A function to be run when the menu item is updated. The default is `nil`. The accessors of this argument are `menu-item-update-function` and `set-menu-item-update-function`.

`:help-spec`
: A value describing the Balloon Help for the menu item. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

Example

```
? (setq yu-shiang-kitty-paws
      (make-instance 'menu-item
         :menu-item-title "Yu Shiang Kitty-Paws"
         :help-spec "Prints a horrible pun."
         :menu-item-action
         #'(lambda ()
            (print "The paws that refreshes."))))
#<MENU-ITEM "Yu Shiang Kitty Paws">
```

---

menu-item-action [*Generic function*]

Syntax `menu-item-action` (*menu-item* `menu-item`)

Description The `menu-item-action` generic function is called whenever the user chooses the menu item or presses the keyboard equivalent. The method defined on `menu-item` calls the function that is the value of `menu-item-action` of *menu-item*.

Argument *menu-item* A menu item.

`menu-item-action-function` [*Generic function*]

Syntax `menu-item-action-function` (*menu-item* `menu-item`)

Description The `menu-item-action-function` accessor function returns the function that is the value of `menu-item-action` of *menu-item*.

Argument *menu-item* A menu item.

`set-menu-item-action-function` [*Generic function*]

Syntax `set-menu-item-action-function` (*menu-item* `menu-item`) *new-function*

Description The `set-menu-item-action-function` generic function sets the value of `menu-item-action-function` of *menu-item* to *new-function* and returns *new-function.*

Arguments

menu-item	A menu item.
new-function	The new function associated with the menu item.

`menu-item-title` [*Generic function*]

Syntax `menu-item-title` (*menu-item* `menu-item`)

Description The `menu-item-title` generic function returns the title of the menu item as a string.

Argument *menu-item* A menu item.

`set-menu-item-title` [*Generic function*]

Syntax `set-menu-item-title` (*menu-item* `menu-item`) *new-title*

Description The `set-menu-item-title` generic function sets the title of the menu item to *new-title* and returns *new-title.*

If `menu-item-title` is `"-"`, then the menu item appears as an unselectable dotted line. Such items are useful for separating sets of items in a menu.

Arguments

menu-item	A menu item.
new-title	A string, the new title of the menu.

Example

```
? (menu-item-title hot-machine-item)
"Hunan Lambda"
? (set-menu-item-title hot-machine-item "Szechuan Mac")
"Szechuan Mac"
? (menu-item-title hot-machine-item)
"Szechuan Mac"
```

`menu-item-disable` [*Generic function*]

Syntax `menu-item-disable` (*menu-item* `menu-element`)

Description The `menu-item-disable` generic function disables *menu-item* so that it cannot be chosen. The function has no effect if the menu item is already disabled.

Argument *menu-item* A menu item or menu; a menu element.

`menu-item-enable` [*Generic function*]

Syntax `menu-item-enable` (*menu-item* `menu-element`)

Description The `menu-item-enable` generic function enables a menu item so that the user can choose it. The function has no effect if the menu item is already enabled.

Argument *menu-item* A menu item or menu; a menu element.

`menu-item-enabled-p` [*Generic function*]

Syntax `menu-item-enabled-p` (*menu-item* `menu-element`)

Description The generic function `menu-item-enabled-p` returns `t` if the menu item is enabled and `nil` if the menu item is disabled.

Argument *menu-item* A menu item or menu; a menu element.

---

command-key [*Generic function*]

Syntax `command-key` (*menu-item* `menu-element`)

Description The `command-key` generic function returns the keyboard equivalent of the menu item. If there is no keyboard equivalent, the function returns `nil`.

Argument *menu-item* A menu item or menu; a menu element.

---

set-command-key [*Generic function*]

Syntax `set-command-key` (*menu-item* `menu-element`) *character*

Description The `set-command-key` generic function sets the keyboard equivalent of the menu item to *character*, or to nothing if *character* is `nil`.

To change the command key, call `set-command-key` again.

Arguments

menu-item	A menu item or menu; a menu element.
character	The character to use as the keyboard equivalent. This should be a character or `nil`. If it is `nil`, the menu item has no keyboard equivalent. Characters used as equivalents are usually uppercase.

Example

This code sets the keyboard equivalent of `yu-shiang-kitty-paws` to Command-R:

```
? (set-command-key yu-shiang-kitty-paws #\R)
NIL
```

Note that when you use this keyboard command, you do not need to type the R as an uppercase letter; that is, you press Command-R, not Command-Shift-R.

---

menu-item-check-mark [*Generic function*]

Syntax `menu-item-check-mark` (*menu-item* `menu-item`)

Description The `menu-item-check-mark` generic function returns the character currently used as a check mark beside the menu item, or `nil` if the command is not currently checked.

Argument *menu-item* A menu item.

`set-menu-item-check-mark` [*Generic function*]

Syntax `set-menu-item-check-mark` (*menu-item* `menu-item`) *new-mark*

Description The `set-menu-item-check-mark` generic function sets the character to be used as a check mark beside the menu item.

If *new-mark* is `nil`, no check mark appears next to the command. If *new-mark* is `t`, then a standard check-mark symbol (√) appears beside the command. If it is a character or the ASCII value of a character, then the corresponding character appears next to the menu item. The function returns *new-mark.*

Arguments

menu-item	A menu item.
new-mark	A character, the ASCII value of a character, `t`, or `nil`.

Example

Here is an example of putting a check mark beside the menu item `yu-shiang-kitty-paws`. (The reader macro for the check mark character is `#\CheckMark`.)

```
? (set-menu-item-check-mark yu-shiang-kitty-paws t)
#\CheckMark
```

`menu-item-style` [*Generic function*]

Syntax `menu-item-style` (*menu-item* `menu-element`)

Description The `menu-item-style` generic function returns the font style in which the menu item appears.

Styles are `:plain`, `:bold`, `:italic`, `:shadow`, `:outline`, `:underline`, `:condense`, and `:extend`. The keyword `:plain` indicates the absence of other styles.

Argument *menu-item* A menu item or menu; a menu element.

Example

```
? (menu-item-style yu-shiang-kitty-paws)
:PLAIN
```

`set-menu-item-style` [*Generic function*]

Syntax `set-menu-item-style` (*menu-item* `menu-element`) *new-styles*

Description The `set-menu-item-style` generic function sets the font style in which the menu item appears.

Styles are `:plain`, `:bold`, `:italic`, `:shadow`, `:outline`, `:underline`, `:condense`, and `:extend`. The keyword `:plain` indicates the absence of other styles.

Arguments

menu-item	A menu item or menu; a menu element.
new-styles	A keyword or list of keywords. Allowable keywords are `:plain`, `:bold`, `:italic`, `:shadow`, `:outline`, `:underline`, `:condense`, and `:extend`. The keyword `:plain` indicates the absence of other styles.

Example

```
? (set-menu-item-style yu-shiang-kitty-paws
                       '(:shadow :underline))
(:SHADOW :UNDERLINE)
```

`menu-item-update` [*Generic function*]

Syntax `menu-item-update` (*menu-item* `menu-item`)

Description The generic function `menu-item-update` is called when a user clicks a menu if the menu does not have its own `menu-update-function`. In this case, `menu-item-update` is called on each menu item in the menu. The user normally does not need to call this function; it is called indirectly by the MCL event system.

Argument *menu-item* A menu item.

`menu-item-update-function` [*Generic function*]

Syntax `menu-item-update-function` (*menu-item* `menu-item`)

Description The `menu-item-update-function` generic function returns the function that is the value of `menu-item-update-function` for *menu-item*.

Argument *menu-item* A menu item.

`set-menu-item-update-function` [*Generic function*]

Syntax `set-menu-item-update-function` (*menu-item* `menu-item`) *new-function*

Description The generic function `set-menu-item-update-function` sets the function that is the value of `menu-item-update-function` of *menu-item*.

Arguments

menu-item	A menu item.
new-function	A function or a symbol naming a function.

Example

In this example, a check mark appears beside `yu-shiang-kitty-paws` if Macintosh Common Lisp is running in the Eastern time zone.

```
? (set-menu-item-update-function
   yu-shiang-kitty-paws
   #'(lambda (yu-shiang-kitty-paws)
      (set-menu-item-check-mark yu-shiang-kitty-paws
                                (= (ccl::get-time-zone) 5))))
#<Anonymous Function #x4704A6>
```

A more common use of `set-menu-item-update-function` is in a menu of fonts. Only the font used in the active window is checked; the others are unchecked. A check mark either appears beside or is removed from the commands in the font menu after the `menu-item-update` function, applied by each command, determines the font of the active window.

MCL forms relating to menu item colors

The following functions control the coloring of the menu items.

`part-color` [*Generic function*]

Syntax `part-color` (*menu-item* `menu-item`) *part*

Description The `part-color` generic function returns the color of the part of the menu item specified by *part*. See Chapter 6: Color for a description of color encoding.

Arguments

menu-item	A menu item.

part — A keyword specifying a part of the menu item. The three possible keywords have the following effects:

`:item-title`
: The color used for the title of the menu item. This is also the default color used for the keyboard equivalent and check mark.

`:item-key`
: The color used for the keyboard equivalent of the menu item.

`:item-mark`
: The color used for the check mark beside the menu item.

`set-part-color` [*Generic function*]

Syntax `set-part-color` (*menu-item* `menu-item`) *part color*

Description The `set-part-color` generic function sets the color of *part* to *color* and returns *color*.

Arguments

menu-item — A menu item.

part — A part of the menu item. The same keywords are used as for `part-color`.

color — A color.

`part-color-list` [*Generic function*]

Syntax `part-color-list` (*menu-item* `menu-item`)

Description The `part-color-list` generic function returns a property list of part keywords and colors for the colored parts of the menu item.

Argument *menu-item* — A menu item.

Window menu items

Macintosh Common Lisp provides a special class of menu items for operating on the active window. These are **window menu items.** Many menu items act only on the active window. Any window menu item that does not apply to the active window should be disabled (for example, Save should be disabled when the active window is the Search dialog box). Window menu items provide an easy way to create menu items that act on the active window. Window menu items are automatically disabled when the active window is of the wrong type.

Every window menu item should have as its `menu-item-action-function` a function, a generic function, or a symbol with a function binding. This function should take one argument, a window. When a window menu item is selected, its action function is called with the active window as the argument.

If the action function is a generic function, then the menu item is applicable only if the generic function has a method suitable for the class of the front window. If the action function cannot legally be called with the front window as its argument, the menu item is disabled.

For example, the Save command has as its `menu-item-action-function` the function `window-save`. If the active window has no method for `window-save` (for example, if the active window is the Listener), then Save is disabled. If the class of the active window has a method for `window-save` (and if a subsidiary function, `window-needs-saving-p`, returns true), then Save is enabled; choosing this menu item causes the active window to perform `window-save`.

The menu item may be affected by the context in which it is called; for example, the Undo menu item may be renamed to reflect what action will be undone (for instance, Undo Cut, Undo Typing, and so on).

Many of the built-in menu items in Macintosh Common Lisp, including Save, Save As, Revert, Print, Cut, Copy, Paste, and Select All, are window menu items. The Search menu item is not a window menu item, because the Search dialog box can stay on the screen to search whatever window is currently active.

Window menu item functions

The menu items and their corresponding functions are given in Table 3-1.

■ **Table 3-1** Window menu items

Menu item	Function
Close	`window-close`
Save	`window-save`
Save As…	`window-save-as`
Save Copy As…	`window-save-copy-as`
Revert	`window-revert`
Print…	`window-hardcopy`
Undo	`undo`
Undo More	`undo-more`
Cut	`cut`
Copy	`copy`
Paste	`paste`
Clear	`clear`
Select All	`select-all`
Execute Selection	`window-eval-selection`
Execute Buffer	`window-eval-whole-buffer`
List Definitions	`window-defs-dialog`

If a window has a definition for one of these functions, then the corresponding menu item is enabled when the window is active. If the user chooses the menu item, the function is called on the active window.

Some of these functions are internal to Macintosh Common Lisp.

Window menu item class

The following definitions control the behavior of window menu items.

`window-menu-item` [*Class name*]

Description This is the class of window menu items.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*window-menu-item* `window-menu-item)&rest` *initargs*

Description The `initialize-instance` primary method for `window-menu-item` initializes a window menu item so that it may be installed in a menu. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

window-menu-item
: A window menu item.

initargs
: The initialization arguments for the window menu item. They are the same as for menu items:

`:menu-item-title`
: The title of the window menu item.

`:command-key`
: If the value of `:command-key` is `nil`, then the window menu item has no keyboard equivalent. If the value of `:command-key` is a character, then that character key is the equivalent.

`:menu-item-action`
: The action performed when the window menu item is selected. This may be either a function or a symbol with a function binding. The accessors for this initialization argument are `menu-item-action-function` and `set-menu-item-action-function`.

`:disabled`
: If the value of `:disabled` is true, the window menu item is disabled.

`:menu-item-colors`
: A property list of part keywords and colors.

`:menu-item-checked`
: The value of this keyword may be `t`, `nil`, a character, or a number indicating the check mark of the window menu item. The values have the same meanings as for the function `set-menu-item-check-mark.`

`:style`
: A keyword or list of keywords indicating the style of the window menu item. See the description of the function `set-menu-item-style`.

`:update-function`
: A function to be run when the menu item is updated. The default is `nil`.

`:help-spec`
: A value describing the Balloon Help for the menu. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

The `:menu-item-action` specified for a window menu item is used in a special way. When the menu item is selected, the function is called with the active window as the argument. The menu item is disabled when the function is a generic function that has no method applicable to the active window.

Updating the menubar

Macintosh Common Lisp provides a convenient mechanism for updating the menubar to reflect the program state. The update routine is run whenever the user clicks a menu title in the menubar or presses a keyboard equivalent. The routine is run *before* a pull-down menu or a menu item is chosen. In this way, the menus and menu items can be changed before the user sees them.

The update routine is very simple: the generic function `menu-update` is run on every installed menu. The default version of `menu-update` runs `menu-item-update` on each of its menu items. You can specialize update behavior for a menu or menu item by defining auxiliary methods of `menu-update` or `menu-item-update`.

The `menu-item-update` primary methods are not designed to do the updating themselves, but rather call `menu-item-update-function`. If you write an entirely new menu, you can write a method for `menu-update` that handles all the menu items and not have to write any `menu-item-update` methods. An example appears in the file `view-example.lisp` in the Examples folder.

The Apple menu

The Apple menu is treated differently from other menus. In particular, the Apple menu can never be removed. Calling `menu-deinstall` on the Apple menu does nothing. One implication of this is that the Apple menu remains in the menubar even after you call `(set-menubar nil)`.

If you wish to create an application with its own About menu item in the Apple menu, first remove all the menu items from the Apple menu and then install your own. You begin with the expression

```
(apply #'remove-menu-items *apple-menu* (menu-items *apple-
menu*))
```

Don't worry: the desk accessories won't be removed! Once you have done this, you can add your own menu items to the Apple menu. Any menu items added are automatically placed above the desk accessories. Normally, an application has one About menu item and one blank line.

The Apple menu remains installed as you work on it.

Example: A font menu

The file `font-menus.lisp`, distributed with Macintosh Common Lisp and available in your MCL Examples folder, contains an example of code implementing a font menu. You can load this file to see how it works.

Chapter 4:

Views and Windows

Contents

This chapter covers the implementation of views and windows in Macintosh Common Lisp. Macintosh Common Lisp provides Macintosh windows and dialog boxes as standard MCL classes. Macintosh Common Lisp also provides facilities for you to create customized kinds of windows. The features of these parts of the MCL system are described in this chapter.

The relationship of dialogs and dialog items to views and windows is described in this chapter. They are defined in Chapter 5: Dialog Items and Dialogs.

Views and Windows

To understand how Macintosh Common Lisp handles drawing and display, it is necessary to know the relationship between the class `simple-view` and its subclasses.

The Macintosh Operating System draws and displays by means of **views.** Views and their subclasses provide generalized drawing rectangles, store information about them, and display them.

- The most generalized drawing and display class is `simple-view`, the class used for all views that do not have subviews.
- A subclass of `simple-view` is `view`, which includes all the views that contain subviews.
- The subclasses of `view` include `window` and its subclasses.

Windows govern the relationship of views to the screen. Before a view can draw itself, it must be contained in a window—a screen display mechanism. Windows cannot be contained within windows.

Until you are used to it, this relationship can be confusing. In Macintosh Common Lisp, the class `window` is a subclass of `view`, but instances of views are contained within instances of windows.

Views and windows are implemented this way because views provide a more generalized behavior than windows. Views know how to draw themselves inside *any* coordinate system. Windows know how to draw themselves inside a specialized coordinate system defined by the screen. Windows also have additional behavior to perform event handling.

Because windows have the more specialized behavior, they are a subclass of views.

For many purposes the relationship between views and windows is transparent; `window` simply calls the method for its superclass, `view`.

What simple views do

Simple views have no subviews—no subordinate display objects. In Macintosh Common Lisp, you say they **contain** no subviews. Thus they can use simpler and faster drawing methods.

Simple views are drawn and clicked while focused to their **container,** the view that contains them. Focusing on a view means installing the GrafPort of *view* as the current GrafPort and setting the clip region and origin so that drawing will occur in the coordinate system of *view*.

For interface programming, the most important built-in subclass of `simple-view` is the class of dialog items, `dialog-item.`

The class `dialog-item` is a subclass of `simple-view` because dialog items have no subviews. Dialog items are drawn while focused to the dialog box or other window in which they are contained.

(Because dialog items have many specialized subclasses and methods, they are described in a separate place, Chapter 5: Dialog Items and Dialogs.)

What views do

Most graphics operations are defined on views. Views and the generic functions associated with them determine the position of the view in its coordinate system, its font, its relationship to mouse activity, and whether or not the view is currently being drawn in.

Views have other views contained within them: for instance, a view can contain simple views such as radio buttons or checkboxes.

Views draw their contents relative to their own coordinate system. Each view has its own coordinate system, with the point `(0,0)` in the upper-left corner of its content area. The position of all the view's subviews is defined by this coordinate system.

For this reason, a view's subviews are drawn after the view. For example, a static-text item in a dialog box is drawn after the dialog box.

When a view draws itself inside its container, it uses the container's coordinate system and is clipped to the boundaries of its container. For example, if a static-text item is too large to fit inside the boundaries of a dialog box, only the part of the item that fits inside the dialog box is drawn.

What windows do

Because windows are built on views, the distinction between a view and a window is transparent for many purposes. You can simply work with `window`, using both `window` and the `view` operations it inherits.

Window functions include closing a window and deallocating the associated Macintosh data structures, positioning a window on screen, sizing a window, showing and hiding windows, setting the layer of a window, determining whether the window displays in color, and ensuring that a window is on screen.

Events (such as keystrokes, presses of the mouse button, and activation events) are usually handled by the top window and its views. Views and windows can be redrawn, resized, activated, and so on, in response to events.

Macintosh Common Lisp provides several subclasses of `window`. These include

- Fred windows, used by Fred, the editor. These windows have functionality for editing text.
- Floating windows (whose class is `windoid`), a special class of window that always appears in front of other windows. Floating windows are typically used for creating tool palettes.
- Dialogs, in which you display information and initiate action in structured ways. Dialog items may appear in any view or subclass of view, not only in dialog boxes. The `dialog` class is preserved for compatibility with earlier versions of Macintosh Common Lisp, but it doesn't exist in any functional sense.

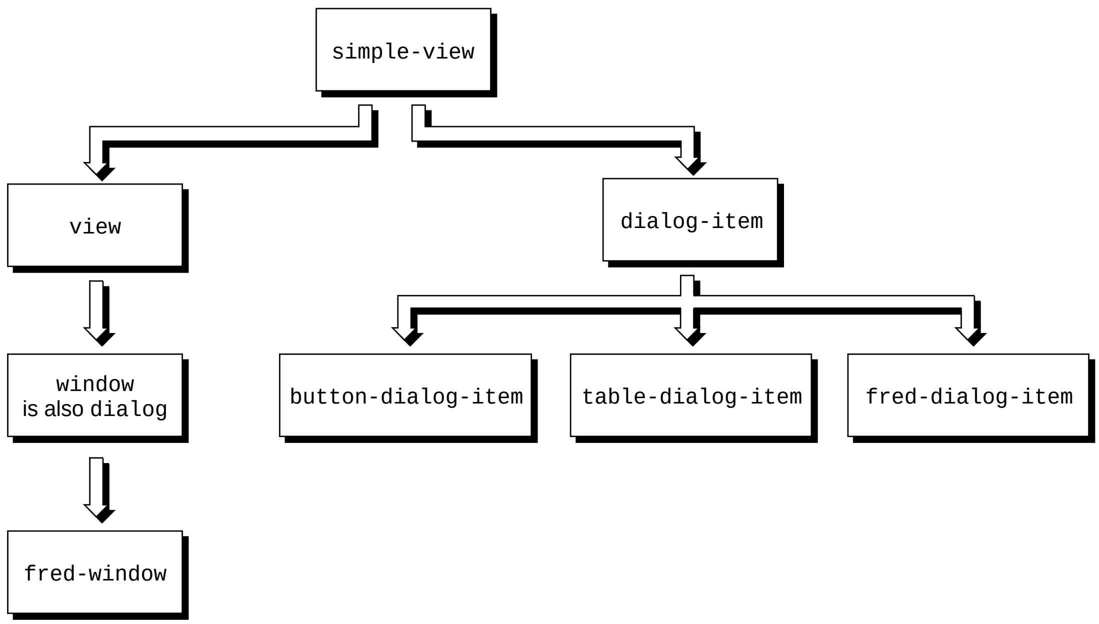
Class hierarchy of views

Figure 4-1 shows the class hierarchy of views from `simple-view` downward.

- The class `simple-view` is the parent of both `view` and `dialog-item`.
- The class `view` is the subclass of `simple-view` that defines the behavior of all views with subviews.
- The class `window` is a subclass of `view`, and `fred-window` and `dialog` (among others) are subclasses of `window`.
- The class `dialog` is simply `window` with slightly different default initial arguments, and dialog items do appear inside it exclusively; the class `window` and its subclasses are usable as dialog boxes.
- The class `dialog-item` is a subclass of `simple-view` because dialog items do not possess subviews.

  The class `dialog-item` itself is abstract. The subclasses of `dialog-item` include `button-dialog-item`, `fred-dialog-item`, and `table-dialog-item` (among others). It is these subclasses that actually have instances.

■ **Figure 4-1** The class hierarchy of views from `simple-view` downward



Summary

To summarize:

- Simple views have no subviews.
- Views have subviews.
- Views define graphics operations within other views.
- Windows define screen operations.
- Dialogs are windows with slightly different default values, good for dialog boxes.
- Fred windows have special methods to deal with, among other things, the display and editing of Lisp code and text.
- Dialog items are simple views since they have no subviews. They may appear in any view or window. The class `dialog-item` is never instantiated; only its subclasses have instances.

◆ *Note:* A window *instance* contains zero or more views (that is, it provides facilities to display zero or more views on screen), but the `window` *class* is a subclass of the `view` class.

For more information

Dialog items and dialogs are described in Chapter 5: Dialog Items and Dialogs.

For information on the size, resolution, and other physical characteristics of the display, see Chapter 2: Points and Fonts .

Information on using color is given in Chapter 6: Color .

The event-related behavior of windows and views is described in Chapter 10: Events.

Information on drawing in views with QuickDraw is given in Appendix D: QuickDraw Graphics.

MCL expressions relating to simple views and views

The following MCL forms are used to define and program simple views and views.

`simple-view` [*Class name*]

Description The class `simple-view` is the basic class of views, from which all views inherit. A simple view does not have subviews and thus can be drawn more easily. Views and dialog items are built on simple views.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*view* `simple-view`) `&rest` *initargs*

Description The `initialize-instance` primary method for `simple-view` initializes a simple view so that it can be used. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

view A simple view.

initargs A list of keywords and values used to initialize the simple view. The following keywords are available:

`:wptr` A pointer to a window record on the Macintosh heap. This record can be examined or passed to Macintosh traps that take a window pointer. The value is `nil` if the view is not contained in a window.

`:view-position`
: The position of the view in its container. The default is `(view-default-position` *view*`)`.

`:view-size`
: The size of the view. The default is `(view-default-size` *view*`)`.

`:view-nick-name`
: The nickname of the view. This keyword is used in conjunction with `view-named`. The default value is `nil`.

`:view-font`
: The font specification used by the view. The default is `nil`, which means that the view inherits its font from its container.

`:help-spec`
: A specification of a string for Balloon Help. The simplest specification is a string. For a description of the other possible `:help-spec` forms, see the file `help-manager.lisp` in your MCL Examples folder.

`:view-container`
: A view. If this argument is specified and non-`nil`, the instantiation procedure calls `set-view-container` to make this argument the container of the view being instantiated.

---

view [*Class name*]

Description The `view` class is the class of views that can include subviews. It is built on `simple-view`.

---

initialize-instance [*Generic function*]

Syntax `initialize-instance` (*view* `view`) `&rest` *initargs*

Description The `initialize-instance` primary method for `view` initializes a view so that it can be used. (When you make an instance, use `make-instance`, which calls `initialize-instance`.)

Arguments *view* A view.

initargs A list of keywords and values used to initialize the view. The following keywords are available:

`:wptr` A pointer to a window record on the Macintosh heap. This record can be examined or passed to Macintosh traps that take a window pointer. The value is `nil` if the view is not contained in a window.

`:view-position`
: The position of the view in its container. The default is `#@(0 0)`.

`:view-size`
: The size of the view. The default is `#@(100 100)`.

`:view-nick-name`
: The nickname of the view. This keyword is used in conjunction with `view-named`. The default value is `nil`.

`:view-font`
: The font specification used by the view. The default is `nil`, which means that the view inherits its font from its container.

`:view-scroll-position`
: The initial scroll position of the view. This corresponds to the origin in a Macintosh GrafPort. The default value is `#@(0 0)`.

`:help-spec`
: A specification of a string for Balloon Help. The simplest specification is a string. For a description of the other possible `:help-spec` forms, see the file `help-manager.lisp` in your MCL Examples folder.

`:view-container`
: A view. If this argument is specified and non-`nil`, the instantiation procedure calls `set-view-container` to make this argument the container of the view being instantiated.

`:view-subviews`
: A list of the views that will be made subviews of *view.*

Example

Here is an example of a view being instantiated.

```
? (setf my-view (make-instance 'view
                 :view-scroll-position #@(20 30)
                 :view-font '("Monaco" 12)
                 :view-container (setf win
                                  (make-instance 'window))))
#<VIEW #x43C6F1>
? (view-subviews win)
#<VIEW #x43C6F1>
```

current-view [*Variable*]

Description The `*current-view*` variable is bound to the view where drawing currently occurs. See `focus-view` and `with-focused-view`.

mouse-view [*Variable*]

Description The `*mouse-view*` variable is bound to the view that the mouse is over. This variable is updated by the `window-update-cursor` generic function.

The `*mouse-view*` view is the one whose `view-cursor` method decides which cursor to select.

with-focused-view [*Macro*]

Syntax `with-focused-view` *view* {*form*}*

Description The `with-focused-view` macro executes *forms* with the current GrafPort set for drawing into *view*. This involves setting the current GrafPort and setting the origin and clip region so that drawing occurs in *view*. When the forms exit (normally or abnormally), the old view is restored.

Arguments

view	A view installed in a window, or `nil`. If `nil`, the current GrafPort is set to an invisible GrafPort.
form	Zero or more forms to be executed with the current view set.

Example

Here is an example of using `with-focused-view` to paint a round-cornered rectangle within a window `window1`, using the Macintosh trap `#_PaintRoundRect`.

```
(defparameter *w* (make-instance 'window))
(rlet ((r :rect :top 20 :left 20 :bottom 80 :right 60))
  (with-focused-view *w*
    (#_paintroundrect r 30 30)))
```

`focus-view` [*Generic function*]

Syntax `focus-view (`*view* `simple-view) &optional` *font-view*
`focus-view (`*view* `null) &optional` *font-view*

Description The `focus-view` function installs the GrafPort of *view* as the current GrafPort and sets the clip region and origin so that drawing will occur in the coordinate system of *view*.

The `focus-view` function is not normally called directly. In general, `with-focused-view` should be used when drawing to views.

Arguments

view A view installed in a window, or `nil`. If `nil`, the current GrafPort is set to an invisible GrafPort.

font-view A view or `nil`. If `nil`, the font is unchanged. If non-`nil`, the `view-font-codes` of *font-view* are installed after the rest of the focusing is completed. The default is `nil`. (See“Implementation of font codes” on page 75 for information on font codes.)

`with-font-focused-view` [*Macro*]

Syntax `with-font-focused-view` *view* `{`*form*`}*`

Description The macro `with-font-focused-view` focuses on the font of *view,* then calls `with-focused-view`.

Arguments

view A view installed in a window, or `nil`. If `nil`, the current GrafPort is set to an invisible GrafPort.

form Zero or more forms to be executed with the current view set.

Example

Stream output operations on views always use `with-font-focused-view`. Hence, you need to use `with-font-focused-view` explicitly only if you need to do lower-level output. Here is an example.

```
(defvar *w* (make-instance 'window))
(defvar *view* (make-instance 'view
                 :view-container *w*
                 :view-font '("Times" 12)
                 :view-size (view-size *w*)
                 :view-position #@(0 0)))
(with-pstrs ((s "Hello there."))
```

```
(terpri *view*)
(with-focused-view *view*
  ; This string will draw in the default font
  (#_DrawString s))
(terpri *view*)
(with-font-focused-view *view*
  ; This string will draw in times 12 font.
  (#_DrawString s)))
```

---

view-container [*Generic function*]

Syntax `view-container` (*view* `view`)

Description The `view-container` generic function returns the view's containing view.

Argument *view* A view or subview, but not a window. Instances of `window` cannot have containers.

---

set-view-container [*Generic function*]

Syntax `set-view-container` (*view* `view`) *new-container*

Description The `set-view-container` generic function sets *view*'s containing view to *new-container*. If *view*'s window is changed by giving it a new container, `remove-view-from-window` is called on *view* and the old window, and `install-view-in-window` is called on *view* and the new window.

Arguments *view* A view or subview, but not a window. Instances of `window` cannot have containers. If `set-view-container` is called on a window, it signals an error.

new-container The new container of the view.

---

install-view-in-window [*Generic function*]

Syntax `install-view-in-window` (*view* `simple-view`) *window*
`install-view-in-window` (*view* `view`) *window*

Description The generic function `install-view-in-window` installs *view* in the window *window*.

This function performs initialization tasks that require the containing window. It should never be called directly by user code. However, it may be shadowed. Specialized versions of `install-view-in-window` should always perform `call-next-method`.

Arguments *view* A view or subview, but not a window. Instances of `window` cannot have containers.

window A window.

`remove-view-from-window` [*Generic function*]

Syntax `remove-view-from-window` (*view* `simple-view`)
`remove-view-from-window` (*view* `view`)

Description The generic function `remove-view-from-window` removes *view* from its container. It should never be called directly by user code. However, it may be shadowed. Specialized versions of `remove-view-from-window` should dispose of any Macintosh data the item uses (that is, data not subject to garbage collection) and should always perform a `call-next-method`.

Argument *view* A view or subview, but not a window. Instances of `window` cannot have containers.

`subviews` [*Generic function*]

Syntax `subviews` (*view* `view`) `&optional` *subview-type*

Description The `subviews` generic function returns the subviews of *view*. If *subview-type* is present, only subviews matching that type are returned.

Arguments *view* A view.

subview-type A Common Lisp type specifier.

`view-subviews` [*Generic function*]

Syntax `view-subviews` (*view* `view`)

Description The `view-subviews` generic function returns a vector containing all of the view's subviews. This vector should never be changed directly. It is updated automatically by calls to `set-view-container`.

Argument *view* A view.

`do-subviews` [*Macro*]

Syntax `do-subviews (`*subview-var view* `[`*subview-type*`]) {`*form*`}*`

Description For each subview of *view* of the given *subview-type*, the macro `do-subviews` executes *form* with *subview-var* bound to the subview.

Arguments

subview-var	A variable.
view	A view.
subview-type	A Common Lisp type specifier.
form	Zero or more MCL forms.

Example

Here is how `do-subviews` might be used to define a method on `map-subviews` for `view`.

```
? (defmethod map-subviews ((view view) function
                           &optional subview-type)
    (if subview-type
      (do-subviews (subview view subview-type)
        (funcall function subview))
      (do-subviews (subview view)
        (funcall function subview))))
#<STANDARD-METHOD MAP-SUBVIEWS (VIEW T)>
```

`map-subviews` [*Generic function*]

Syntax `map-subviews` (*view* `view`) *function* `&optional` *subview-type*

Description For each subview of *view* of the given *subview-type*, the generic function `map-subviews` calls *function* with the subview as its single argument.

Arguments

view	A view.
function	A function.
subview-type	A Common Lisp type specifier.

Example

Here is how `map-subviews` might be used to define a method on `subviews` for `view`.

```
? (defmethod subviews ((view view) &optional subview-type)
           (let ((result nil))
             (flet ((f (subview) (push subview result)))
               (declare (dynamic-extent #'f))
               (map-subviews view #'f subview-type))
```

```
              (nreverse result)))
#<STANDARD-METHOD SUBVIEWS (VIEW)>
```

---

view-named [*Generic function*]

Syntax `view-named` *name* (*view* `view`)

Description The `view-named` generic function returns the first subview of *view* whose nickname is *name*. The subviews are searched in the order in which they were added to *view.*

Arguments

name	Any object, but usually a symbol. Nicknames are compared using `eq.`
view	A view.

Example

Here is an example of using `view-named` to find a button nicknamed `pearlie` in the dialog `dialog1`.

```
? (view-named 'pearlie dialog1)
#<RADIO-BUTTON-DIALOG-ITEM #x374BA9>
```

---

find-named-sibling [*Generic function*]

Syntax `find-named-sibling` (*view* `simple-view`) *name*

Description The `find-named-sibling` generic function performs a search in *view*'s container and returns the first item in the container whose nickname is *name*. For example, given a dialog item *view*, it performs a search in the view that is *view*'s container to find another item with the nickname *name*. The items are searched in the order in which they were added to *view*'s container.

Arguments

view	A simple view.
name	Any object, but usually a symbol. Nicknames are compared using `eq.`

Example

The generic function `find-named-sibling` might be implemented as follows.

```
? (defmethod find-named-sibling ((view simple-view) name)
    (let ((container (view-container view)))
      (and container (view-named name container))))
```

`add-subviews` [*Generic function*]

Syntax `add-subviews` (*view* `view`) `&rest` *subviews*

Description The `add-subviews` generic function sets the container of each of *subviews* to *view*.

If any of the subviews are already owned by *view*, `add-subviews` does nothing.

Arguments

view	A view.
subviews	A view or simple view, but not a window; *subviews* must be able to be contained within *view*.

Examples

This function could be defined as follows:

```
? (defmethod add-subviews ((view view) &rest subviews)
    (dolist (su subviews)
      (set-view-container su view)))
#<STANDARD-METHOD ADD-SUBVIEWS (VIEW)>
```

The following code adds a checkbox to a window, then checks to see whether it's there:

```
? (setf bim (make-instance 'window))
#<WINDOW "Untitled" #x4E42A9>
? (setf boxy (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4E5249>
? (add-subviews bim boxy)
NIL
? (subviews bim)
(#<CHECK-BOX-DIALOG-ITEM #x4E5249>)
```

`remove-subviews` [*Generic function*]

Syntax `remove-subviews` (*view* `view`) `&rest` *subviews*

Description The `remove-subviews` generic function removes each of *subviews* from *view*.

If a subview is not in *view*, an error is signaled.

Arguments

view	A view.
subviews	A view or simple view, but not a window; *subviews* must be able to be contained within *view*.

`find-clicked-subview` [*Generic function*]

Syntax

`find-clicked-subview` (*view* `simple-view`) *where*
`find-clicked-subview` (*view* `view`) *where*
`find-clicked-subview` (*view* `null`) *where*

Description

The `find-clicked-subview` generic function returns the subview of *view* that contains the point *where* in its click region. The method for `null` searches all windows for a subview containing *where* in its click region.

This function is similar to `find-view-containing-point`, but `find-clicked-subview` calls `point-in-click-region-p`, and `find-view-containing-point` calls `view-contains-point-p`. The default method of `point-in-click-region-p` for views or simple views simply calls `view-contains-point-p`, but users can write methods to make views invisible to mouse clicks.

Arguments

view	A view or subview.
where	A point in the local coordinate system of the view's container.

`view-corners` [*Generic function*]

Syntax

`view-corners` (*view* `simple-view`)
`view-corners` (*window* `window`)

Description

The `view-corners` method for `simple-view` returns two points, the upper-left and lower-right corners of *view*. The method for `window` returns the view size.

Arguments

view	A simple view or subclass of `simple-view`.
window	A window.

Example

```
? (view-corners (make-instance 'view
                  :view-position #@(10 20)
                  :view-size #@(30 40)))
1310730
3932200
? (point-string 1310730)
"#@(10 20)"
? (point-string 3932200)
"#@(40 60)"
```

invalidate-corners [*Generic function*]

Syntax `invalidate-corners` (*view* `simple-view`) *topleft bottomright* `&optional` *erase-p*

Description The `invalidate-corners` generic function calls the Macintosh trap `#_InvalRgn` on the rectangle formed by *topleft* and *bottomright* in *view*.

Arguments

view A simple view.

topleft The upper-left corner of the rectangle to invalidate.

bottomright The lower-right corner of the rectangle to invalidate.

erase-p A value indicating whether or not to add the invalidated rectangle to the erase region of *view*'s window. The default is `nil`.

invalidate-view [*Generic function*]

Syntax `invalidate-view` (*view* `simple-view`) `&optional` *erase-p*
`invalidate-view` (*view* `view`) `&optional` *erase-p*

Description The `invalidate-view` generic function invalidates *view* by running `invalidate-corners` on the region bounded by its `view-corners`.

Arguments

view A view or simple view.

erase-p A value indicating whether or not to add the invalidated region to the erase region of *view*'s window. The default is `nil`.

Example

For examples of the use of `invalidate-view`, see in your MCL Examples folder the files `view-example.lisp` and `text-edit-dialog-item.lisp`.

invalidate-region [*Generic function*]

Syntax `invalidate-region` (*view* `simple-view`) *region* `&optional` *erase-p*

Description The `invalidate-region` generic function focuses on the view and calls `#_InvalRgn`. If the value of *erase-p* is true, the function adds this region to *view*'s window erase region; the next time `window-update-event-handler` runs, it will be erased. If *erase-p* is `nil` and the window was created with the `:erase-anonymous-invalidations` initarg set to true (the default), the function adds this region to the window's explicit invalidation region; `window-update-event-handler` will not erase it.

The function `invalidate-region` is called by `invalidate-view` and `invalidate-corners`, and indirectly by `set-view-position`, `set-view-size`, and `set-view-container`.

Arguments

view	A simple view.
region	The region to invalidate. The region must be a Macintosh region handle, that is, the result of `(#_NewRgn)`.
erase-p	A value indicating whether or not to add the invalidated view to the erase region of *view*'s window. The default is `nil`.

---

validate-corners [*Generic function*]

Syntax

`validate-corners` (*view* `simple-view`) *topleft bottomright*
`validate-corners` (*view* `view`) *topleft bottomright*

Description The `validate-corners` generic function erases the previous contents of the rectangle formed by *topleft* and *bottomright* and calls `#_ValidRgn` on the rectangle. It also removes the rectangle from the erase region of *view*'s window

Arguments

view	A view or simple view.
topleft	The upper-left corner of the view to invalidate.
bottomright	The lower-right corner of the view to invalidate.

---

validate-view [*Generic function*]

Syntax

`validate-view` (*view* `simple-view`)
`validate-view` (*view* `view`)

Description The `validate-view` generic function validates *view* by running `validate-corners` on the region bounded by its `view-corners`.

Argument *view* A view or simple view.

validate-region [*Generic function*]

Syntax `validate-region` (*view* `simple-view`) *region*

Description The `validate-region` generic function focuses on the view and calls `#_ValidRgn`, removing the region from *view*'s window erase region and explicit invalidation region.

Arguments

view A simple view.

region A region. The region must be a Macintosh region handle, that is, the result of `(#_NewRgn)`.

wptr [*Generic function*]

Syntax `wptr` (*view* `simple-view`)

Description The `wptr` generic function holds the pointer to a window record on the Macintosh heap. This record can be examined or the pointer passed to Macintosh traps that require a window pointer.

This generic function returns a window pointer if the view is contained in a window, or `nil` if the view is not contained in a window.

All views contained in a given window have the same `wptr`.

Argument

view A simple view or subclass of `simple-view`.

Examples

Both a view and its subview have the same `wptr`.

```
? (setf bim (make-instance 'window))
#<WINDOW "Untitled" #x4E42A9>
? (setf boxy (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4E5249>
? (add-subviews bim boxy)
NIL
? (wptr boxy)
#<A Mac Zone Pointer Size 156 #x2C35B4>
? (wptr bim)
#<A Mac Zone Pointer Size 156 #x2C35B4>
```

You can test if a view's window has been closed by checking whether the value of its `wptr` slot is `nil`.

```
? (window-close bim)
NIL
```

```
? (wptr bim)
NIL
? (wptr boxy)
NIL
```

---

view-window [*Generic function*]

Syntax `view-window (`*view* `simple-view)`

Description The `view-window` generic function returns the window containing *view*, or `nil` if the view is not contained in a window. If *view* is a window, `view-window` returns the window.

Argument *view* A simple view or subclass of `simple-view`.

Example

This code checks to determine that a simple view (a checkbox dialog item) is contained in a window:

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (setf win (make-instance 'window))
#<WINDOW "Untitled" #x4CFBE9>
? (add-subviews win checkbox)
NIL
? (view-window checkbox)
#<WINDOW "Untitled" #x4CFBE9>
```

---

view-position [*Generic function*]

Syntax `view-position (`*view* `simple-view)`

Description The `view-position` generic function returns the position of the view in its container.

Argument *view* A view or simple view.

Example

This code returns the position of `checkbox`, a checkbox dialog item:

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (view-position checkbox)
262148
```

`set-view-position` [*Generic function*]

Syntax `set-view-position (`*view* `simple-view)` *h* `&optional` *v*

Description The `set-view-position` generic function sets the position of the view in its container.

The positions are given in the container's coordinate system.

Arguments

view	A view or simple view.
h	The horizontal coordinate of the new position, or the complete position (encoded as a point) if *v* is `nil` or not supplied.
v	The vertical coordinate of the new position, or `nil` if the complete position is given by *h*.

Example

This code sets the position of `checkbox`, a checkbox dialog item:

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (set-view-position checkbox #@(20 20))
1310740
? (point-string 1310740)
"#@(20 20)"
```

`view-default-position` [*Generic function*]

Syntax `view-default-position (`*view* `simple-view)`

Description The method of `view-default-position` for `simple-view` returns `#@(0 0)`. This function is called to determine the default value of the `:view-position` initarg of *view*.

Argument *view* A simple view or subclass of `simple-view`.

`view-size` [*Generic function*]

Syntax `view-size (`*view* `simple-view)`

Description The `view-size` generic function returns the size of the view.

Argument *view* A simple view or subclass of `simple-view`.

Example

This code returns the size of `checkbox`, a checkbox dialog item:

```
? (view-size checkbox)
1048596
```

set-view-size

[*Generic function*]

Syntax `set-view-size` (*view* `simple-view`) *h* `&optional` *v*

Description The `set-view-size` generic function sets the size of the view.

Arguments

view	A simple view or subclass of `simple-view`.
h	The width of the new size, or the complete size (encoded as an integer) if *v* is `nil` or not supplied.
v	The height of the new size, or `nil` if the complete size is given by *h*.

view-default-size

[*Generic function*]

Syntax `view-default-size` (*view* `simple-view`)

Description The method of `view-default-size` for `simple-view` returns `#@(100 100)`. This function is called to determine the default value of the `:view-size` initarg of *view*.

Argument *view* A simple view or subclass of `simple-view`.

view-scroll-position

[*Generic function*]

Syntax `view-scroll-position` (*view* `simple-view`)

Description The `view-scroll-position` generic function returns the current scroll position of the view, which is the coordinate of the upper-left corner of the view. This position corresponds to the origin of a Macintosh GrafPort.

Argument *view* A simple view or subclass of `simple-view`.

set-view-scroll-position

[*Generic function*]

Syntax `set-view-scroll-position` (*view* `view`) *h* `&optional` *v* *scroll-visibly*

Description The generic function `set-view-scroll-position` sets the position of the view's scroll position. It is usually called in response to a mouse click in a scroll bar. The function returns `nil`.

Arguments

view	A simple view or subclass of `simple-view`.
h	The horizontal coordinate of the new scroll position, or the complete scroll position (encoded as a point) if *v* is `nil` or not supplied.
v	The vertical coordinate of the new scroll position, or `nil` if the complete scroll position is given by *h*.
scroll-visibly	An argument specifying whether the scrolling is done immediately. If true, the function calls `#_ScrollRect` to do the scrolling immediately. Otherwise, the function invalidates the view so that it is redrawn the next time `window-update-event-handler` is called.

Example

```
? (setq foo (make-instance 'fred-window))
#<FRED-WINDOW "New" #x438D21>
? (view-scroll-position foo)
0
? (set-view-scroll-position foo 20 20)
NIL
```

---

view-nick-name

[*Generic function*]

Syntax `view-nick-name (`*view* `simple-view)`
`view-nick-name (`*view* `view)`

Description The `view-nick-name` generic function returns the nickname of the view. The nickname is used in conjunction with `view-named`.

Argument *view* A view or simple view.

---

set-view-nick-name

[*Generic function*]

Syntax `set-view-nick-name (`*view* `view)` *new-name*

Description The `set-view-nick-name` generic function sets the nickname of the view to *new-name* and returns *new-name*.

Arguments

view	A view or simple view.
new-name	A name, usually a symbol or string.

find-view-containing-point [*Generic function*]

Syntax

`find-view-containing-point (`*view* `view)` *h* `&optional` *v*
direct-subviews-only
`find-view-containing-point (`*view* `null)` *h* `&optional` *v*
direct-subviews-only

Description

The generic function `find-view-containing-point` returns the view containing the point specified by *h* and *v*. This may be the view or one of its subviews.

The null method searches all windows for a view that contains the point. The `null` class and its use are documented in *Common Lisp: The Language,* pages 780–783.

Arguments

view A view.

h The horizontal coordinate of the point, or the complete point if *v* is not supplied.

v The vertical coordinate of the point.

direct-subviews-only
If *direct-subviews-only* is `nil` (the default), the most specific view is returned; subviews are searched for subviews, and so on. If true, then only the view or one of its direct subviews is returned.

Examples

This code determines the subview of the window `win` that contains the point `#@(21 21)`.

```
? (find-view-containing-point win #@(21 21))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
```

The following code returns the view that contains the mouse, when you don't know which window it's over:

```
(find-view-containing-point nil (view-mouse-position nil))
```

view-contains-point-p [*Generic function*]

Syntax

`view-contains-point-p (`*view* `simple-view)` *where*
`view-contains-point-p (`*window* `window)` *where*

Description

The generic function `view-contains-point-p` returns `t` if *view* contains *where;* otherwise it returns `nil`. The method for `simple-view` takes *where* in the coordinates of the parent view; the method for `window` uses its own coordinates..

Arguments	*view*	A simple view or view.
	window	A window.
	where	The cursor position in the local coordinate system of the view's container when the mouse is clicked. If *view* is a window, the cursor position in the window's coordinate system.

`point-in-click-region-p` [*Generic function*]

Syntax `point-in-click-region-p` (*view* `simple-view`) *where*

Description The generic function `point-in-click-region-p` is called by `view-click-event-handler` to determine whether *where* is in *view*. The default method calls `view-contains-point-p`..

Arguments	*view*	A simple view or view.
	where	For a view, the cursor position of the view in the local coordinate system when the mouse is clicked. For a simple view, the cursor position of the simple view in the local coordinate system of the view's container when the mouse is clicked.

`view-activate-event-handler` [*Generic function*]

Syntax `view-activate-event-handler` (*view* `simple-view`)
`view-activate-event-handler` (*view* `view`)

Description The generic function `view-activate-event-handler` is called by the event system when the window containing the view is made active.

The definition for `simple-view` does nothing. The definition for `view` calls `view-activate-event-handler` on each subview. Specialize this generic function if your view needs to indicate visually that it is active.

Argument *view* A simple view or view.

`view-deactivate-event-handler` [*Generic function*]

Syntax `view-deactivate-event-handler` (*view* `simple-view`)
`view-deactivate-event-handler` (*view* `view`)

Description The generic function `view-deactivate-event-handler` is called by the event system to deactivate a view. It is called when the window containing the view is active and a different window is made active.

The definition for `simple-view` does nothing. The definition for `view` calls `view-deactivate-event-handler` on each subview. Specialize this generic function if your view needs to indicate visually that it has been deactivated.

Argument *view* A simple view or view.

`view-click-event-handler` [*Generic function*]

Syntax `view-click-event-handler (`*view* `simple-view)` *where*
`view-click-event-handler (`*view* `view)` *where*

Description The generic function `view-click-event-handler` is called by the event system when a mouse click occurs. The `simple-view` method does nothing. The `view` method calls `view-convert-coordinates-and-click` on the first subview for which `point-in-click-region-p` returns `t`.

The function `view-click-event-handler` scans subviews in the opposite order as does `view-draw-contents`. The first view added is the first one drawn but the last one to be queried during clicking.

If you define any `view-click-event-handler` methods for `window`, they must call `call-next-method`.

Arguments *view* A simple view or view.

where For a view, the mouse click position (the position when the mouse is clicked) of the view in the local coordinate system. For a simple view, the mouse click position of the simple view in the local coordinate system of the view's container.

Example

This function might be defined as follows, except that it does not do any consing:

```
? (defmethod view-click-event-handler-1 ((view view) where)
     (dolist (subview (nreverse (subviews view)) view)
       (if (point-in-click-region-p subview where)
         (return
           (view-convert-coordinates-and-click
               subview where view)))))
#<STANDARD-METHOD VIEW-CLICK-EVENT-HANDLER-1 (VIEW T)>
```

For further examples, see the files `grapher.lisp`, `shapes-code.lisp`, `thermometer.lisp`, and `view-example.lisp` in your MCL Examples folder.

`view-convert-coordinates-and-click` [*Generic function*]

Syntax

`view-convert-coordinates-and-click` (*view* `simple-view`) *where container*

`view-convert-coordinates-and-click` (*view* `view`) *where container*

Description

The generic function `view-convert-coordinates-and-click` runs `view-click-event-handler` on the cursor position within the view's container.

Arguments

view	A simple view or view.
where	For a view, the mouse click position (the position when the mouse is clicked) of the view in the local coordinate system. For a simple view, the mouse click position of the simple view in the local coordinate system of the view's container.
container	The view's container.

`view-draw-contents` [*Generic function*]

Syntax

`view-draw-contents` (*view* `simple-view`)
`view-draw-contents` (*view* `view`)

Description

The generic function `view-draw-contents` is called by the event system whenever a view needs to redraw any portion of its contents.

The default `simple-view` method does nothing. It should be shadowed by views that need to redraw their contents. The default `view` method calls `view-focus-and-draw-contents` on each of the view's subviews.

When `view-draw-contents` is called by the event system, the view's clip region is set so that drawing occurs only in the portions that need to be updated. This normally includes areas that have been covered by other windows and then uncovered.

Argument

view	A simple view or view.

`view-focus-and-draw-contents` [*Generic function*]

Syntax `view-focus-and-draw-contents (`*view* `simple-view)` `&optional` *visrgn cliprgn*
`view-focus-and-draw-contents (`*view* `view) &optional` *visrgn cliprgn*

Description The generic function `view-focus-and-draw-contents` is used whenever a view needs to be focused on before any portion of its contents is redrawn. The method for `view` focuses on the view, then calls `view-draw-contents` if the *visrgn* and *cliprgn* region records overlap. The method for `simple-view` focuses on the view's container, then calls `view-draw-contents`.

Arguments

view	A simple view or view.
visrgn, *cliprgn*	Region records from the view's `wptr`.

Example

The method of `view-focus-and-draw-contents` for `simple-view` shows the use of the region record arguments.

```
(defmethod view-focus-and-draw-contents
               ((view simple-view) &optional visrgn cliprgn)
  (with-focused-view (view-container view)
    (when (regions-overlap-p visrgn cliprgn)
      (view-draw-contents view))))
```

The function `regions-overlap-p` takes two arguments, which must be pointers to Macintosh regions as returned by `(#_NewRgn)`. It returns true if they have a nonempty intersection and `nil` if they do not.

`convert-coordinates` [*Function*]

Syntax `convert-coordinates` *point source-view destination-view*

Description The `convert-coordinates` function converts *point* from the coordinate system of *source-view* to the coordinate system of *destination-view*.

The source view and destination view should be in the same view hierarchy (that is, they should have a common container, or one should be contained in the other).

Arguments

point	A point, encoded as an integer.
source-view	A view in whose coordinate system *point* is given.

destination-view

A view in whose coordinate system the return point is given.

Example

Here is a way of defining `view-convert-coordinates-and-click` by means of `convert-coordinates`.

```
? (defmethod view-convert-coordinates-and-click
     ((view view) where container)
   (view-click-event-handler view
     (convert-coordinates where container view)))
#<STANDARD-METHOD VIEW-CONVERT-COORDINATES-AND-CLICK (VIEW T
T)>
```

Windows

Windows are a subclass of `view`. Their behavior is specialized on that of `view`, and they inherit slots from `view`. Windows may contain subviews, but a window cannot be a subview. (If they could, windows would attempt to display inside windows, and that is wrong: windows display views.)

Windows are used to display information on the screen. Because windows are views, graphics operations can also be performed on them. For many applications, the distinction between a window and a view is insignificant and you don't need to worry about views at all. You can simply work with windows, using both window and view operations.

The base class of windows is `window`. The features of `window` are common to all windows.

Macintosh Common Lisp also provides several subclasses of `window`. These include

- `fred-window`, a subclass of windows used for text editing. The functionality of Fred windows is discussed in Chapter 14: Programming the Editor.
- `windoid`, the class of floating windows. Floating windows always appear in front of other windows. You generally use them to create tool palettes. They are described in "Floating windows" on page 180.

- `dialog`. The `dialog` class exists for convenience. It is a subclass of the `window` class and is identical except that its default window type is `:document` instead of `:document-with-zoom`, its default title is `"Untitled Dialog"` instead of `"Untitled"`, its default size is `#@(300 200)` instead of `*window-default-size*`, and its default position is `'(:top 100)` instead of `*window-default-position*`.

  You do not need to use the `dialog` class. You can use any window to create a dialog box, and dialog items can appear in any window.

  Dialogs are described in Chapter 5: Dialog Items and Dialogs.

MCL functions for programming windows

The following MCL functions are used for creating, reporting on, and modifying windows.

`window` [*Class name*]

Description The class `window` is the class of windows, built on `view`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*window* `window`) `&rest` *initargs*

Description The `initialize-instance` primary method for `window` initializes a window so that it can be used. (You make an instance with `make-instance`, which calls `initialize-instance`.)

Arguments

window A window.

initargs A list of keywords and values used to initialize the window. The following keywords are available:

`:view-position`
A point, keyword, or list giving the initial position of the window. The default is the result of calling `view-default-position` on the window. For a description of the list form of `view-position`, see the generic function `set-view-position` later in this section.

`:auto-position`
: `A keyword or nil`, indicating an automatically calculated position for the window. These keywords correspond to the `WIND` and `DLOG` resource codes with the same names.
`nil` (same as `:noAutoCenter`)
`:noAutoCenter`
`:alertPositionParentWindow`
`:centerMainScreen`
`:staggerParentWindow`
`:alertPositionMainScreen`
`:centerParentWindowScreen`
`:staggerMainScreen`
`:alertPositionParentWindowScreen`
`:centerParentWindow`
`:staggerParentWindowScreen.`

`:view-size`
: A point giving the initial size of the window. The default is the result of calling `view-default-size` on the window.

`:view-nick-name`
: The nickname of the view. This keyword is used in conjunction with `view-named`. The default value is `nil`.

`:view-scroll-position`
: The initial scroll position of the view. This corresponds to the origin in a Macintosh GrafPort. The default value is `#@(0 0)`.

`:view-subviews`
: A list of initial subviews for the window.

`:window-title`
: A string specifying the title of the window. The default is `"Untitled"`.

`:window-show`
: If this argument is true (the default), a window is shown when it is created. If `nil`, the window is created invisibly. See `window-show` and `window-hide`.

`:view-font`
: The font specification used by the window. The default is the result of calling `view-default-font` on the window.

`:window-layer`
: An integer describing the layer in which the new window will be created. By default this is `0` (the front window). For details, see `set-window-layer`, later in this section.

`:color-p`
: If `nil` (the default), the window is a normal window created by the `#_newWindow` trap. If non-`nil`, the window is a color window, created by the `#_newCWindow` trap.

`:window-type`
: A keyword describing the type of window to be created. The default is `:document-with-zoom`. This argument should be one of the following keywords:
`:document`
`:document-with-grow`
`:document-with-zoom`
`:double-edge-box`
`:single-edge-box`
`:shadow-edge-box`
`:tool`

`:procid`
: A number indicating the window definition ID (procID) of the window to be created. This is an alternative to specifying `:window-type`, for programmers who want to use window definitions with nonstandard IDs.

`:window-do-first-click`
: A Boolean value determining whether the click that selects a window is also passed to `window-click-event-handler.` The default value is `nil`.

The click that selects an application in Multifinder is not passed to the application unless either the window clicked on is not the front window or the Get Front Clicks bit is set in the application's size resource.

`:close-box-p`
: A Boolean value determining whether the window will have a close box. Close boxes aren't available on all windows.

`:wptr`
: For use by advanced programmers, an argument used as a pointer to a window record on the Macintosh heap. Instead of creating a new window, `initialize-instance` builds a window object around the window specified by `:wptr`. This is useful when you want to create the window yourself and integrate it with the MCL window object system.

`:erase-anonymous-invalidations`
An argument determining behavior when *window* is refreshed. If the value of this initialization argument is true (the default), any parts of the invalid region of *window* that were not added by `invalidate-region` are erased when *window* is refreshed. If this value is `nil`, no extra erasing is done. Since erasing draws the background color and background pattern, and since anonymous invalidation usually happens only because a formerly covered part of the window is exposed, you usually should use the default. (The function `invalidate-region` is called by `invalidate-view` and `invalidate-corners`, and indirectly by `set-view-position`, `set-view-size`, and `set-view-container`.) If your code invalidates parts of a window without calling `invalidate-region`, for example, by calling `#_InvalRgn`, you may notice flickering on redraw if you use the default value of `:erase-anonymous-invalidations.`

Example

Here is an example of instantiating a window.

```
? (setq baz (make-instance 'window
             :window-title "Bazwin"
             :view-position #@(200 300)
             :window-type :tool
             :color-p t))
#<WINDOW "Bazwin" #x5DB8C9>
```

---

windows [*Function*]

Syntax `windows &key :class :include-invisibles :include-windoids`

Description The `windows` function returns a list of existing windows that are instances of `:class`. The list is ordered from front to back.

Arguments

`:class` A class used to filter output. Only windows that match the value of `:class` are included in the returned list. The default is `window`, which includes all windows.

`:include-invisibles`
If the value of this variable is true, invisible windows are included in the list. If false (the default), invisible windows are not included.

`:include-windoids`
: If the value of this variable is true, floating windows (the class `windoid`) are included in the list. If false (the default), floating windows are not included. Floating windows are also included if the value of the `:class` argument is `windoid`.

Examples

Here are some examples of the use of `windows`.

```
? (windows)
(#<LISTENER "Listener" #x49EB31>
 #<APROPOS-DIALOG "Apropos" #x532EF1>
 #<FRED-WINDOW "New" #x51CC61>)
? (windows :class 'fred-window)
(#<LISTENER "Listener" #x49EB31>
 #<FRED-WINDOW "New" #x51CC61>)
? (windows :class 'apropos-dialog)
(#<APROPOS-DIALOG "Apropos" #x532EF1>)
```

`front-window` [*Function*]

Syntax `front-window &key :class :include-invisibles :include-windoids`

Description The `front-window` function returns the frontmost window satisfying the arguments. If no windows satisfy the tests, `nil` is returned.

Arguments

`:class`
: A class used to filter output. The frontmost window that is an instance of the value of `:class` is returned. The default is `window`, which includes all windows.

`:include-invisibles`
: If the value of this variable is true, the frontmost window, visible or invisible, is returned. If false (the default), the frontmost visible window is returned.

`:include-windoids`
: If the value of this variable is true, the frontmost window or floating window is returned. If false (the default), the frontmost window that is not a floating window is returned.

Example

```
? (front-window)
#<LISTENER "Listener" #x5204C9>
```

`target` [*Function*]

Syntax `target`

Description The `target` function returns the second window on the list of windows; it is equivalent to `(second (windows))`.

Example

```
? (windows)
(#<LISTENER "Listener" #x49EB31>
 #<APROPOS-DIALOG "Apropos" #x532EF1>
 #<FRED-WINDOW "New" #x51CC61>)
? (target)
 #<APROPOS-DIALOG "Apropos" #x532EF1>
```

`map-windows` [*Function*]

Syntax `map-windows` *function* `&key :class :include-invisibles :include-windoids`

Description The `map-windows` function calls *function*, a function of one argument, on each window that satisfies the keywords.

Arguments

function A function of one argument.

`:class` A class used to filter output. The function *function* is called only on windows that match the value of `:class`. The default is `window`, which includes all windows.

`:include-invisibles` If the value of this variable is true, *function* is applied to both visible and invisible windows that are instances of `:class`. If the value is false, *function* is applied only to visible windows.

`:include-windoids` If the value of this variable is true, *function* is applied to floating windows. If the value is false, it is not.

Example

The following code provides a simple way to implement `front-window` using `map-windows`:

```
? (defun simple-front-window
         ()
   (let ((f #'(lambda (w)
                (return-from simple-front-window w))))
```

```
   (declare (dynamic-extent f))
   (map-windows f)))
SIMPLE-FRONT-WINDOW
```

---

find-window [*Function*]

Syntax `find-window` *title* `&optional` *class*

Description The `find-window` function returns the frontmost window of the class *class* for which a prefix of the window's title is `string-equal` to *title*. If no window has *title* as its title, `nil` is returned. (The cross that appears in the title bar of modified Fred windows is ignored when comparing the title.)

Arguments

title	A string specifying the title of the window to search for.
class	A class used to filter the result. The frontmost window that inherits from *class* is returned. The default is `window`.

Example

```
? (find-window "Listener")
#<LISTENER "Listener" #x5204C9>
? (find-window 'listener)
#<LISTENER "Listener" #x5204C9>
? (find-window "lis")
#<LISTENER "Listener" #x5204C9>
? (find-window "ist")
NIL
```

---

window-close [*Generic function*]

Syntax `window-close` (*window* `window`)

Description The `window-close` generic function closes the window. The associated Macintosh data structures will be deallocated the next time the garbage collector runs. This operation is the inverse of `initialize-instance`. When a window is closed, its state is lost and cannot be recovered.

The MCL event system calls `window-close` when the user clicks a window's close box or chooses Close from the File menu.

Argument *window* A window.

Example

You can tell if a window has been closed by determining whether `wptr` called on the window returns `nil`.

```
? (setq baz (make-instance 'window
                           :window-title "bazwin"))
#<WINDOW "bazwin" #x6143D1>
? (window-title baz)
"Bazwin"
? (wptr baz)
#<A Mac Zone Pointer Size 156 #x715930>
? (window-close baz)
NIL
? (window-title baz)
"<No title>";the window's state is lost
? (wptr baz)
NIL
```

view-position [*Generic function*]

Syntax `view-position` (*window* `window`)

Description The `view-position` generic function returns the position of the upper-left corner of the window as a point.

Argument *window* A window.

set-view-position [*Generic function*]

Syntax `set-view-position` (*window* `window`) *h* `&optional` *v*

Description The `set-view-position` generic function moves the window and returns the new position of the upper-left corner, expressed as a point.

For windows with title bars, such as document windows and tool windows, the position is not the upper-left corner of the title bar but the upper-left corner of the content area of the window.

Arguments

window A window.

h The horizontal coordinate of the new position, or the complete position.
This may also be a keyword or list specifying how to center the window.

To center a window, specify the new position as the keyword `:centered.` If the position is `:centered`, the window will be centered vertically and horizontally.

The position may also be a list of the form `(`*reference offset*`)`, where *reference* is one of the keywords `:top`, `:left`, `:bottom`, or `:right`, and *offset* is a number.

- If *reference* is `:top`, the top of the window is offset *offset* number of pixels from the top of the screen, and the window is centered horizontally.
- If *reference* is `:bottom`, the bottom of the window is offset *offset* number of pixels from the bottom of the screen, and the window is centered horizontally.
- If *reference* is `:left`, the left side of the window is offset *offset* number of pixels from the left of the screen, and the window is centered vertically.
- If *reference* is `:right`, the right side of the window is offset *offset* number of pixels from the right of the screen, and the window is centered vertically.

v The vertical coordinate of the new position, or `nil` if the complete position is given by *h*.

Examples

```
? (setq bim (make-instance 'window
                   :view-position #@(50 50)))
#<WINDOW "Untitled" #x506829>
? (point-string (view-position bim))
"#@(50 50)"
? (set-view-position bim #@(100 100))
6553700
? (point-string (view-position bim))
"#@(100 100)"
```

Here is an example of the use of `:centered`.

```
? (setq bim (make-instance 'window
                  :view-position :centered))
#<WINDOW "Untitled" #x509F59>
```

---

view-size [*Generic function*]

Syntax `view-size` (*window* `window`)

Description The `view-size` generic function returns returns the size of the window as a point.

Argument *window* A window.

`set-view-size` [*Generic function*]

Syntax `set-view-size` (*window* `window`) *h* `&optional` *v*

Description The `set-view-size` generic function sets the size of the window.

The upper-left corner of the window is anchored, and the lower-right corner moves according to the new size. If both *h* and *v* are given, they should be the new horizontal and vertical dimensions of the window. If the value of *v* is `nil` or not supplied, *h* is taken to be an encoded point holding both dimensions.

The new size is returned, expressed as a point.

Arguments

window	A window.
h	The new width of the window, or both the width and height (encoded as an integer point) if the value of *v* is `nil`.
v	The new height of the window, or `nil` if the height and width are both given by *h*.

`window-size-parts` [*Generic function*]

Syntax `window-size-parts` (*window* `window`)
`window-size-parts :before` (*window* `window`)

Description The `window-size-parts` generic function can be specialized to resize the subviews of a window whenever the size of the window is changed. This function is called directly or indirectly by the methods specialized on *window* for the generic functions `initialize-instance`, `set-view-size`, `window-zoom-event-handler`, and `window-grow-event-handler`.

The primary method for `window` does nothing. The `:before` method for `window` ensures that the `view-clip-region` and `view-origin` of each of the window's subviews are recomputed the next time they are needed. The method for `fred-window` resizes the horizontal and vertical scroll bars as well as the main text area of the window.

Argument *window* A window or Fred window.

`*window-default-position*` [*Variable*]

Description The default position of a newly opened window. The initial value is `#@(6 44)`.

`*window-default-size*` [*Variable*]

Description The default size of a newly opened window. The initial value is `#@(502 150)`.

`view-default-position` [*Generic function*]

Syntax `view-default-position` (*window* `window`)

Description When a window is created, the `view-default-position` generic function is called if no position is explicitly specified either as the `:view-position` initialization argument to `make-instance` or as a default initialization argument in the class definition. The value returned is used as the initial position of the window. It must be a valid position specifier, either a point or a centering specifier as documented under `set-view-position`. The system-supplied method specialized on `window` returns the value of `*window-default-position*`.

Argument *window* A window.

`view-default-size` [*Generic function*]

Syntax `view-default-size` (*window* `window`)

Description When a window is created, the `view-default-size` generic function is called if no size is explicitly specified either as the `:view-size` initialization argument to `make-instance` or as a default initialization argument in the class definition. The value returned is used as the initial size of the window. It must be a point. The system-supplied method specialized on `window` returns the value of `*window-default-size*`.

Argument *window* A window.

`window-title` [*Generic function*]

Syntax `window-title` (*window* `window`)
`window-title` (*window* `fred-window`)

Description The `window-title` generic function returns the window title as a string. It ignores the crosses in the title bars of modified Fred windows.

Argument *window* A window.

`set-window-title` [*Generic function*]

Syntax `set-window-title` (*window* `window`) *new-title*
`set-window-title` (*window* `fred-window`) *new-title*

Description The `set-window-title` generic function sets the window title to *new-title*. It ignores the crosses in the title bars of modified Fred windows.

Arguments

window	A window.
new-title	A string to be used as the new title.

`view-font` [*Generic function*]

Syntax `view-font` (*window* `window`)
`view-font` (*window* `fred-window`)
`view-font` (*window* `listener`)

Description The `view-font` generic function returns the font spec used for drawing text in the window. Due to an idiosyncrasy of the Macintosh computer, a font size of 0 points may appear as a font size of 12 points.

For the Listener, `view-font` changes `:bold` to `:plain` in the result of `call-next-method`.

For Fred windows, `view-font` returns three values: the current font for newly inserted characters; the font of the first character after the insertion point, or of the first character in the selection if there is a selection; and a Boolean value specifying whether all the selected text is in the same font as the current font.

Argument *window* A window, Fred window, or Listener window.

`view-default-font` [*Generic function*]

Syntax

`view-default-font` (*window* `window`)
`view-default-font` (*view* `simple-view`)
`view-default-font` (*window* `listener`)

Description

If a `:view-font` initialization argument is not specified when a view is created, the generic function `view-default-font` is called to determine its font.

The `window` method on `view-default-font` returns the value of `*fred-default-font-spec*`. The `listener` method returns the value of `*listener-default-font-spec*`. The initial value of both these variables is `("Monaco" 9 :PLAIN).` The `simple-view` method returns `nil`, meaning that the view inherits its font from its container.

Every window has a font spec associated with it, even if the window never uses fonts.

Arguments

window	A window.
view	A simple view.

`set-view-font` [*Generic function*]

Syntax

`set-view-font` (*window* `window`) *font-spec*
`set-view-font` (*window* `fred-window`) *font-spec*
`set-view-font` (*window* `listener`) *font-spec*

Description

The `set-view-font` generic function sets the font spec of *window* to *font-spec*.

Arguments

window	A window.
font-spec	A font specifier. If *font-spec* doesn't specify all four components of a font spec, the missing components are taken from the window's current font. (See Chapter 2: Points and Fonts for a complete description of font specs.)

Example

Here is an example of setting a window font.

```
? (setf freddy (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4A20A1>
? (view-font freddy)
("Monaco" 9 :SRCOR :PLAIN)
NIL
NIL
? (set-view-font freddy '(:bold 14))
(:BOLD 14)
```

```
? (view-font freddy)
("Monaco" 14 :SRCOR :BOLD)
NIL
NIL
```

For another example of the use of `set-view-font`, see the file `font-menus.lisp` in your MCL Examples folder.

`view-font-codes` [*Generic function*]

Syntax

`view-font-codes` (*view* `simple-view`)
`view-font-codes` (*window* `window`)

Description

The `view-font-codes` generic function returns two values, the font-face code and mode-size code for *view*'s font. (Font codes are a more efficient way of encoding font specs; they are described in *Inside Macintosh*.)

Arguments

view	A simple view.
window	A window.

Example

```
? (setq w (make-instance 'window
            :view-font '("New York" 10 :bold)))
#<WINDOW "Untitled" #xDB5B39>
? (view-font w)
("New York" 10 :SRCOR :BOLD)
? (view-font-codes w)
131328
65546
? (font-spec 131328 65546)
("New York" 10 :SRCOR :BOLD)
```

`set-view-font-codes` [*Generic function*]

Syntax

`set-view-font-codes` (*view* `simple-view`) *ff ms* `&optional`
ff-mask ms-mask
`set-view-font-codes` (*window* `window`) *ff ms* `&optional`
ff-mask ms-mask

Description

The `set-view-font-codes` generic function changes the view font codes of *view*. The font-face code is changed only in the bits that are set in *ff-mask*. The mode-size code is changed only in the bits that are set in *ms-mask*. These masks default to passing all bits of *ff* and *ms*.

For full details of font codes, see *Inside Macintosh.*

Arguments

view	A simple view.
window	A window.
ff	The font-face code. A font-face code is a 32-bit integer that stores the encoded name of the font and its face (plain, bold, italic, and so on). If there is no *ff*, the value of *ff* is set to 0.
ms	The mode-size code. A mode-size code is a 32-bit integer that indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size. If there is no *ms,* the value of *ms* is set to `0`.
ff-mask	A mask that allows `set-view-font-codes` to look only at certain bits of the font-face integer. Only windows use *ff-mask;* views ignore it.
ms-mask	A mask that allows `set-view-font-codes` to look only at certain bits of the mode-size integer. Only windows use *ms-mask*; views ignore it.

Example

```
? (font-codes '("Geneva" 9))
196608
65545
-65536
65535
? (font-spec 196608 65545)
("Geneva" 9 :SRCOR :PLAIN)
? (set-view-font-codes w 196608 65545 -65536 65535)
NIL
? (view-font w)
("Geneva" 9 :SRCOR :BOLD)
? (set-view-font-codes w 196608 65545)
NIL
? (view-font w)
("Geneva" 9 :SRCOR :PLAIN)
```

part-color [*Generic function*]

Syntax `part-color` (*window* `window`) *part*

Description The `part-color` generic function returns the color of the part of the window indicated by *part.*

Arguments

window	A window.

part	A keyword specifying which part of the window should be set. The five possible keywords have the following meanings:
`:content`	The frames of `:double-edge-box` windows; unused in other windows.
`:frame`	The outline of the window and the title bar of `:tool` windows.
`:text`	The title of `:document` windows.
`:hilite`	The lines in the title bar of `:document` windows.
`:title-bar`	The background of the title bar in `:document` windows and the title in `:tool` windows.

set-part-color [*Generic function*]

Syntax `set-part-color` (*window* `window`) *part color*

Description The `set-part-color` generic function sets the part of the window indicated by *part* to *color* and returns *color*, encoded as an integer. If *color* is `nil`, the default color is restored.

Arguments

window	A window.
part	A keyword specifying a part of the window whose color should be returned. The same are allowed as for `part-color`.
color	A color, either symbolic or encoded as an integer.

Example

```
? (setf fred (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4B4C79>
? (set-part-color fred :content *red-color*)
14485510
? (set-part-color fred :frame 2078484)
2078484
```

part-color-list [*Generic function*]

Syntax `part-color-list` (*window* `window`)

Description The `part-color-list` generic function returns a property list of keywords and colors for all the colored components of the window. The same keywords apply as for `part-color`. Components whose color has not been set are not included.

Argument *window* A window.

Example

```
? (part-color-list fred)
(:FRAME 2078484 :CONTENT 14485510)
```

`window-show` [*Generic function*]

Syntax `window-show` (*window* `window`)

Description The `window-show` generic function makes a window visible on the screen (assuming the window is not at an off-screen position).

Argument *window* A window.

`window-hide` [*Generic function*]

Syntax `window-hide` (*window* `window`)

Description The `window-hide` generic function makes a window invisible on the screen.

Argument *window* A window.

`window-shown-p` [*Generic function*]

Syntax `window-shown-p` (*window* `window`)

Description The `window-shown-p` generic function returns true if the window is visible, and false if it is hidden.

Argument *window* A window.

`window-ensure-on-screen` [*Generic function*]

Syntax `window-ensure-on-screen` (*window* `window`) `&optional` *default-position default-size*

Description The generic function `window-ensure-on-screen` ensures that the window is entirely visible on one or more of the Macintosh screens. It may overlap two screens, but if it is not entirely visible, as determined by `window-on-screen-p`, it is moved to the position *default-position*. If it is still not entirely visible, its size is changed to *default-size.*

This function is useful when window positions are saved and restored on Macintosh computers with different screen configurations.

If you hold down the shift key while selecting a window from the Windows menu, `window-ensure-on-screen` is called on it.

Arguments

window A window.

default-position
The position to which the window is moved if it needs to be. The default *default-position* is the value of `*window-default-position*`.

default-size The default size of the window. The default *default-size* is the value of `*window-default-size*`.

`window-on-screen-p` [*Generic function*]

Syntax `window-on-screen-p` (*window* `window`)

Description The `window-on-screen-p` generic function returns `t` if all of *window* is on the screen, `nil` otherwise.

Argument *window* A window.

`window-active-p` [*Generic function*]

Syntax `window-active-p` (*window* `window`)

Description The `window-active-p` generic function returns `t` if *window* is the active window, `nil` otherwise.

Except when Macintosh Common Lisp is not the active application, it returns `t` for all floating windows and for the frontmost non-floating visible window.

Argument *window* A window.

`window-layer` [*Generic function*]

Syntax `window-layer` (*window* `window`) `&optional` *include-invisibles*

Description The `window-layer` generic function returns the number of windows in front of *window*. Floating windows are counted.

Arguments

window A window.

include-invisibles
A Boolean value specifying whether or not to include invisible windows in the count. The default value is `nil`, indicating that `window-layer` counts only visible windows.

`set-window-layer` [*Generic function*]

Syntax

`set-window-layer` (*window* `window`) *new-layer* `&optional` *include-invisibles*
`set-window-layer` (*window* `windoid`) *new-layer* `&optional` *include-invisibles*

Description The `set-window-layer` generic function changes the layer of the window to *new-layer*. Floating windows are counted.

To make a window the frontmost window that is not a floating window, set its layer to `*windoid-count*`.

You can use `set-window-layer` to move a regular window in front of a floating window. Once other events occur, however, the floating window moves back to the front.

Arguments

window A window.

windoid A floating window.

new-layer A nonnegative integer indicating how many windows should be in front of *window*. If *new-layer* is equal to or greater than the number of windows on screen, *window* is moved all the way to the back. If the value of *new-layer* is `0`, *window* is moved to the front.

include-invisibles
A variable specifying whether the layering should take invisible windows into account. If the value of *include-invisibles* is false (the default), invisible windows are ignored. If it is true, invisible windows are counted.

`window-select` [*Generic function*]

Syntax

`window-select` (*window* `window`)
`window-select` (*window* `windoid`)
`window-select` (*window* `null`)

Description The `window-select` generic function brings a window to the front, activates it, and shows it if it is hidden. The previously active window is deactivated.

Argument *window* A window.

Advanced window features

The following operations are useful for advanced programmers working with windows.

`window-zoom-position` [*Generic function*]

Syntax `window-zoom-position` (*window* `window`)

Description The `window-zoom-position` generic function returns the zoom position of a window, that is, its position after the user clicks the zoom box. This value is either the last value given to `set-window-zoom-position` for *window* or the value returned by calling `window-default-zoom-position` on *window.*

Argument *window* A window.

`window-default-zoom-position` [*Generic function*]

Syntax `window-default-zoom-position` (*window* `window`)

Description The `window-default-zoom-position` generic function determines the default zoom position of a window, that is, its new position after the user clicks the zoom box.

Argument *window* A window.

Example

See the example under the definition of `set-window-zoom-position`.

window-default-zoom-position [*Variable*]

Description The `*window-default-zoom-position*` variable stores the default zoom position of a window, that is, its new position after the user clicks the zoom box.

This variable and `*window-default-zoom-size*` are initialized at startup to make a zoomed window fill the screen containing the menu bar with a 3-pixel border all around.

set-window-zoom-position [*Generic function*]

Syntax `set-window-zoom-position` (*window* `window`) *h* `&optional` *v*

Description The `set-window-zoom-position` generic function sets the zoom position of a window, that is, its new position after the user clicks the zoom box, and returns the new position, encoded as an integer.

Arguments

window	A window.
h	The horizontal coordinate of the new position, or the complete position (encoded as an integer) if *v* is `nil` or not supplied.
v	The vertical coordinate of the new position, or `nil` if the complete position is given by *h*.

Example

Here is an example of setting the zoom position of a class of windows and of an instance.

```
? (defclass my-window-class (window) ())
#<STANDARD-CLASS MY-WINDOW-CLASS>
? (defmethod window-default-zoom-position
       ((w my-window-class))
            #@(10 50))
#<STANDARD-METHOD WINDOW-DEFAULT-ZOOM-POSITION (MY-WINDOW-
CLASS)>
? (defvar *w* (make-instance 'my-window-class))
*W*
? (set-window-zoom-position *w* #@(20 60))
3932180
```

window-zoom-size [*Generic function*]

Syntax `window-zoom-size` (*window* `window`)

Description The `window-zoom-size` generic function returns the zoom size of a window, that is, its size after the user clicks the zoom box. This value is either the last value given to `set-window-zoom-size` for *window* or the value returned by calling `window-default-zoom-size` on *window.*

Argument *window* A window.

window-default-zoom-size [*Generic function*]

Syntax `window-default-zoom-size` (*window* `window`)

Description The generic function `window-default-zoom-size` determines the default zoom size of a window, that is, its new size after the user clicks the zoom box. The provided method returns the value of `*window-default-zoom-size*`.

Argument *window* A window.

window-default-zoom-size [*Variable*]

Description The `*window-default-zoom-size*` variable stores the default zoom size of a window, that is, its new size after the user clicks the zoom box.

This variable and `*window-default-zoom-position*` are initialized at startup to make a zoomed window fill the screen containing the menu bar with a 3-pixel border all around.

set-window-zoom-size [*Generic function*]

Syntax `set-window-zoom-size` (*window* `window`) *h* `&optional` *v*

Description The generic function `set-window-zoom-size` sets the zoom size of a window, that is, its new size after the user clicks the zoom box, and returns the new size, encoded as an integer.

Arguments *window* A window.

h	The horizontal coordinate of the new position, or the complete position (encoded as an integer) if *v* is `nil` or not supplied.
v	The vertical coordinate of the new position, or `nil` if the complete position is given by *h*.

`window-grow-rect` [*Generic function*]

Syntax `window-grow-rect` (*window* `window`)

Description The `window-grow-rect` generic function returns a rectangle record whose upper-left and lower-right components determine the minimum and maximum sizes to which *window* can be resized with the mouse.

The window can still assume other sizes when the user clicks the zoom box, and other sizes can be set with the function `set-view-size`.

Argument *window* A window.

Example

```
(let ((r (window-grow-rect (target))))
  (format t "(~S, ~S)  (~S, ~S)"
          (pref r :rect.top)
          (pref r :rect.left)
          (pref r :rect.bottom)
          (pref r :rect.right)))
```

`window-drag-rect` [*Generic function*]

Syntax `window-drag-rect` (*window* `window`)

Description The `window-drag-rect` generic function returns a rectangle record whose value constrains how *window* can be moved with the mouse. Whenever the pointer moves outside this rectangle, the gray window outline disappears, indicating that the user is out of bounds.

Argument *window* A window.

`view-cursor` [*Generic function*]

Syntax `view-cursor` (*view* `simple-view`) *point*
`view-cursor` (*window* `window`) *point*

Description The `view-cursor` generic function returns the cursor shape to display when the mouse is at *point*, a point in *view*. It is called by `window-update-cursor` as part of the default `window-null-event-handler`.

Specialize the `view-cursor` generic function to change your view's cursor to one of the following predefined cursors or to a user-defined cursor.

`*arrow-cursor*`
: The standard north-northwest arrow cursor.

`*i-beam-cursor*`
: The I-beam shape used when the cursor is over an area of editable text.

`*watch-cursor*`
: The watch-face shape shown during time-consuming operations, when event processing is disabled.

Arguments

view	A simple view.
window	A window.
point	The position of the cursor, expressed as a point.

`window-cursor` [*Generic function*]

Syntax `window-cursor` (*window* `window`)

Description The `window-cursor` generic function returns the current cursor of *window*. The system-supplied `view-cursor` method for `window` calls `window-cursor` to determine the cursor of *window.*

Argument *window* A window.

`window-object` [*Function*]

Syntax `window-object` *wptr*

Description The `window-object` function returns the window object pointed at by *wptr*.

Argument *wptr* A macptr to a window record..

Example

```
? (window-object (wptr (target)))
#<FRED-WINDOW "New" #x454909>
```

`with-port` [*Macro*]

Syntax `with-port` *grafport* {*form*}*

Description The `with-port` macro executes *form* with *grafport* as the current GrafPort. Upon exit, the previously current GrafPort is restored. The *form* is executed within the special form `without-interrupts`.

This macro is a very low-level way of binding the QuickDraw GrafPort. It is not recommended for general use; instead use `with-focused-view`.

Arguments

grafport	A GrafPort, usually the `wptr` of a window.
form	Zero or more Lisp forms to be executed with the GrafPort set to *grafport*. These usually include direct calls to QuickDraw routines.

Supporting standard menu items

Many of the menu items in the default MCL menu bar operate on the top window. These menu items are instances of the class `window-menu-item`. (See Chapter 3: Menus.) These commands can work in any window if the class of the window has an appropriate method.

The menu items and their corresponding functions are as follows:

Close	`window-close`
Save	`window-save`
Save As…	`window-save-as`
Save Copy As…	`window-save-copy-as`
Revert	`window-revert`
Print…	`window-hardcopy`
Undo	`undo`
Undo More	`undo-more`
Cut	`cut`
Copy	`copy`
Paste	`paste`
Clear	`clear`
Select All	`select-all`
Execute Selection	`window-eval-selection`
Execute Buffer	`window-eval-buffer`
List Definitions	`window-defs-dialog`

If the class of the active window has a method definition for one of these functions, then the corresponding menu item is enabled. If the user chooses the menu item, the function is called on the active window. Enabling of items on the Edit menu is controlled by the generic function `window-can-do-operation`, described later in this section.

---

window-needs-saving-p [*Generic function*]

Syntax `window-needs-saving-p` (*window* `window`)

Description The `window-needs-saving-p` generic function determines whether the Save menu item in the File menu should be enabled for windows that have a definition of `window-save`.

The Save menu item is enabled if the class of the active window has a method definition for `window-save`, unless the window has a method definition for `window-needs-saving-p` and a call to `window-needs-saving-p` returns `nil`. If the window has a method definition for `window-needs-saving-p`, then Save is enabled only if a call to `window-needs-saving-p` returns true.

Argument *window* A window.

---

window-can-do-operation [*Generic function*]

Syntax `window-can-do-operation` (*view* `fred-mixin`) *operation* `&optional` *menu-item*

`window-can-do-operation` (*window* `window`) *operation* `&optional` *menu-item*

Description The `window-can-do-operation` generic function returns a Boolean value indicating whether *view* can perform *operation.* (This is a more general replacement for the older MCL function `window-can-undo-p`, which could check only for Undo.) If the value returned is true, the menu item for *operation* is enabled; otherwise, it is disabled.

The `window-can-do-operation` method for `window` returns `t` if there is a method for *operation* defined for the class of *window* that is more specific than the built-in method defined for the class `window`. Otherwise `window-can-do-operation` returns the result of calling `window-can-do-operation` on the current key handler of *window*, if there is one. If not, it returns `nil`.

The method for `fred-mixin` returns `t` if the operation is meaningful for the current state of the Fred window or Fred dialog item.

Arguments

view	A Fred window or Fred dialog item.
window	A window.

operation	A symbol specifying one of the standard editing operations: `cut`, `clear`, `copy`, `paste`, `select-all`, `undo`, or `undo-more`.
menu-item	The corresponding Edit menu item.

Example

```
? (window-can-do-operation *top-listener* 'paste)
T
```

Floating windows

Floating windows are a subclass of windows that appear frontmost on the screen. (That is, they always "float to the top.") Floating windows are generally used for creating tool palettes.

Floating windows respond to clicks, handle `activate` and `deactivate` events, and respond to keystroke events. See the file `windoid-key-events.lisp` in the MCL Examples folder for commented sample code.

These expressions are used in defining and counting floating windows.

windoid [*Class name*]

Description The class `windoid` is the class of floating windows, built on `window`. Floating windows may be mixed in with other window classes, such as dialog boxes. In this case, `windoid` should appear first in the inheritance path.

initialize-instance [*Generic function*]

Syntax `initialize-instance` (*windoid* `windoid`) `&rest` *initargs*

Description The `initialize-instance` primary method for `windoid` initializes a floating window.

Arguments

windoid	A floating window.
initargs	A list of keywords and values used to initialize the floating window. No special keywords are used. The following keywords have default values:

`:view-size`

The default value of the size of the floating window is `#@(115 150)`.

`:window-do-first-click`

The value of this initialization argument determines whether the click that selects a window is also passed to `view-click-event-handler`. For all floating windows, the default value of this variable is true.

The click that selects an application in Multifinder is not passed to the application unless either the clicked window is not the front window or the Get Front Clicks bit is set in the application's size resource.

`*windoid-count*` [*Variable*]

Description The `*windoid-count*` variable contains the number of visible floating windows currently in the MCL environment.

Chapter 5:

Dialog Items and Dialogs

Contents

This chapter describes the dialog functionality and the built-in dialog item classes in Macintosh Common Lisp.

The dialog functionality is very flexible in Macintosh Common Lisp. Dialog items display information and may initiate an action when clicked by the user. In Macintosh Common Lisp, dialog items can appear in any window. They are built from the class `dialog-item`, which is not used directly; you specialize it and use the subclasses. In turn, `dialog-item` is built from the class `simple-view`, since dialog items have no subviews. Built-in subclasses of `dialog-item` include radio buttons, checkboxes, and editable-text fields, as well as pop-up menus, scroll bars, and tables in dialog boxes.

You should read this chapter if you are programming specialized types of dialog items.

Before reading this chapter, you should be familiar with the MCL implementation of views and windows, described in Chapter 5, “Views and Windows.” The subclass of `dialog-item` that supports editable text is `fred-dialog-item`, documented in Chapter 14, “Programming the Editor.”

Dialogs in Macintosh Common Lisp

In the standard Macintosh interface, actions are performed by dialog items within dialog boxes. Macintosh Common Lisp supports this functionality and makes it more generalized.

Dialog items

Instead of setting up a specialized class for dialog boxes and alerts, Macintosh Common Lisp defines any structured communication as simply a collection of dialog items in a window. You can add dialog items to any view or window, or you can write specialized classes based on `window`, in which dialog items may appear.

Therefore, for creating dialog functionality the important class is `dialog-item`.

Built-in MCL dialog items include buttons, radio buttons, checkboxes, tables, editable text, scroll bars, pop-up menus, and static text. They are discussed in “Dialog items” on page 185.

In addition, the sample files in the Examples and Library folders contain code for kinds of dialog items. Of course, you can also define your own classes of dialog items.

Dialog boxes

Dialog boxes initiate and control well-defined actions in a structured way. You use them whenever you want the user to do something complex in which the range of response is predictable or needs to be controlled. The Print Options dialog box is a good example; it includes text fields, which the user fills in, and a defined set of choices that are represented by radio buttons and checkboxes.

Alerts query an action, displaying a message such as “Are you sure you want to reformat your hard disk?” They request the user to confirm explicitly before proceeding, or to cancel.

MCL dialogs are unspecialized subclasses of `window`, provided for backward compatibility with earlier versions of Macintosh Common Lisp. They have methods for all `window` and `view` operations, but no methods of their own. Display, for instance, works the same way in dialogs as in all other windows. Dialogs are only one of the places you can use dialog items.

Macintosh Common Lisp provides four predefined standard dialog boxes for alerts and user responses, discussed in "Simple turnkey dialog boxes" on page 239.

You can write standard Macintosh dialog boxes quite easily, while the same functionality also adapts well to other uses. For example, you can create a hypertext system that includes text, graphics, and dialog items, or an interactive forms manager, or a spreadsheet, all using largely the same code.

A simple way to design dialogs and program dialog items

MCL contains a dialog design tool, part of the Interface Toolkit, that works like a simple paint system. You can choose buttons and fields from a palette and move them into a new dialog. You can set their default states and actions. This tool is supplied as source code so it can be customized; you'll find it in the Interface Tools folder supplied with your copy of Macintosh Common Lisp. Its operation is described in Chapter 7: The Interface Toolkit.

Changes to dialogs in Macintosh Common Lisp as of version 2

If you have used an earlier version of Macintosh Common Lisp, you will find that the implementation of dialogs has changed substantially, making them more flexible to use and easier to program.

- The `dialog` class, which is a subclass of `window`, exists only for compatibility. No methods are specialized on it and it adds no slots.
- Dialog items may now be added to all views.
- Some functions have changed to reflect the new definition of `dialog`.
- All new functions are CLOS generic functions.

The file `old-dialog-hooks.lisp`, distributed in the Examples folder that is part of Macintosh Common Lisp, contains code defining the old dialog functions in terms of the new ones. You should find it quite easy, however, to port your old dialog code to Macintosh Common Lisp version 2.

Table 5-1 summarizes the functions that have changed.

■ **Table 5-1** Summary of changed dialog functions in Macintosh Common Lisp

Old	**New**
`add-dialog-items`	`add-subviews`
`add-self-to-dialog`	`install-view-in-window`
`buffer-char-font`	`buffer-char-font-spec`
`buffer-replace-font`	`buffer-replace-font-spec`
`buffer-set-font`	`buffer-set-font-spec`
`catch-abort`	`use restart-case`
`catch-error`	`use handler-case`
`catch-error-quietly`	`ignore-errors`
`color-window-mixin`	`:color-p initialization argument`
`:dialog-item-colors`	`:part-color-list intialization argument`
`dialog-item-default-size`	`view-default-size`
`dialog-item-dialog`	`view-container`
`(set-)dialog-item-font`	`(set-)view-font`
`dialog-item-nick-name`	`view-nick-name`
`(set-)dialog-item-size`	`(set-)view-size`
`(set-)dialog-item-position`	`(set-)view-position`
`ed-skip-fwd-wsp&comments`	`buffer-skip-fwd-wsp&comments`
`find-named-dialog-items`	`view-named, find-named-sibling`
`item-named`	`view-named`
`markp`	`buffer-mark-p`
`:parent` keyword to windows, etc.	`:class` keyword
`remove-dialog-items`	`remove-subviews`
`remove-self-from-dialog`	`remove-view-from-window`
`window-(de)activate-event-handler`	`view-(de)activate-event-handler`
`window-buffer`	`fred-buffer`

`window-click-event-handler`	`view-click-event-handler`
`window-font`	`view-font`
`window-hpos`	`fred-hpos`
`window-line-vpos`	`fred-line-vpos`
`window-mouse-position`	`view-mouse-position`
`(set-)window-position`	`(set-)view-position`
`(set-)window-size`	`(set-)view-size`
`window-start-mark`	`fred-display-start-mark`
`window-update`	`fred-update`
`window-vpos`	`fred-vpos`

Dialog items

Dialog items do two things: appear within a view, and perform actions. Generally speaking, a dialog item inherits its display behavior from `simple-view` or from its class; its default methods are also determined at the class level. You can add specific action at the instance level.

The base class from which all other dialog items inherit is `dialog-item`. This class is not meant to be instantiated directly. Instead, it is the superclass from which more specific classes of dialog items are built.

The dialog item subclasses provided by Macintosh Common Lisp are

`button-dialog-item`

`check-box-dialog-item`

`editable-text-dialog-item`

`fred-dialog-item`

`radio-button-dialog-item`

`sequence-dialog-item` (a subclass of `table-dialog-item`)

`static–text-dialog–item`

`table-dialog-item`

The class `fred-dialog-item` is discussed in Chapter 14: Programming the Editor. The others are discussed in this chapter.

In addition, you can use sample files in your MCL Examples and Library folders to make several other kinds of dialog items, including scroll bars, icons, and pop-up menus, and of course you can create your own subclasses of `dialog-item`.

MCL forms relating to dialog items

The following MCL expressions are used in creating dialog items.

dialog-item [*Class name*]]

Description The class `dialog-item` provides the basic functionality shared by all dialog items. It is built on `simple-view`.

initialize-instance [*Generic function*]

Syntax `initialize-instance` (*dialog-item* `dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `dialog-item` initializes a dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

dialog-item A dialog item.

initargs A list of keywords and default values used to initialize a dialog item. The *initargs* keywords for all dialog items are as follows:

`:view-size`
The size of the dialog item. If not specified or `nil`, this value is calculated so that the item's `dialog-item-text` is visible. If the specified value is too small, the item is clipped when it is drawn. The default value is `nil`.

`:view-container`
The dialog box or other view that contains the item.

`:view-position`
: The position in the dialog box where the item will be placed, in the coordinate system of its container. If this argument is not specified or is specified as `nil`, the first available position large enough to hold the item is used. If no space is large enough, the dialog item is placed in the upper-left corner of the dialog.

`:view-nick-name`
: The nickname of the dialog item. This feature is used in conjunction with `view-named`. The default value is `nil`.

`:view-font`
: The font in which the text of the dialog item appears. If `nil`, the window font is used.

`:dialog-item-text`
: The text of the dialog item. The initial value is `nil`.

`:dialog-item-handle`
: For advanced programmers, this option specifies the handle of the dialog item. See the description of the `dialog-item-handle` generic function on page 199. This option is used only for creating specialized subclasses of dialog items. The handle is usually allocated by the `install-view-in-window` method. Its initial value is `nil`.

`:dialog-item-enabled-p`
: The state (enabled or disabled) of the item. Disabled items are dimmed, and their actions are not run when the user clicks them.

`:part-color-list`
: A property list of colors to which the parts of the dialog item should be set. The four possible keywords are `:frame`, the outline of the dialog item; `:text`, its text; `:body`, its body; and `:thumb`, its scroll box. (The scroll box is the white box that slides inside the scroll bar; scroll bars are the only dialog items that can have them.)

`:dialog-item-action`
: The action run when the dialog item is selected. The value of this keyword should be a function or a symbol with a global function definition. It is called with a single parameter, the dialog item.

`:help-spec`
: A value describing the Balloon Help for the item. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

`:wptr` — A pointer to a window record on the Macintosh heap. This record can be examined or passed to Macintosh traps that take a window pointer. The value is `nil` if the item is not contained in a window.

`dialog-items` [*Generic function*]

Syntax `dialog-items` (*view* `view`) `&optional` *item-class must-be-enabled*

Description The `dialog-items` generic function returns a list of the dialog items in *view*.

Arguments

view — A view.

item-class — If the value of *item-class* is specified and non-`nil`, then only dialog items matching *item-class* are returned. The default value is `nil`.

must-be-enabled — If the value of *must-be-enabled* is true, then only dialog items that are enabled are returned. The default value is `nil`.

`make-dialog-item` [*Function*]

Syntax `make-dialog-item` *class position size text* `&optional` *action* `&rest` *attributes*

Description The `make-dialog-item` function creates a dialog item using `make-instance`.

Arguments

class — The class of the dialog item.

position — The position of the dialog item with respect to its container.

size — The size of the dialog item.

text — The text included within the dialog item.

action — The action associated with the dialog item.

attributes — One or more attributes belonging to the dialog item. The number and nature of these depend on the type of dialog item.

Example

This function could be defined as follows:

```
? (defun make-dialog-item (class position size text
                           &optional action &rest attributes)
       (apply #'make-instance class
              :view-position position
```

```
:view-size size
:dialog-item-text text
:dialog-item-action action
attributes))
```

dialog-item-action [*Generic function*]

Syntax `dialog-item-action` (*item* `dialog-item`)

Description The generic function `dialog-item-action` is called whenever the user clicks a dialog item. The method for `dialog-item` calls *item*'s `dialog-item-action-function`, if it is not `nil`. Otherwise, it does nothing.

The `dialog-item-action` function is normally called when the mouse button is released, not when it is pressed.

If an item is disabled, its action is not run.

Since `dialog-item-action` is usually called by `view-click-event-handler` as a result of event processing, event processing is ordinarily disabled while the `dialog-item-action` function is running. This means that other dialog items cannot be selected during the action. To avoid locking out other event processing, you can use `eval-enqueue` to insert forms into the read-eval-print loop. For details, see Chapter 10: Events.

Argument *item* A dialog item.

dialog-item-action-function [*Generic function*]

Syntax `dialog-item-action-function` (*item* `dialog-item`)

Description The generic function `dialog-item-action-function` returns the value set by the `:dialog-item-action` initialization argument or the `set-dialog-item-action-function` generic function. Unless it is `nil`, this function is called with a single argument, *item,* by the `dialog-item-action` method for `dialog-item`.

This generic function is called by the `view-click-event-handler` method for `dialog-item` when the user clicks a dialog item.

Argument *item* A dialog item.

`set-dialog-item-action-function` [*Generic function*]

Syntax `set-dialog-item-action-function` (*item* `dialog-item`)
new-function

Description The generic function `dialog-item-action-function` sets the value it accesses.

Arguments

item	A dialog item.
new-function	A function of one argument or a symbol that has a global function binding or is `nil`.

`view-click-event-handler` [*Generic function*]

Syntax `view-click-event-handler` (*item* `dialog-item`) *where*

Description The generic function `view-click-event-handler` is called by the event system when the user clicks the dialog item. The method for `dialog-item` calls *item*'s `dialog-item-action-function` with *item* as the single argument. If *item*'s `dialog-item-action-function` is `nil`, nothing is done.

Arguments

item	A dialog item.
where	The cursor position. It is ignored.

`view-focus-and-draw-contents` [*Generic function*]

Syntax `view-focus-and-draw-contents` (*item* `dialog-item`)
`&optional` *visrgn cliprgn*

Description The method for dialog items of the generic function `view-focus-and-draw-contents` focuses on the container of the dialog item, then calls `view-draw-contents.`

Arguments

item	A dialog item.
visrgn, *cliprgn*	Region records from the view's `wptr`. They are ignored.

`view-size` [*Generic function*]

Syntax `view-size` (*item* `dialog-item`)

Description The method for dialog items of the generic function `view-size` returns the size of the dialog item as a point.

Argument *item* A dialog item.

---

set-view-size [*Generic function*]

Syntax `set-view-size` (*item* `dialog-item`) *h* `&optional` *v*

Description The method for dialog items of the generic function `set-view-size` changes the size of the dialog item to the width and height represented by *h* and *v*, and returns the new size.

Arguments

item A dialog item.

h Horizontal position.

v Vertical position. If *v* is `nil`, *h* is assumed to represent a point.

Example

```
? (add-subviews my-window
    (setf eddie
      (make-instance 'editable-text-dialog-item)))
NIL
? (point-string (view-size eddie))
"#@(6 17)"
? (set-view-size eddie #@(300 20))
1311020
```

---

dialog-item-text [*Generic function*]

Syntax `dialog-item-text` (*item* `dialog-item`)

Description The generic function `dialog-item-text` returns the string of text associated with the dialog item.

Argument *item* A dialog item.

---

set-dialog-item-text [*Generic function*]

Syntax `set-dialog-item-text` (*item* `dialog-item`) *text*

Description The generic function `set-dialog-item-text` sets the text associated with the dialog item to *text* and returns *text*.

The text of a dialog item has a different meaning for each class of dialog item. It is the text of static-text and editable-dialog text items. It is the label displayed inside buttons and to the right of radio buttons and checkboxes.

If you prefer to put text in a different location, set the text to the empty string and use a separate static-text item to place the text where you would like it.

Tables do not display their dialog item text.

Arguments *item* A dialog item.

text A string to be used as the new text of the dialog item.

`view-font` [*Generic function*]

Syntax `view-font` (*item* `dialog-item`)

Description The generic function `view-font` returns, as a font spec, the font used by *item*, or `nil` if *item* does not have its own font. (If *item* does not have its own font, it uses its container's font.)

Argument *item* A dialog item.

`set-view-font` [*Generic function*]

Syntax `set-view-font` (*item* `dialog-item`) *new-font*

Description The generic function `set-view-font` sets the font of the dialog item to *new-font*.

Arguments *item* A dialog item.

new-font A font specifier. If `nil`, the dialog item uses the font of its window.

`view-font-codes` [*Generic function*]

Syntax `view-font-codes` (*item* `dialog-item`)

Description The generic function `view-font-codes` returns two values, the font-face code and mode-size code for *item*'s font. (Font codes, an efficient way of encoding font specs, are described in *Inside Macintosh* and in "Implementation of font codes" on page 75)

Argument *item* A dialog item.

`set-view-font-codes` [*Generic function*]

Syntax `set-view-font-codes` (*item* `dialog-item`) *ff ms* `&optional` *ff-mask ms-mask*

Description The generic function `set-view-font-codes` changes the view font codes of *item*.

Arguments

item	A dialog item.
ff	The font/face code. A font/face code is a 32-bit integer that stores the encoded name of the font and its face (plain, bold, italic, and so on).
ms	The mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size.
ff-mask	A mask that instructs `set-view-font-codes` to look only at certain bits of the font/face integer.
ms-mask	A mask that instructs `set-view-font-codes` to look only at certain bits of the mode/size integer.

`part-color` [*Generic function*]

Syntax `part-color` (*item* `dialog-item`) *part*

Description The generic function `part-color` returns the color of the part indicated by *part*.

Arguments

item	A dialog item.
part	A keyword specifying which part of the dialog item should be set. The four possible keywords are `:frame`, the outline of the dialog item; `:text`, its text; `:body`, its body; and `:thumb`, its scroll box. (The scroll box is the white box that slides inside the scroll bar; scroll bars are the only dialog items that can have them.)

`set-part-color` [*Generic function*]

Syntax `set-part-color` (*item* `dialog-item`) *part new-color*

Description The generic function `set-part-color` sets the color of part of the dialog item, as specified by the arguments, and returns *new-color*.

If you create a new class of dialog items, you may want to define `view-draw-contents` to pay attention to these values.

In addition to the keywords specified by *part*, individual cells in table dialog items can have colors. See the method on `set-part-color` for `table-dialog-item`.

Arguments	*item*	A dialog item.
	part	A keyword specifying a part of the dialog item. The same keywords are allowed as for `part-color`.
	new-color	A color, encoded as an integer.

`part-color-list` [*Generic function*]

Syntax `part-color-list` (*item* `dialog-item`)

Description The generic function `part-color-list` returns a list of part keywords and colors for all the colored components of the dialog item. Components whose color has not been set are not included.

Argument *item* A dialog item.

`dialog-item-enable` [*Generic function*]

Syntax `dialog-item-enable` (*item* `dialog-item`)

Description The generic function `dialog-item-enable` enables the dialog item. The item is not dimmed, and its action is run when the user clicks it. The function returns `nil`.

Argument *item* A dialog item.

`dialog-item-disable` [*Generic function*]

Syntax `dialog-item-disable` (*item* `dialog-item`)

Description The generic function `dialog-item-disable` disables the dialog item. The dialog item is dimmed; clicks in the item are ignored, and the action of the item is never run. Disabling a checkbox does not alter its status as checked, and disabling a radio button does not alter its status as clicked (you may want to remove the check or click explicitly). The function returns `nil`.

Argument *item* A dialog item.

`dialog-item-enabled-p` [*Generic function*]

Syntax `dialog-item-enabled-p` (*item* `dialog-item`)

Description The generic function `dialog-item-enabled-p` returns `t` if the dialog item is enabled, and `nil` if it is disabled.

Argument *item* A dialog item.

Advanced dialog item functions

The following functions can be defined for user-created classes of dialog items. They can also be shadowed in specialized classes of the predefined dialog items. For general use of dialog items, you do not need to use the following functions.

Several sample files demonstrate the use of dialog items. In your MCL Examples folder, `text-edit-dialog-item.lisp` shows how to implement dialog items if you do not want to make Fred a part of your implementation. In the Library folder, `graphic-items.lisp`, `scroll-bar-dialog-items.lisp`, and `scrolling-fred-dialog-item.lisp` implement several specialized types of dialog items.

`install-view-in-window` [*Generic function*]

Syntax `install-view-in-window` (*item* `dialog-item`) *window*

Description The generic function `install-view-in-window` is called by `set-view-container` when an item becomes part of a view.

This function performs initialization tasks that require the containing window. It should never be called directly by user code. However, it may be shadowed. Specialized versions of `install-view-in-window` should always perform `call-next-method`.

The default method sets the size of the dialog item if it does not already have one, and finds an empty position for the dialog item if it does not already have a position.

Arguments *item* A dialog item.

window The window to which the dialog item is being added.

`remove-view-from-window` [*Generic function*]

Syntax `remove-view-from-window` (*item* `dialog-item`)

Description The generic function `remove-view-from-window` is called when a dialog item is removed from a view by a call to `set-view-container`. It should never be called directly by user code. However, it may be shadowed. Specialized versions of `remove-view-from-window` should dispose of any Macintosh data the item uses (that is, data not subject to garbage collection) and should always perform a `call-next-method.`

Argument *item* A dialog item.

`dialog-item-handle` [*Generic function*]

Syntax `dialog-item-handle` (*item* `dialog-item`)

Description The generic function `dialog-item-handle` retrieves the handle associated with *item*. Dialog items are often associated with handles to Macintosh data structures, such as control records. By convention, this handle is stored in the location referenced by `dialog-item-handle` and modified by `set-dialog-item-handle`. The handle is usually `nil` when the dialog item is not contained in a window. It is generally set by `install-view-in-window` and is reset to `nil` by `remove-view-from-window`.

Argument *item* A dialog item.

`set-dialog-item-handle` [*Generic function*]

Syntax `set-dialog-item-handle` (*item* `dialog-item`) *handle*

Description The generic function `set-dialog-item-handle` sets the dialog item handle associated with *item* to a new handle.

Arguments *item* A dialog item.

handle A handle to a Macintosh data structure.

`view-activate-event-handler` [*Generic function*]

Syntax `view-activate-event-handler :around` (*item* `dialog-item`)

Description The generic function `view-activate-event-handler` is called when the window containing the dialog item is activated.

If the appearance of the dialog item needs to change to indicate that it is active, this is the method that should make that change. For example, Fred dialog items change their highlighting from a pixelwide box to a solid rectangle and scroll bars make their arrows and scroll box visible.

The `view-activate-event-handler` generic function is called by `set-view-container` if the window in which the newly installed view appears is active.

Argument *item* A dialog item.

`view-deactivate-event-handler` [*Generic function*]

Syntax `view-deactivate-event-handler` (*item* `dialog-item`)

Description The generic function `view-deactivate-event-handler` is called when the window containing the dialog items is deactivated.

If the appearance of the dialog item needs to change to indicate that it is not active, this is the method that should make that change. For example, Fred dialog items change their highlighting from a solid rectangle to a 1-pixel-wide box and scroll bars become an empty rectangle.

The `view-deactivate-event-handler` generic function is called by `set-view-container` if the window in which the newly installed view appears is not active.

Argument *item* A dialog item.

`view-default-size` [*Generic function*]

Syntax `view-default-size` (*item* `dialog-item`)

Description The generic function `view-default-size` is called by the default version of `install-view-in-window`. It is called for dialog items that are not given an explicit size. The `dialog-item` method of `view-default-size` calculates a size according to the font and text of the dialog item and the width correction associated with the class of the dialog item. (See the documentation of `dialog-item-width-correction`.)

Argument *item* A dialog item.

---

dialog-item-width-correction [*Generic function*]

Syntax `dialog-item-width-correction` (*item* `dialog-item`)

Description The generic function `dialog-item-width-correction` returns an integer representing the number of pixels of white space added to the left and right of the text of a dialog item. The default method for `dialog-item` returns `0`. Users can write methods for `dialog-item-width-correction` if they wish to specialize it for their own classes of dialog items.

Argument *item* A dialog item.

---

with-focused-dialog-item [*Macro*]

Syntax `with-focused-dialog-item (`*item* `&optional` *container*`)`
`&body` *body*

Description The macro `with-focused-dialog-item` executes *body* with the drawing environment set up in the coordinate system of *container* and the font of *item*. This is the correct environment for calling `view-draw-contents` on a dialog item. When the body exits (normally or abnormally), the old drawing environment is restored.

Arguments

item	A dialog item (or any simple view).
container	The view focused on whose coordinate system *body* will run.
body	Forms to be executed with the specified drawing environment.

Examples

The macro `with-font-focused-view` could be defined as follows.

```
(defmacro with-font-focused-view (view &body body)
  (let ((view-sym (gensym)))
    `(let ((,view-sym ,view))
       (with-focused-dialog-item (view view) ,@body))))
```

The macro `view-focus-and-draw-contents` for `dialog-item` could be defined as follows.

```
(defmethod view-focus-and-draw-contents ((item dialog-item)
&optional
                                          visrgn cliprgn)
  (declare (ignore visrgn cliprgn))
  (with-focused-dialog-item item
    (view-draw-contents item)))
```

Specialized dialog items

Button, static text, editable text, checkboxes, radio button, tables, sequences, and user-defined dialog items fall into the category of specialized dialog items.

The initialization argument keywords documented for the `dialog-item` class are applicable to all dialog items. Only the additional keywords that are specific to each specialized dialog item are documented in the following sections.

Buttons

Button dialog items are rounded rectangles that contain text. The following MCL expressions operate on button dialog items.

`button-dialog-item` [*Class name*]

Description This is the class used to make buttons. Clicking a button usually has an immediate result. Buttons are generally given a function for `dialog-item-action-function` via the `:dialog-item-action` initialization argument.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*item* `button-dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `button-dialog-item` initializes a button dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item — A button dialog item.

initargs — A list of keywords and values used to initialize the button. These are its special *initargs* keywords (in addition to those for `dialog-item`):

`:default-button`
: An argument specifying whether the button is made the default button. If this value is `nil` (the default), the button is not made the default button. Note that if the dialog has a default button and `:allow-returns` is true for the current key handler, then the Return key will be handled by the key handler rather than the default button.

`:border-p`
: An argument specifying whether the button has a border. If this value is true (the default), the button has a border.

Example

```
? (setq pearl (make-instance 'button-dialog-item
                             :default-button t))
#<BUTTON-DIALOG-ITEM #x42C699>
```

---

press-button [*Generic function*]

Syntax `press-button` (*button* `button-dialog-item`)

Description The `press-button` generic function highlights *button*, then calls the `dialog-item-action` method for *button.*

Argument *button* — A button dialog item.

Default buttons

Default buttons are a convenient subclass of button dialog items; they serve as the default button. A dialog may have one default button. This button has a bold border and usually may be selected by one of the keystrokes Return or Enter.

The following MCL expressions operate on default-button dialog items.

`default-button-dialog-item` [*Class name*]

Description The `default-button-dialog-item` class is the class of default buttons, a subclass of `button-dialog-item`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*item* `default-button-dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `default-button-dialog-item` initializes a default-button dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item A default-button dialog item.

initargs A list of keywords and values used to initialize the button. This class has no additional *initargs* keywords, but has two default values:

`:dialog-item-text`
The default value of this initialization argument is `"OK"`.

`:default-button`
The default value of this initialization argument is true.

`default-button` [*Generic function*]

Syntax `default-button` (*window* `window`)

Description The `default-button` generic function returns the current default button, or `nil` if the window has no default button. The default button is the button whose action is run when the user presses Return or Enter. It is outlined with a heavy black border.

If carriage returns are allowed in the current editable-text item, they are sent to that item rather than to the default button.

Argument *window* A window.

`set-default-button` [*Generic function*]

Syntax `set-default-button` (*window* `window`) *new-button*

Description The `set-default-button` generic function changes the default button according to the value of *new-button* and returns *new-button*.

If carriage returns are allowed in the current editable-text item, they are sent to that item rather than to the default button.

Arguments

window	A window.
new-button	The button that should be made the default button, or `nil`, indicating that there should be no default button.

`default-button-p` [*Generic function*]

Syntax `default-button-p` (*item* `button-dialog-item`)

Description The `default-button-p` generic function returns true if *item* is the default button in the `view-window` of *item*. Otherwise it returns `nil`.

Argument *item* A button dialog item.

Static text

The next two entries define and initialize the class of static-text dialog items.

`static-text-dialog-item` [*Class name*]

Description This is the class of static-text dialog items. Static text may be positioned anywhere in a dialog window to supply additional information to the user. The text appears in the window's font unless otherwise specified. Clicking text does not generally initiate an action, but it may.

Depending on the amount of text and the size of the item, the text may wrap to fit in its area. If the size is not specified, a size that accommodates the text without wrapping is used.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*item* `static-text-dialog-item`)
`&rest` *initargs*

Description The `initialize-instance` primary method for `static-text-dialog-item` initializes a static-text dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item	A static-text dialog item.
initargs	A list of keywords and values used to initialize the static-text dialog item. The subclass `static-text-dialog-item` does not have any additional keyword arguments beyond those for `dialog-item`.

Editable text

The following entries pertain to the class of editable-text dialog items.

`editable-text-dialog-item` [*Class name*]

Description This is the class of editable-text dialog items, a subclass of `fred-dialog-item`. Its superclasses include `fred-mixin` and `key-handler-mixin`. The class adds no new initialization arguments, and there is only one method specialized on the class, `view-default-font`.

The user can give standard Macintosh commands to edit the text of such items. For instance, the user can select, cut, copy, and paste the text of editable-text dialog items.

Editable text is usually surrounded by a box, although this feature may be disabled.

At any given time, there is only one current editable-text dialog item. This is the item with a blinking cursor or a highlighted selection. User typing is directed to this item by a call to `view-key-event-handler`. Pressing the Tab key makes the next editable-text dialog item current, cycling back to the first after the last. The current editable-text dialog item can be determined by calling `current-key-handler` and can be changed by calling `set-current-key-handler`.

The text of an `editable-text-dialog-item` can be accessed by calling `dialog-item-text` and changed by calling `set-dialog-item-text`. When an editable text item is created, the initial text is specified using the `:dialog-item-text` initialization argument.

To refer unambiguously to an editable-text dialog item, you can give it a nickname.

Tip The file `text-edit-dialog-item.lisp`, in your Examples folder, provides an implementation of the class `editable-text-dialog-item` using the Macintosh TextEdit Manager. If your application does not require full Fred editing capability in editable text, you may wish to use `text-edit-dialog-item` instead of `editable-text-dialog-item`. Most of the built-in MCL dialogs containing editable text items instantiate these items as `editable-text-dialog-item` rather than as `fred-dialog-item`. If your application needs to use built-in dialogs but does not need Fred editing capability within those dialogs, you can redefine the class `editable-text-dialog-item` to be a subclass of `text-edit-dialog-item`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*item* `editable-text-dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `editable-text-dialog-item` initializes an editable-text dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item An editable-text dialog item.

initargs A list of keywords and values used to initialize the editable-text dialog item. It has no new initialization arguments beyond those it inherits from `fred-dialog-item`.

`view-key-event-handler` [*Generic function*]

Syntax `view-key-event-handler` (*item* `fred-mixin`) *char*

Description The generic function `view-key-event-handler` examines the current keystroke and determines what is to be done with it.

The method for `fred-mixin` binds the `*current-keystroke*` variable to the keystroke of the current event and runs the Fred command associated with the keystroke.

Arguments

item An editable-text dialog item.

char Any keystroke. If *char* is a carriage return, this function is called only if `allow-returns-p` is true for the item.

`key-handler-mixin` [*Class name*]

Description The class `key-handler-mixin` should be mixed into any class that handles key events. The class `fred-dialog-item` includes `key-handler-mixin`.

`key-handler-p` [*Generic function*]

Syntax `key-handler-p` (*item* `dialog-item`)
`key-handler-p` (*key-handler* `key-handler-mixin`)

Description The `key-handler-p` generic function checks to see whether *item* is a key handler. When `key-handler-p` is called on an instance of a class one of whose superclasses is `key-handler-mixin`, the function returns `t` unless the key handler is disabled. The method for `dialog-item` returns `nil`.

Arguments

item	A dialog item.
key-handler	An object one of whose superclasses is `key-handler-mixin`.

`exit-key-handler` [*Generic function*]

Syntax `exit-key-handler` (*item* `key-handler-mixin`) *new-text-item*

Description The generic function `exit-key-handler` is called when an editable-text dialog item that is the current key handler is about to be exited. At this point, it is still the current key handler, but soon it won't be. If the function returns `t` (as the method for `key-handler-mixin` does), *new-text-item* is made the new key handler. If it returns `nil`, *item* remains the `current-key-handler`.

Arguments

item	An editable-text dialog item.
new-text-item	The editable-text dialog item about to be made current.

`enter-key-handler` [*Generic function*]

Syntax `enter-key-handler` (*item* `key-handler-mixin`) *old-text-item*

Description The generic function `enter-key-handler` is called when a key handler such as an editable-text dialog item has just been made current.

The method for `key-handler-mixin` doesn't do anything; it is a hook on which you can specialize behavior. For example, you can set another dialog item as the current key handler, as in the example.

Arguments

item	An editable-text dialog item.
old-text-item	The previously current editable-text item in the dialog. This is `nil` the first time an editable-text item is added to a dialog.

Example

Here is an example of entering and exiting fields by polling through the key handlers `enter-key-handler` and `exit-key-handler`. The dialog `foo` contains two editable-text dialog items, `Changing` and `Checking`. `Checking` is a simple instance of `editable-text-dialog-item`. `Changing` is an instance of a subclass, `changer-text-item`, which has methods for `enter-key-handler` and `exit-key-handler`. These methods do all the work.

If you edit the text of `Changing`, the `exit-key-handler` method for `changer-text-item` brings up a message when the next item is clicked. If you edit the text of `Checking`, the `enter-key-handler` method for `changer-text-item` returns `nil` and `Checking` remains the `current-key-handler` until the original text is restored.

This example is available as the file `check-and-change.lisp` in the Examples folder distributed as part of Macintosh Common Lisp.

```
;;Checking is a simple editable-text-dialog-item
(setq Checking (make-instance 'editable-text-dialog-item
                :dialog-item-text "Click here to check"
                :view-position #@(16 16)))

;; changer-text-item is a new subclass
(defclass changer-text-item (editable-text-dialog-item) ()
          (:default-initargs :dialog-item-text
                          "Change me and see what happens"))

;; changer-text-item has methods for enter-key-handler
;; and exit-key-handler
(defmethod exit-key-handler
           ((changer-text-item changer-text-item) next-item)
           (declare (ignore next-item))
        (unless (equalp (dialog-item-text changer-text-item)
                           "Change me and see what happens")
             (message-dialog "You changed me!"))
 t)

(defmethod enter-key-handler
```

```
           ((changer-text-item changer-text-item) old-text)
           (unless (equalp (dialog-item-text Checking)
                           "Click here to check")
             (set-current-key-handler
              (view-window changer-text-item) old-text)))

(setq foo (make-instance 'dialog))

(setq Changing (make-instance 'changer-text-item
                :view-position #@(10 100)
                :draw-outline nil))

(add-subviews foo Checking Changing)
```

`allow-returns-p` [*Generic function*]

Syntax `allow-returns-p (`*item* `key-handler-mixin)`

Description The generic function `allow-returns-p` returns true if carriage returns are allowed in the editable-text dialog item. Otherwise, it returns false.

Argument *item* An editable-text dialog item.

`set-allow-returns` [*Generic function*]

Syntax `set-allow-returns (`*item* `key-handler-mixin)` *value*

Description The generic function `set-allow-returns` sets whether carriage returns are allowed in the editable-text dialog item.

Arguments *item* An editable-text dialog item.

value If *value* is true, carriage returns are allowed. If it is `nil`, they are not.

`allow-tabs-p` [*Generic function*]

Syntax `allow-tabs-p (`*item* `key-handler-mixin)`

Description The `allow-tabs-p` generic function returns true if tabs are allowed in the editable-text dialog item. Otherwise, it returns false.

Argument *item* An editable-text dialog item.

`set-allow-tabs` [*Generic function*]

Syntax `set-allow-tabs (`*item* `editable-text-dialog-item)` *value*

Description The `set-allow-tabs` generic function sets whether tabs are allowed in the editable-text dialog item.

Arguments

item An editable-text dialog item.

value If *value* is true, tabs are allowed. If it is `nil`, they are not.

`cut` [*Generic function*]
`copy` [*Generic function*]
`paste` [*Generic function*]
`clear` [*Generic function*]
`undo` [*Generic function*]
`undo-more` [*Generic function*]
`select-all` [*Generic function*]

Syntax

`cut (`*window* `window)`
`copy (`*window* `window)`
`paste (`*window* `window)`
`clear (`*window* `window)`
`undo (`*window* `window)`
`undo-more (`*window* `window)`
`select-all (`*window* `window)`

Description These generic functions are each specialized on the `window` class (as well as on `fred-mixin`, described inChapter 14: Programming the Editor). Each generic function calls the same generic function on the current key handler of *window*, if there is one. The methods applicable to `fred-mixin` perform the operation.

Argument

window A window whose first direct superclass is `fred-mixin`, which provides editing capability.

Checkboxes

Checkboxes are small squares that toggle an X mark on and off when clicked. The following class and functions govern the behavior of checkboxes.

`check-box-dialog-item` [*Class name*]

Description The `check-box-dialog-item` class is the class of checkbox dialog items.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*dialog-item* `check-box-dialog-item`)
`&rest` *initargs*

Description The `initialize-instance` primary method for `check-box-dialog-item` initializes a checkbox dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item A checkbox dialog item.

initargs A list of keywords and values used to initialize the checkbox. The additional initialization argument keyword for checkboxes is

`:check-box-checked-p`
This keyword specifies whether the item is initially checked. Its value is true if the item is checked and `nil` if it is not. Its default value is `nil`.

`dialog-item-action` [*Generic function*]

Syntax `dialog-item-action` (*item* `check-box-dialog-item`)

Description The `check-box-dialog-item` primary method for `dialog-item-action` toggles the state of the box from unchecked to checked or vice versa, then calls `call-next-method`.

Argument *item* A checkbox dialog item.

`check-box-check` [*Generic function*]

Syntax `check-box-check` (*item* `check-box-dialog-item`)

Description The `check-box-check` generic function places an X in the checkbox. The function merely places an X in the box; it does not run the action of the dialog item.

Argument *item* A checkbox dialog item.

`check-box-uncheck` [*Generic function*]

Syntax `check-box-uncheck` (*item* `check-box-dialog-item`)

Description The `check-box-uncheck` generic function removes the X from the checkbox. The function merely removes the X from the box; it does not run the action of the dialog item. The function returns `nil`.

Argument *item* A checkbox dialog item.

`check-box-checked-p` [*Generic function*]

Syntax `check-box-checked-p` (*item* `check-box-dialog-item`)

Description The `check-box-checked-p` generic function returns `t` if there is an X in the checkbox and `nil` otherwise. The function merely reports on the state of the box; it does not run the action of the dialog item.

Argument *item* A checkbox dialog item.

Radio buttons

Radio buttons are small circles that contain a black dot when they are selected ("pushed"). Radio buttons occur in clusters, and only one button in a cluster may be pushed at a time. Clicking a radio button unpushes the previously pushed one. The following class and functions govern the behavior of radio buttons.

`radio-button-dialog-item` [*Class name*]

Description The `radio-button-dialog-item` class is the class of radio-button dialog items.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*item* `radio-button-dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `radio-button-dialog-item` initializes a radio-button dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item A radio-button dialog item.

initargs A list of keywords and values used to initialize a radio button. The *initargs* keywords, in addition to those for `dialog-item`, are

`:radio-button-cluster`
The cluster to which the radio button belongs. Only one button from a given cluster can be pushed at a time. Whenever the user clicks a button, the function `radio-button-unpush` is applied to all other buttons having the same value for `radio-button-cluster.` To check to see whether two buttons are in the same cluster, use `eq.` The default cluster is 0.

`:radio-button-pushed-p`
This keyword determines whether the radio button is initially pushed. The default value is `nil`.

`radio-button-cluster` [*Generic function*]

Syntax `radio-button-cluster` (*item* `radio-button-dialog-item`)

Description The `radio-button-cluster` generic function returns the cluster of *item* as an integer.

Argument *item* A radio-button dialog item.

`pushed-radio-button` [*Generic function*]

Syntax `pushed-radio-button` (*window* `window`) `&optional` *cluster*

Description The `pushed-radio-button` generic function returns the pushed radio button from the specified cluster. The value `nil` is returned if there is no such cluster or if all the radio buttons in a cluster are disabled.

Arguments

window	A window.
cluster	The cluster of radio buttons to search. Radio button clusters are numbered, starting with 0. The default is 0.

`radio-button-push` [*Generic function*]

Syntax `radio-button-push` (*item* `radio-button-dialog-item`)

Description The `radio-button-push` generic function pushes a radio button and unpushes the previously pushed one. The function merely toggles the states of the two radio buttons; it does not run any action. The function returns `nil`.

Argument *item* A radio-button dialog item.

`radio-button-unpush` [*Generic function*]

Syntax `radio-button-unpush` (*item* `radio-button-dialog-item`)

Description The `radio-button-unpush` generic function unpushes the radio button and returns `nil`.

Argument *item* A radio-button dialog item.

`radio-button-pushed-p` [*Generic function*]

Syntax `radio-button-pushed-p` (*item* `radio-button-dialog-item`)

Description The `radio-button-pushed-p` generic function returns `t` if the radio button is pushed and `nil` if it is not. The default value is `nil`.

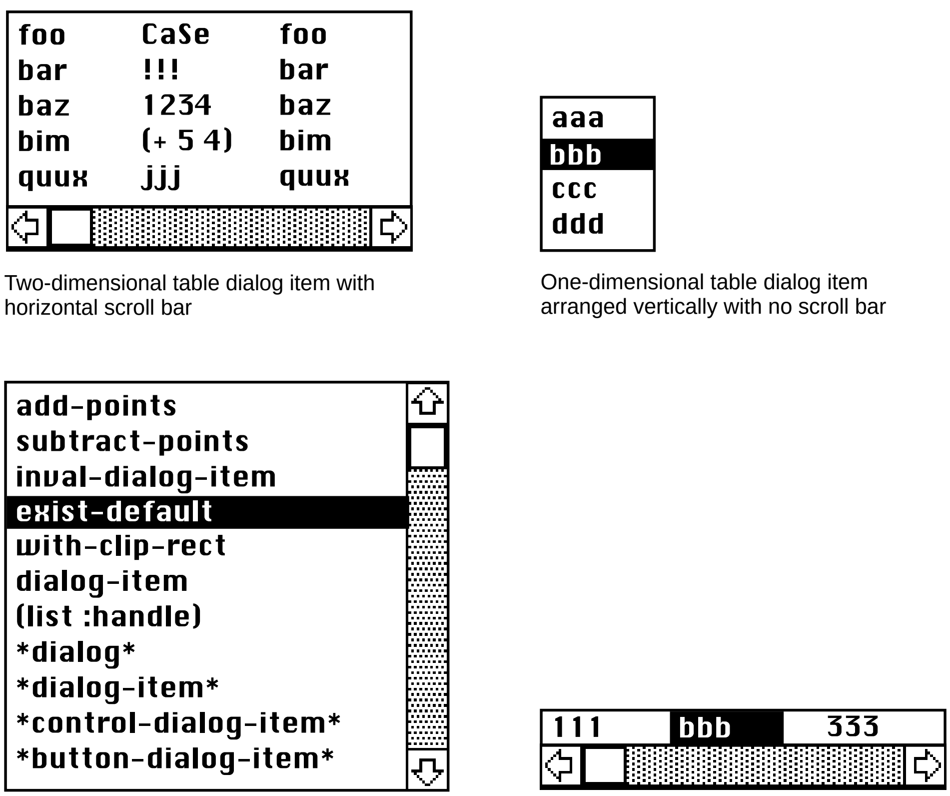
Argument *item* A radio-button dialog item.

Table dialog items

Table dialog items are tables within a window. They allow the user to view a set of items and select items from the set. These tables may be one- or two-dimensional (see Figure 5-1). Two-dimensional tables look like spreadsheets. One-dimensional tables look like the file selection boxes displayed after a user chooses the Save as command. Each item in a table takes up one cell, and there is an 8 KB limit on the total number of cells a table may have.

Table dialog items are implemented using the Macintosh List Manager (but are not called "lists" to avoid confusion with Lisp lists).

■ **Figure 5-1** Examples of tables used in dialog boxes



Two-dimensional table dialog item with horizontal scroll bar

One-dimensional table dialog item arranged vertically with no scroll bar

One-dimensional table dialog item arranged vertically with vertical scroll bar

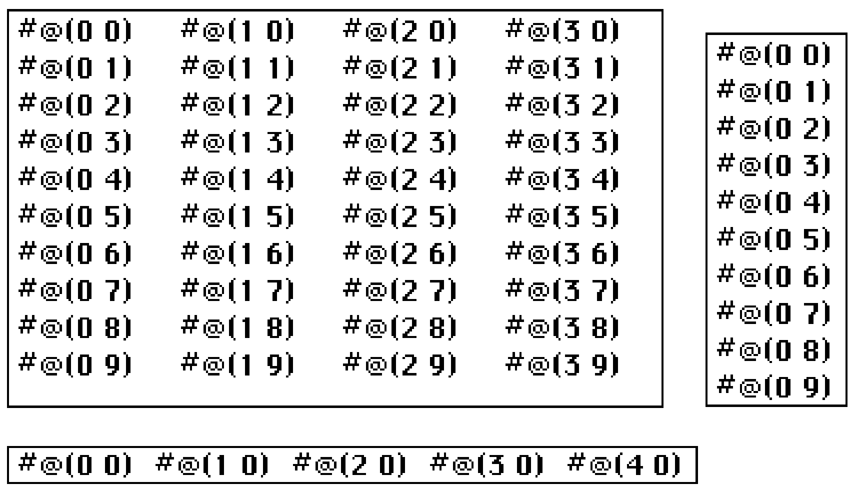
One-dimensional table dialog item arranged horizontally with horizontal scroll bar

All the functions used with other dialog items (such as `view-size` and `view-position`) work for tables, except that the text of table dialog items is not shown.

Table dialog items are rectangles with a series of cells (see Figure 5-2). Your program can access information about table dialog items, such as the cells that are selected, the position of any cell, and the contents of any cell.

A cell is referenced by a point, encoding the horizontal and vertical indices of the cell within the table.

■ **Figure 5-2** Cell positions represented as points



`table-dialog-item` [*Class name*]

Description

The `table-dialog-item` class provides the base functionality for all types of table dialog items. You should not directly instantiate this class, but should create subclasses from it.

The common uses of table dialog items are provided by sequence dialog items, described “Sequence dialog items” on page 234. However, you may want to implement your own subclass of table dialog items with specialized behavior. The file `array-dialog-item.lisp` in your MCL Examples folder implements a class of tables displaying multidimensional arrays.

`initialize-instance` [*Generic function*]

Syntax

```
initialize-instance (item table-dialog-item)
      &rest initargs &key :table-dimensions :selection-type
      :table-print-function :table-vscrollp :table-hscrollp
      :grow-icon-p :cell-fonts :cell-size
      :visible-dimensions
```

Description The `initialize-instance` primary method for `table-dialog-item` initializes a table dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item A table dialog item.

initargs The initialization arguments for the menu item and their initial values, if any. These are its special *initargs* keywords (in addition to those for `dialog-item`):

`:table-dimensions`
The horizontal and vertical dimensions of the table in number of cells, expressed as a point. The default value is `#@(0 0)`. Due to a limitation of the Macintosh List Manager, no table dialog item may have more than 8192 (8 KB) cells.

`:selection-type`
This keyword determines whether the table dialog item allows single or multiple selections, and whether multiple selections must be contiguous. Possible keywords are `:single`, `:contiguous`, and `:disjoint`. The default value is `:single.`

◆*Note:* To get a :`disjoint` selection, you must hold down the Command key as you select items. To get a `:contiguous` selection, hold down the Shift key.

`:table-print-function`
The function used by `draw-cell-contents` to print the contents of the cell. The default value is `#'princ`. If given, this should be a function of two arguments, the value to be printed and the stream.

`:table-vscrollp`
This keyword determines whether the table dialog item has a vertical scroll bar. The default is to include a scroll bar if one is needed in order to view the entire table.

`:table-hscrollp`
This keyword determines whether the table dialog item has a horizontal scroll bar. The default is to include a scroll bar if one is needed in order to view the entire table.

`:grow-icon-p`
: The value passed as the `HasGrow` parameter to the `#_LNew` trap when `install-view-in-window` creates the table. The default value is `nil`.

`:cell-fonts`
: A property list of cells and font specs. See the description of `set-cell-font`, later in this section.

`:cell-size`
: Horizontal and vertical dimensions of the cells in the table dialog item. The default value is `nil`, meaning that the cell size is computed to be big enough to accommodate the values of all the cells.

`:visible-dimensions`
: The visible dimensions of the table. The default value is `nil`, meaning that the visible dimensions of the table are calculated and the entire table is visible.

`cell-contents` [*Generic function*]

Syntax `cell-contents` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-contents` generic function returns the contents of the cell specified by *h* and *v*. The method for `table-dialog-item` returns `nil`.

The `cell-contents` method should be specialized by subclasses of `table-dialog-item`. It is called by `draw-cell-contents`.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`redraw-cell` [*Generic function*]

Syntax `redraw-cell` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `redraw-cell` generic function redraws the contents of *cell*. When a single cell changes, calling this function explicitly is much more efficient than redrawing the entire table dialog item.

Redrawing the cell involves three operations:

1. Setting the dialog’s clip rectangle so that drawing is restricted to the cell.
2. Moving the pen to a position 3 pixels above the bottom of the cell and 3 pixels to the right of the left edge of the cell.

3. Calling `draw-cell-contents`.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

draw-cell-contents [*Generic function*]

Syntax `draw-cell-contents` (*item* `table-dialog-item`) *h*
`&optional` *v*

Description The `draw-cell-contents` generic function draws the contents of *cell*. It may be shadowed to provide a specialized display. This function should not be called directly. It should be called only by `redraw-cell`, which prepares the window for the drawing.

The default method of `draw-cell-contents` shows the printed representation of the cell contents (using the function stored in the function cell of `:table-print-function`, which defaults to `princ`). If the contents are too long to fit in the cell, an ellipsis is added at the end.

The `draw-cell-contents` function may be shadowed to provide specialized drawing (for example, to create a table of icons or patterns). In many cases, however, you don't need to redefine `draw-cell-contents`; you can often achieve the desired results with a function in `:table-print-function`.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

highlight-table-cell [*Generic function*]

Syntax `highlight-table-cell` (*item* `table-dialog-item`) *cell rect*
selectedp

Description The `highlight-table-cell` generic function highlights *cell*. This function may be shadowed to provide a specialized display. The `highlight-table-cell` function should not be called directly. It is automatically called by the `view-click-event` handler for `table-dialog-item`.

Arguments

item	A table dialog item.
cell	The cell to be drawn.

rect	The bounding rectangle of *cell*.
selectedp	The state (selected or unselected) of the cell. If the value of *selectedp* is true, the cell is selected. If it is `nil`, the cell is unselected.

`table-dimensions` [*Generic function*]

Syntax `table-dimensions` (*item* `table-dialog-item`)

Description The `table-dimensions` generic function returns a point indicating the number of cells horizontally and vertically in the table dialog item.

Argument *item* A table dialog item.

`set-table-dimensions` [*Generic function*]

Syntax `set-table-dimensions` (*item* `table-dialog-item`) *h*
`&optional` *v*

Description The `set-table-dimensions` generic function sets the number of cells horizontally and vertically according to *h* and *v*.

There is an 8 KB limit on the total number of cells.

Arguments

item	A table dialog item.
h	Horizontal number of cells.
v	Vertical number of cells. If the value of *v* is `nil`, *h* is assumed to represent a point.

`visible-dimensions` [*Generic function*]

Syntax `visible-dimensions` (*item* `table-dialog-item`)

Description The `visible-dimensions` generic function returns a point indicating the number of cells visible in the horizontal and vertical dimensions.

Argument *item* A table dialog item.

`set-visible-dimensions` [*Generic function*]

Syntax `set-visible-dimensions` (*item* `table-dialog-item`) *h*
`&optional` *v*

Description The `set-visible-dimensions` generic function resizes the table so that *h* cells are visible per row and *v* cells are visible per column. The new dimensions are returned as a point.

Arguments

item	A table dialog item.
h	Horizontal number of cells.
v	Vertical number of cells. If the value of *v* is `nil`, *h* is assumed to represent a point.

The `cell-size` and `set-cell-size` functions that follow provide an alternative to `view-size` for specifying the size of a table dialog item.

`cell-size` [*Generic function*]

Syntax `cell-size` (*item* `table-dialog-item`)

Description The `cell-size` generic function returns the size of a cell in the table dialog item. All the cells have the same size.

Argument

item	A table dialog item.

`set-cell-size` [*Generic function*]

Syntax `set-cell-size` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `set-cell-size` generic function sets the cell size according to *h* and *v* and returns the new size as a point.

Arguments

item	A table dialog item.
h	Horizontal size (width).
v	Vertical size (height). If the value of *v* is `nil`, *h* is assumed to represent a point.

`cell-font` [*Generic function*]

Syntax `cell-font` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-font` generic function returns the font used by a cell (specified by *h* and *v*) or `nil` if the cell uses the font of the dialog item.

Arguments

item	A table dialog item.
h	Horizontal index.

v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`set-cell-font` [*Generic function*]

Syntax `set-cell-font` (*item* `table-dialog-item`) *cell font-spec*

Description The `set-cell-font` generic function sets the font of *cell* to *font-spec*.

Arguments

item	A table dialog item.
cell	A cell in the table dialog item, encoded as a point.
font-spec	A font spec.

`part-color` [*Generic function*]

Syntax `part-color` (*item* `table-dialog-item`) *part*

Description The `part-color` method for `table-dialog-item` returns the color of the part of the table dialog item indicated by *part*.

Arguments

item	A table dialog item.
part	A keyword. In addition to the keywords allowed for dialog items, *part* may be a point indicating a cell.

`set-part-color` [*Generic function*]

Syntax `set-part-color` (*item* `table-dialog-item`) *part color*

Description The `set-part-color` method for `table-dialog-item` sets the color of part of the table dialog item, as specified by the arguments, and returns *color*.

Arguments

item	A table dialog item.
part	In addition to the keywords allowed for dialog items, *part* may be an integer indicating a cell. The default cell-drawing routine draws the contents of the cell in the color you have specified.
color	A color, encoded as a point.

`cell-select` [*Generic function*]

Syntax `cell-select` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-select` generic function selects the cell specified by *h* and *v*. Previously selected cells are not affected.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`cell-deselect` [*Generic function*]

Syntax `cell-deselect` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-deselect` generic function deselects the cell specified by *h* and *v*.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`cell-selected-p` [*Generic function*]

Syntax `cell-selected-p` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-selected-p` generic function returns `t` if the cell specified by *h* and *v* is selected. Otherwise, it returns `nil`.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`selected-cells` [*Generic function*]

Syntax `selected-cells` (*item* `table-dialog-item`)

Description The `selected-cells` generic function returns a list of all the cells selected in the table dialog item. Each cell is represented by a point. If no cells are selected, `nil` is returned.

Argument *item* A table dialog item.

`scroll-to-cell` [*Generic function*]

Syntax `scroll-to-cell` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `scroll-to-cell` generic function causes the table dialog item to scroll so that the cell specified by *h* and *v* is in the upper-left corner.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`scroll-position` [*Generic function*]

Syntax `scroll-position` (*item* `table-dialog-item`)

Description The `scroll-position` generic function returns the cell indices of the cell in the upper-left corner of the table dialog item. (This is not a position in window coordinates but indicates which cell is in the upper-left corner.)

Argument *item* A table dialog item.

`cell-position` [*Generic function*]

Syntax `cell-position` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-position` generic function returns the position of the upper-left corner of the cell if the cell is visible. It returns `nil` if the cell is not currently visible. The position returned is in the coordinate system of the item's container.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`point-to-cell` [*Generic function*]

Syntax `point-to-cell` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `point-to-cell` generic function returns the cell enclosing the point represented by *h* and v, or `nil` if the point is not within a cell.

Arguments

item	A table dialog item.
h	Horizontal position.
v	Vertical position. If the value of *v* is `nil`, *h* is assumed to represent a point.

`table-hscrollp` [*Generic function*]

Syntax `table-hscrollp` (*item* `table-dialog-item`)

Description The `table-hscrollp` generic function returns `t` if *item* has a horizontal scroll bar and `nil` otherwise.

Argument *item* A table dialog item.

`table-vscrollp` [*Generic function*]

Syntax `table-vscrollp` (*item* `table-dialog-item`)

Description The `table-vscrollp` generic function returns `t` if *item* has a vertical scroll bar and `nil` otherwise.

Argument *item* A table dialog item.

`table-print-function` [*Generic function*]

Syntax `table-print-function` (*item* `table-dialog-item`)

Description The `table-print-function` generic function returns the function used by `draw-cell-contents` to print the contents of the cell.

Argument *item* A table dialog item.

Pop-up menu dialog items

A pop-up menu dialog item is a menu within a dialog box or other view containing dialog items. The Commands menu in Inspector windows is an example of a pop-up menu. For other examples, look at the file `CCL:library;pop-up-menu.lisp`.

The following MCL expressions govern the behavior of pop-up menus.

`pop-up-menu` [*Class name*]

Description The class `pop-up-menu` is the class of pop-up menus, built on `menu`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*menu* `pop-up-menu`) `&rest` *initargs*

Description The `initialize-instance` primary method for `pop-up-menu` initializes a pop-up menu. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

menu A pop-up menu.

initargs A set of initial arguments and values used for initializing the pop-up menu:

`:default-item`
An integer identifying the default item that will be selected from the menu. The default is `1`. The first item is `1`, not `0`.

`:auto-update-default`
An argument specifying how defaults are handled. If true (the default), each time an item is selected from the pop-up menu, it becomes the default. Otherwise, the default item remains fixed.

`:item-display`
An argument specifying whether the menu item or its value is displayed. If the value is `:selection` (the default), displays the default menu item. Otherwise the value itself is displayed as if by `(format t "~a"` *value*`)`.

`:menu-items`
A list of items to be added to the newly created pop-up menu.

`:menu-colors`
: A property list of menu parts and colors. The allowable parts are given in the definition of `set-part-color`. For details, see “Menubar colors” on page 98 and Chapter 6: Color.

`:dialog-item-text`
: The text of the pop-up menu. The default value is `""`. If a value is specified and is not `""`, this becomes a label for the pop-up menu, which is displayed to the left of the box for the `:item-display`.

`:dialog-item-action`
: The `dialog-item-action` generic function is not called by `view-click-event-handler` for a pop-up menu.

`:help-spec`
: A value describing the Balloon Help for the item. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

Example

See the file `pop-up-menu.lisp` in your MCL Library folder.

Scroll-bar dialog items

A scroll-bar dialog item is a dialog item that is a scroll bar. The following MCL expressions govern the behavior of scroll-bar dialog items.

`scroll-bar-dialog-item` [*Class name*]

Description The `scroll-bar-dialog-item` class is the class of scroll-bar dialog items.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*item* `scroll-bar-dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `scroll-bar-dialog-item` initializes a scroll-bar dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

For full information on scroll bars, see *Inside Macintosh.*

Arguments

item A scroll-bar dialog item.

initargs A set of initial arguments and values used for initializing the scroll-bar dialog item:

`:direction`
The direction of the scroll bar. Valid values are `:horizontal` and `:vertical` (the default).

`:max` The maximum setting of the scroll bar. This value must be an integer; it defaults to 100.

`:min` The minimum setting of the scroll bar. This value must be an integer; it defaults to 0.

`:page-size`
The amount the setting of the scroll bar will change when the user clicks the gray area above or below the scroll box. The default value is 5.

`:scroll-size`
The amount the setting of the scroll bar will change when the user clicks one of the arrows at its two ends. The default value is 1.

`:setting` The initial setting of the scroll bar.

`:track-thumb-p`
An argument specifying behavior during scrolling. If true, the scroll box is moved and `scroll-bar-changed` is called as the user drags the scroll box. Otherwise, an outline is dragged and the scrolling does not actually happen until the user releases the mouse button. The default value is `nil`.

`:scrollee`
An argument specifying what it is that the scroll bar scrolls. The default value is `nil`.

`:pane-splitter`
An argument specifying the position of a pane splitter. If the scroll bar is `:vertical`, a value of `:top` means above the scroll bar and any other non-`nil` value means below it. If the scroll bar is `:horizontal`, a value of `:left` means to the left of the scroll bar and any other non-`nil` value means to the right of it. If `nil`, there is no pane splitter. The default value is `nil`.

`:help-spec`

A value describing the Balloon Help for the item. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

`scroll-bar-length` [*Generic function*]

Syntax `scroll-bar-length` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-length` generic function returns the length of *item*.

Argument *item* A scroll-bar dialog item.

`set-scroll-bar-length` [*Generic function*]

Syntax `set-scroll-bar-length` (*item* `scroll-bar-dialog-item`) *new-length*

Description The `set-scroll-bar-length` generic function sets the length of *item* to *new-length*.

Arguments *item* A scroll-bar dialog item.

new-length The new length of *item*.

`scroll-bar-max` [*Generic function*]

Syntax `scroll-bar-max` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-max` generic function returns the maximum setting of *item*.

Argument *item* A scroll-bar dialog item.

`set-scroll-bar-max` [*Generic function*]

Syntax `set-scroll-bar-max` (*item* `scroll-bar-dialog-item`) *new-value*

Description The `set-scroll-bar-max` generic function sets the maximum setting of *item* to *new-value*.

Arguments *item* A scroll-bar dialog item.
new-value The new maximum setting of *item*.

`scroll-bar-min` [*Generic function*]

Syntax `scroll-bar-min` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-min` generic function returns the minimum setting of *item*.

Argument *item* A scroll-bar dialog item.

`set-scroll-bar-min` [*Generic function*]

Syntax `set-scroll-bar-min` (*item* `scroll-bar-dialog-item`) *new-value*

Description The `set-scroll-bar-min` generic function sets the minimum setting of *item* to *new-value*.

Arguments *item* A scroll-bar dialog item.
new-value The new minimum setting of *item*.

`scroll-bar-page-size` [*Generic function*]

Syntax `scroll-bar-page-size` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-page-size` generic function returns the page size of *item*.

Argument *item* A scroll-bar dialog item.

`scroll-bar-scroll-size` [*Generic function*]

Syntax `scroll-bar-scroll-size` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-scroll-size` generic function returns the scroll size of *item*.

Argument *item* A scroll-bar dialog item.

`scroll-bar-scrollee` [*Generic function*]

Syntax `scroll-bar-scrollee` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-scrollee` generic function retrieves the scrollee of *item* (that is, what *item* is scrolling).

Argument *item* A scroll-bar dialog item.

`set-scroll-bar-scrollee` [*Generic function*]

Syntax `set-scroll-bar-scrollee` (*item* `scroll-bar-dialog-item`) *new-scrollee*

Description The `set-scroll-bar-scrollee` generic function sets the scrollee of *item* (that is, what *item* is scrolling) to *new-scrollee*.

Arguments *item* A scroll-bar dialog item.

new-scrollee The new scrollee of *item*.

`scroll-bar-setting` [*Generic function*]

Syntax `scroll-bar-setting` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-setting` generic function returns the current setting of *item*.

Argument *item* A scroll-bar dialog item.

`set-scroll-bar-setting` [*Generic function*]

Syntax `set-scroll-bar-setting` (*item* `scroll-bar-dialog-item`) *new-setting*

Description The `set-scroll-bar-setting` generic function sets the setting of *item* to *new-setting*. It does not call `dialog-item-action`.

Arguments *item* A scroll-bar dialog item.

new-setting The new setting of *item*.

`scroll-bar-track-thumb-p` [*Generic function*]

Syntax `scroll-bar-track-thumb-p` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-track-thumb-p` generic function returns a value indicating the behavior of *item* when the scroll box is dragged. If true, the scroll box moves and the function `scroll-bar-changed` is called as the user drags the scroll box. If `nil`, only an outline of the scroll box moves and scrolling does not occur until the user releases the mouse button. The default value is `nil`.

Argument *item* A scroll-bar dialog item.

`set-scroll-bar-track-thumb-p` [*Generic function*]

Syntax `set-scroll-bar-track-thumb-p` (*item* `scroll-bar-dialog-item`) *value*

Description The `set-scroll-bar-track-thumb-p` generic function sets the value controlling the behavior of *item* when the scroll box is dragged. If true, the scroll box moves and the function `scroll-bar-changed` is called as the user drags the scroll box. If `nil`, only an outline of the scroll box moves and scrolling does not occur until the user releases the mouse button.

Arguments *item* A scroll-bar dialog item.

value A Boolean value. If *item* does not have a scroll box, the value is `nil`.

`scroll-bar-width` [*Generic function*]

Syntax `scroll-bar-width` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-width` generic function returns the width of *item*.

Argument *item* A scroll-bar dialog item.

`set-scroll-bar-width` [*Generic function*]

Syntax `set-scroll-bar-width` (*item* `scroll-bar-dialog-item`) *new-width*

Description The `set-scroll-bar-width` generic function sets the width of *item* to *new-width*.

Arguments

item	A scroll-bar dialog item.
new-value	The new width of *item*.

`scroll-bar-changed` [*Generic function*]

Syntax `scroll-bar-changed` (*scrollee* `t`) (*scroll-bar* `t`)

Description The `scroll-bar-changed` generic function is called by the `dialog-item-action` method for `scroll-bar-dialog-item` if the `dialog-item-action-function` specified by the `:dialog-item-action` initialization argument is `nil`. The *scrollee* argument is the value of `(scroll-bar-scrollee` *scroll-bar*`)`, as set by `set-scroll-bar-scrollee` or the `:scrollee` initialization argument for *scroll-bar*. The default method does nothing.

Writing a `scroll-bar-changed` method is an easy way to cause user mouse clicks on a scroll-bar dialog item to update another view.

Arguments

scrollee	A scroll-bar scrollee; what is scrolled by the dialog item.
scroll-bar	A scroll bar.

Sequence dialog items

A sequence dialog item is a table dialog item that displays the elements of a sequence, either row by row or column by column. The following class and functions govern the behavior of sequence dialog items.

`sequence-dialog-item` [*Class name*]

Description The `sequence-dialog-item` class is the class of sequence dialog items, used for displaying the elements of a sequence. It is a subclass of `table-dialog-item`. Each instance has an associated sequence. The elements of the sequence are displayed in a table dialog item, in a single row or column, or in multiple rows and columns. The table dialog item has multiple rows and columns only if the length of the sequence is greater than `:sequence-wrap-length`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*item* `sequence-dialog-item`) `&rest` *initargs*

Description The `initialize-instance` primary method for `sequence-dialog-item` initializes a sequence dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item A sequence dialog item.

initargs A list of keywords and values used to initialize the sequence. These are the *initargs* keywords in addition to those used for `table-dialog-item`:

`:table-sequence`
The sequence to be associated with the table dialog item. This argument must be specified by the user.

`:sequence-order`
This keyword determines whether the sequence will fill the table dialog item row by row or column by column. The value of this keyword should be either `:vertical` or `:horizontal`. The default is `:vertical`.

`:sequence-wrap-length`
The number of elements allowed in a row or column before the table dialog item wraps to the next row or column. This number overrides the `:table-dimensions` argument.

`table-sequence` [*Generic function*]

Syntax `table-sequence` (*item* `sequence-dialog-item`)

Description The `table-sequence` generic function returns the sequence associated with the dialog item.

Argument *item* A sequence dialog item.

`set-table-sequence` [*Generic function*]

Syntax `set-table-sequence` (*item* `sequence-dialog-item`) *new-sequence*

Description The `set-table-sequence` generic function sets the sequence associated with the dialog item to *new-sequence*, resets the dimensions of the table dialog item and the scroll bars, and redisplays the dialog item.

Arguments

item	A sequence dialog item.
new-sequence	The sequence to be associated with the sequence dialog item. The elements of this sequence are displayed in the cells of the sequence dialog item.

`cell-to-index` [*Generic function*]

Syntax `cell-to-index` (*item* `sequence-dialog-item`) *h* `&optional` *v*

Description The `cell-to-index` generic function returns an index into the sequence associated with the dialog item, corresponding to the cell whose indices in the table are *h* and *v*. If there is no such cell, it returns `nil`.

This index is suitable for passing to the Common Lisp function `elt`.

Arguments

item	A sequence dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

`index-to-cell` [*Generic function*]

Syntax `index-to-cell` (*item* `sequence-dialog-item`) *index*

Description The `index-to-cell` generic function returns a cell in the dialog item. The cell corresponds to the *index*th element of the table's sequence.

Arguments

item	A sequence dialog item.
index	An index to the sequence (zero based, as would be passed to `elt`).

User-defined dialog items

You can easily add new classes of dialog items to the classes predefined in Macintosh Common Lisp.

New classes of dialog items may be specializations of the types of dialog items listed in this chapter or specializations of the class `dialog-item`. Functions that you may wish to define for classes inheriting from `dialog-item` are listed in “Advanced dialog item functions” on page 198.

For a commented example of how to implement your own class of dialog item, see the file `scrolling-fred-dialog-item.lisp` in the Library folder distributed with Macintosh Common Lisp.

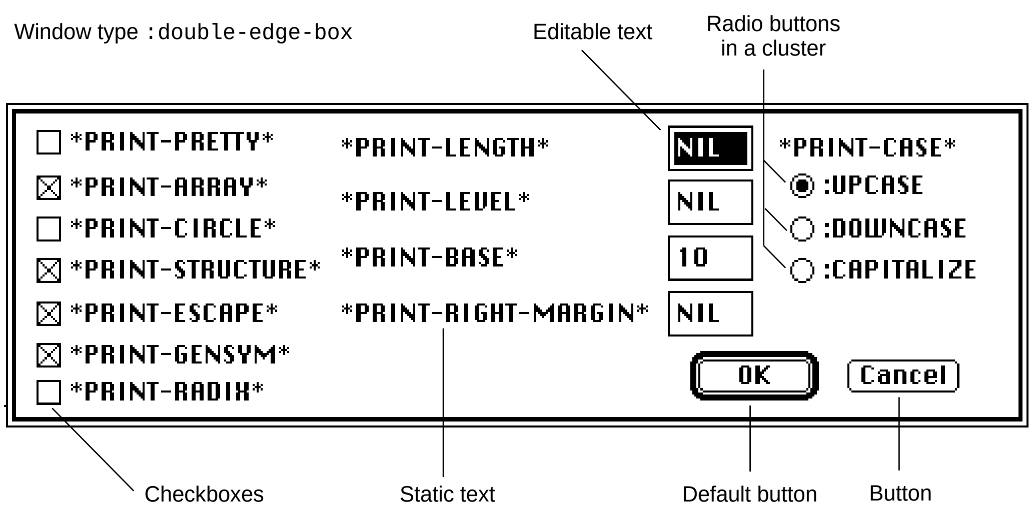
Dialogs

A dialog may be either **modal** or **modeless.**

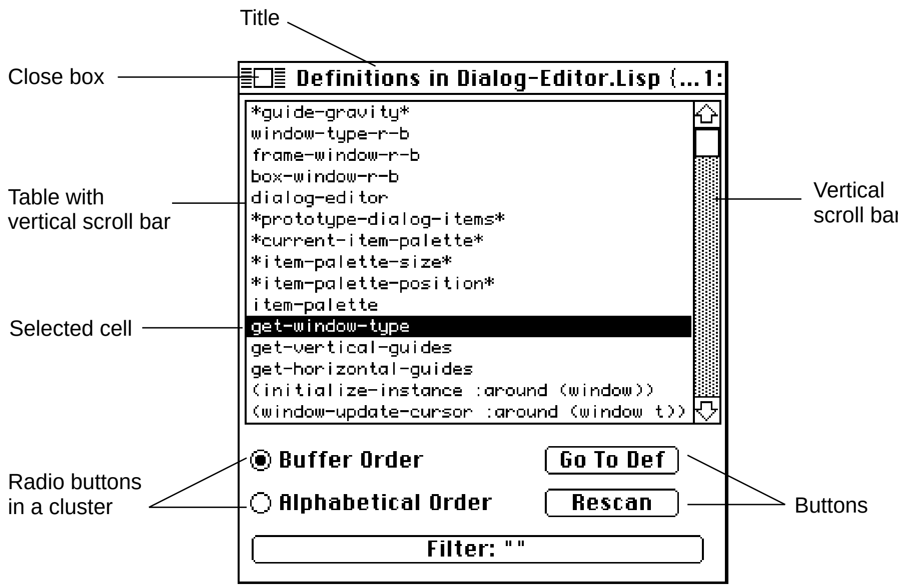
- The user must exit from a modal dialog before performing any other actions. The Print Options dialog box (Figure 5-3) is an example of a modal dialog.
- If the dialog is modeless, other actions can occur while the dialog is still on the screen. The List Definitions dialog box (Figure 5-4) is an example of a modeless dialog.

How the dialog is used determines whether it is modal or modeless. Instance values do not determine its mode.

■ **Figure 5-3** A modal dialog (Print Options on the Tools menu)



■ **Figure 5-4** A modeless dialog (List Definitions on the Tools menu)



Modal dialogs

A modal dialog is activated by calling the `modal-dialog` generic function on the dialog. The dialog is displayed and made the active window. All subsequent user events are processed by the dialog; illegal events produce a beep, and legal events cause the action of the selected dialog item to be performed. The dialog continues to intercept all events until `return-from-modal-dialog` is called. This macro causes the dialog to be closed or hidden and supplies one or more values to be returned from the call to `modal-dialog`. Modal dialogs may be nested. Command-period can always be pressed to exit one or more modal dialogs.

Some predefined modal dialogs are documented in "Simple turnkey dialog boxes" on page 239.

Modeless dialogs

A modeless dialog is available for use whenever it is visible. Like any window that is not active, a modeless dialog becomes the active window when it is clicked. If a modeless dialog box is the active window, then appropriate user events trigger the actions of its items.

Unless otherwise specified, all the text in a dialog (that is, the text of all the items) appears in the window's current font. The desired font should be set before the dialog is made visible (using `set-view-font` or the `:view-font` initialization argument). A special font may be specified for certain dialog items; the rest of the items appear in the window's current font.

Simple turnkey dialog boxes

Macintosh Common Lisp provides four predesigned dialogs for use by applications.

Three of the following dialog boxes provide facilities for dynamic nonlocal exiting (Common Lisp throwing and catching). Clicking Cancel causes a `throw-cancel` to the nearest `catch-cancel`. If this throw is not caught, clicking Cancel causes a return to the top level (or if it occurs during event processing, the execution of the interrupted program resumes). Common Lisp throw and catch are described in *Common Lisp: The Language.*

`throw-cancel` [*Macro*]

Syntax `throw-cancel &optional` *value-form*

Description The `throw-cancel` macro throws the value of *value-form* to the most recent outstanding `catch-cancel.`

Argument *value-form*A value.

Example

```
? (catch-cancel
    (loop
      (throw-cancel 'foo)))
FOO
```

`catch-cancel` [*Macro*]

Syntax `catch-cancel{`*form*`}*`

Description The `catch-cancel` macro sets up a cancel catch and evaluates *form*. It returns the value of the last *form* if there was no cancel throw. Otherwise, it returns the symbol `:cancel.`

Argument *form* Zero or more Lisp forms.

`message-dialog` [*Function*]

Syntax `message-dialog` *message* `&key :ok-text :size :position`

Description The `message-dialog` function displays a dialog box containing the string *message* and a single button. The function returns `t` when the user clicks this button or presses Return or Enter.

Arguments

message	A string to be displayed as the message in the dialog box.
`:ok-text`	The text to be displayed in the button. The default button text is `OK.` If the text is too long, this string is clipped (that is, the button is not enlarged to accommodate the longer string). You can set the size with the `:size` keyword.
`:size`	The size of the dialog box. The default size is `#@(335 100)`. A larger size provides more room for text.
`:position`	The position of the dialog box. The default position is the top center of the screen.

Example

```
? (message-dialog "Get along, little dogies"
       :ok-text "Giddyap!" :size #@(250 75))
T
```

Figure 5-5 shows a message dialog box.

■ **Figure 5-5** A message dialog box

The file `icon-dialog-item.lisp` in your Examples folder includes a variation of this dialog box containing the standard Macintosh alert icons Stop, Note, and Caution. The file `graphic-items.lisp` in your Library folder shows how to implement generalized graphic items in dialog boxes.

---

y-or-n-dialog [*Function*]

Syntax

`y-or-n-dialog` *message* `&key :size :position :yes-text`
`:no-text :cancel-text :help-spec`

Description

The `y-or-n-dialog` function displays a dialog box containing Yes, No, and Cancel buttons. The display of the dialog box is modal.

If the user clicks the Yes button, the function returns `t`. If the user clicks the No button, the function returns `nil`. If the user clicks the Cancel button, a `throw-cancel` occurs. The default button is the Yes button.

Arguments

message	A string to be displayed as the message in the dialog box.
`:size`	The size of the dialog box. The default size is `#@(318 145)`.
`:position`	The position of the dialog box. The default position is the top center of the screen.
`:yes-text`	The text to be displayed in the Yes button. The default is `Yes`. This is the default button of the dialog box.
`:no-text`	The text to be displayed in the No button. The default text is `No.`
`:cancel-text`	The text to be displayed in the Cancel button. The default text is `Cancel`. If this argument is `nil` instead of a string, no Cancel button will appear in the dialog box.
`:help-spec`	A value describing the Balloon Help for the item. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

Typing the initial character of the text of a button activates it. For example, typing `Y` activates the Yes button, whereas typing `N` activates the No button. In the following example, typing `R` activates the Cancel button.

Example

```
? (y-or-n-dialog "Please turn my landlord into a frog."
      :cancel-text "Ribbet")
```

Figure 5-6 shows a yes-or-no dialog box.

■ **Figure 5-6** A yes-or-no dialog box

get-string-from-user [*Function*]

Syntax

```
get-string-from-user message &key :size :position
      :initial-string :ok-text :cancel-text :modeless
      :window-title :action-function
      :allow-empty-strings
```

Description The `get-string-from-user` function displays a dialog prompting the user for a string, which it returns. The display of the dialog can be modal or modeless. If the value of `:modeless` is true, the dialog has a close box and no cancel button. If it is `nil`, there is a cancel button and no close box. If the cancel button is clicked, a `throw-cancel` occurs.

Arguments

message A string to be displayed as the message in the dialog box.

`:size` The size of the dialog box. The default size is `#@(335 100)`.

`:position` The position of the dialog box. See the *Human Interface Guidelines: The Apple Desktop Interface* for the default position for this dialog box.

`:initial-string` The default string to be displayed in the dialog box.

`:ok-text` The string to be displayed in the default button. If the user clicks this button (or presses Return), `get-string-from-user` returns the current string. The default value is `"OK"`.

`:cancel-text` The string to be displayed in the Cancel button. The cancel button is omitted if the value of `:modeless` is true.

`:modeless` An argument specifying whether the dialog box display is modal or modeless. The default is `nil`, meaning that it is modal.

If `:modeless` is specified as true, the `get-string-from-user` function returns the window it creates immediately, without waiting for the user to interact with it. The `action-function` is called when the user clicks the default button or presses the Return or Enter key. The default value is a function that returns the string.

`:window-title`
The title of the window. The default is `""`.

`:action-function`
If the `:modeless` argument is true, this argument should be a function of one argument. It is called with the string that the user types each time the user clicks the default button or presses the Return or Enter key. The default value is a function that returns the string.

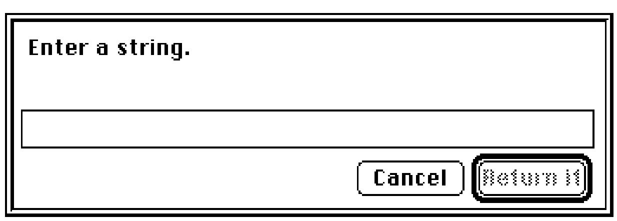
`:allow-empty-strings`
An argument specifying whether the OK button is enabled when the editable-text box contains no text. The default value is `nil`, meaning that the user must type something in the editable-text box to enable the OK button.

Example

```
? (get-string-from-user "Enter a string."
                        :initial-string "A string.")
? (get-string-from-user "Enter a string."
                        :ok-text "Return it")
```

Figure 5-7 shows a dialog box that prompts the user for a string.

■ **Figure 5-7** A `get-string-from-user` dialog box



`select-item-from-list` [*Function*]

Syntax `select-item-from-list` *list* `&key :window-title :table-print-function :selection-type :action-function :modeless :default-button-text`

Description The `select-item-from-list` function displays a dialog box containing a default button and a table that contains the elements of *list*. The function returns a list of the items selected by the user in reverse order, or `nil` if the user chooses the default button. If the value of `:modeless` is `nil` (the default), the dialog has a cancel button; if the user clicks Cancel, a `throw-cancel` occurs.

Arguments

list A list containing the items to be displayed in the table.

`:window-title`
The message displayed at the top of the dialog box.

`:table-print-function`
The print function used by the table in the dialog box. The default is `princ`. You can use this argument to customize the dialog box. For example, you could pass a print function that prints only the first element of lists. (See the documentation of this keyword in "Table dialog items" on page 216.)

`:selection-type`
The type of selection allowed by the table. This should be `:single`, `:contiguous`, or `:disjoint`. The default value is `:single`.

`:action-function`
An argument specifying a function to call when the default button is chosen. The function should take one argument, a list of selected items. The default `action-function` returns a list of selected items.

`:modeless` An argument specifying whether the dialog box display is modal or modeless. The default is `t`, meaning that it is modeless.

If `:modeless` is specified as true, the `select-item-from-list` function returns the window it creates immediately, without waiting for the user to interact with it. The `action-function` is called when the user clicks the default button or presses the Return or Enter key.

`:default-button-text`
A string to appear in the default button. The default value is `"OK"`.

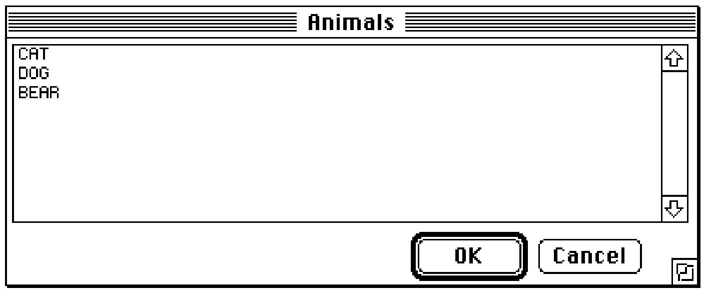
To make a disjoint selection, you must hold down the Command key as you click the selections.

Example

```
? (select-item-from-list '(cat dog bear)
                         :window-title "Animals"
                         :selection-type :disjoint)
; Click the items CAT and BEAR
(BEAR CAT)
```

Figure 5-8 is a modal dialog box that displays a list of items.

■ **Figure 5-8** A `select-item-from-list` dialog box



MCL forms relating to dialogs

The following functions, variables, and macros are useful in programming dialogs (that is, to program instances of `view` or `window` that contain dialog items). Remember that any view or window can contain dialog items, which simply act as subviews within the view, and that any generic function that acts on views or windows can act on ones containing dialog items.

dialog [*Class name*]

Description The `dialog` class is included for compatibility with earlier versions of Macintosh Common Lisp. No methods in Macintosh Common Lisp version 2 are specialized on `dialog`, and it adds no slots.

Instances of `view` or its subclasses can contain a list of dialog items, as you see in the following example, where dialog items appear in a window.

Example

```
? (setq dialog1 (make-instance 'window
     :window-type :document-with-zoom
     :window-title "Button Dialog"
     :view-position '(:TOP 60)
     :view-size #@(300 150)
     :view-font '("Chicago" 12 :SRCOR :PLAIN)
```

```
:view-nick-name 'button-dialog
:view-subviews
   (list
    (setq pearlbutton
     (make-dialog-item 'radio-button-dialog-item
        #@(15 28)
        #@(118 16)
        "Pearl Button"
        #'(lambda (item)
           item
          (print "How elegant!"))
         :view-nick-name 'pearlie
         :view-font '("Chicago" 0 :SRCCOPY :PLAIN)))
    (setq flashbutton
     (make-dialog-item 'radio-button-dialog-item
        #@(15 70)
        #@(217 16)
        "Flashy Plastic Button"
        #'(lambda (item)
           item
          (print "How tacky!"))
        :view-nick-name 'flash
        :view-font '("Chicago" 0 :SRCCOPY :SHADOW))))))
```

modal-dialog [*Generic function*]

Syntax `modal-dialog` (*dialog* `window`)`&optional` *close-on-exit eventhook*

Description The `modal-dialog` generic function displays *dialog* modally. That is, it makes *dialog* the active window, displays it, and then intercepts subsequent user events until a `return-from-modal-dialog` is executed. The function returns the value(s) supplied by `return-from-modal-dialog`.

If *close-on-exit* is true (the default), the window is closed on exit; otherwise, it is hidden.

Closing the dialog box automatically prevents the accumulation of numerous hidden windows during development. Modal dialog boxes may be nested.

◆u *Note:* The body of `modal-dialog` is unwind protected, and so any throw past `modal-dialog` will close or hide the window, as appropriate.

Arguments

window — A window.

close-on-exit — An argument determining whether the window should be closed or simply hidden when the call to `modal-dialog` returns. If this argument is true, the window is closed. If it is false, the window is hidden but not closed. The default is `t.`

eventhook — A hook. The function `modal-dialog` binds `*eventhook*` in order to intercept all event processing; this hook is provided so that you can perform any special event processing while the modal dialog is on the screen. The value of *eventhook* should be a function of no arguments, or a list of functions of no arguments. Whenever `modal-dialog` looks for events, it calls the functions in *eventhook* until one of them returns a non-`nil` result. If all of them return `nil`, `modal-dialog` processes events as it normally would. Otherwise, it assumes that the hook function handled the event.

The variable `*current-event*` is bound to an event record for the current event when each hook function is called.

The default value of *eventhook* is `nil`.

`return-from-modal-dialog` [*Macro*]

Syntax

`return-from-modal-dialog` *values*

Description

The macro `return-from-modal-dialog` causes one or more *values* to be returned from the most recent call to `modal-dialog`.

The dialog is hidden or closed according to the value of *close-on-exit* that was passed to the call to `modal-dialog`. (Any throw past the `modal-dialog` call also causes the dialog box to be hidden or closed.) If the dialog box is only hidden, its contents remain intact and it continues to take up memory until the `window-close` function is explicitly called.

Arguments

values — Any values. The following two values have special meanings:

`:closed` — If a dialog that is used modally has a close box and the window is closed, `return-from-modal-dialog` is called with the value `:closed`.

`:cancel` — If the user selects the cancel button, `return-from-modal-dialog` is called returning `:cancel`. The function `modal-dialog` then performs a `throw-cancel`.

`*modal-dialog-on-top*` [*Variable*]

Description The `*modal-dialog-on-top*` variable is true when a modal dialog is the frontmost window. It is bound during the event processing done by the `modal-dialog` function. Its value is used by the MCL window system code to determine the behavior of floating windows. This value should not be modified by the user, but can be used to determine whether a modal dialog is being processed.

`find-dialog-item` [*Generic function*]

Syntax `find-dialog-item` (*dialog* `dialog`) *string*

Description The `find-dialog-item` generic function returns the first item in the view whose `dialog-item-text` is the same as *string* (using `equalp` for the comparison). The items are searched in the order in which they were added to the view.

This function may yield unexpected results in views with editable-text items. If the user types text identical to the text of another item, the editable-text item may be returned instead of the desired item. For this reason, `find-dialog-item` is best used during programming and debugging sessions.

To identify items in a dialog, you should use nicknames and the functions `view-named` and `find-named-sibling`.

Arguments

dialog	A view or window containing dialog items.
string	A string against which to compare the text of the dialog items.

Chapter 6:

Color

Contents

This chapter describes the implementation of color in Macintosh Common Lisp.

Macintosh Common Lisp includes high-level tools for handling colors. There are functions for encoding and decoding colors (much as points are encoded and decoded), and there are tools for setting the colors of user interface components (windows, menus, and so on).

You should read this chapter before programming color into your application.

For a complete description of color operations on the Macintosh computer, see *Inside Macintosh*.

Color encoding in Macintosh Common Lisp

The Macintosh stores colors as 48-bit red-green-blue (RGB) values, with 16 bits each for the red, green, and blue components. Because current hardware generally supports a maximum of 24 bits of color, Macintosh Common Lisp encodes colors as fixnums with 8 bits each for red, green, and blue (and 5 bits unused). Therefore, creating a color encoding does not allocate memory.

If your application requires more than 24 bits of color, you can redefine the color encoding and decoding operations.

Although they are stored as 8-bit values when encoded in a color, decoded components are expressed as 16-bit values. This allows compatibility with some Macintosh tools (such as the Color Picker). Unfortunately, it also means that the low 8 bits of each color component are lost when the color is encoded and decoded. For example, consider the following expressions, in which the red component of two colors differs in the low 8 bits. Encoding and decoding loses information:

```
? (make-color 32256 14000 27323)  ;;#$7E00=32256
8271466
? (eql 32256 (color-red 8271466))
T
? (make-color 32333 14000 27323)  ;;#$7E4D=32333
8271466
? (equal 32333 (color-red 8271466))
NIL
```

To compare colors for equality as they are actually displayed on the current display device, use the function `real-color-equal`.

```
? (real-color-equal (make-color 32256 14000 27323)
                    (make-color 32333 14000 27323))
T
```

MCL expressions governing color

This section describes the MCL expressions that govern color.

`*color-available*` [*Variable*]

Description The `*color-available*` variable returns a value indicating whether the Macintosh computer on which Macintosh Common Lisp is running supports Color QuickDraw.

If the value of this variable is non-`nil`*, then* the Macintosh computer supports the Color QuickDraw command set. If 32-bit QuickDraw is available, its value is 32.

If the value of this variable is `nil`*, then* Color QuickDraw is not available.

This variable should never be changed by a program.

`make-color` [*Function*]

Syntax `make-color` *red green blue*

Description The `make-color` function returns an encoded color, with components *red*, *green*, and *blue*. The components should be in the range 0–65535. Each component is stored with an accuracy of ±255.

Arguments

red	The red component of the color. This should be an integer in the range 0–65535.
green	The green component of the color. This should be an integer in the range 0–65535.
blue	The blue component of the color. This should be an integer in the range 0–65535.

Example

Note that the color components change value as they are encoded and decoded.

```
? (make-color 32333 14000 27323)
8271466
? (color-values 8271466)
32256
13824
27136
```

`color-red` [*Function*]

Syntax `color-red` *color*

Description The `color-red` function returns the red component of *color* as an integer in the range 0–65535.

Argument *color* A color.

Example

```
? (color-red 8271466)
32256
? (color-red *purple-color*)
17920
```

color-green [*Function*]

Syntax `color-green` *color*

Description The `color-green` function returns the green component of *color* as an integer in the range 0–65535.

Argument *color* A color.

Example

```
? (color-green 8271466)
13824
? (color-green *purple-color*)
0
```

color-blue [*Function*]

Syntax `color-blue` *color*

Description The `color-blue` function returns the blue component of *color* as an integer in the range 0–65535.

Argument *color* A color.

Example

```
? (color-blue 8271466)
27136
? (color-blue *purple-color*)
42240
```

color-values [*Function*]

Syntax `color-values` *color*

Description The `color-values` function returns three values corresponding to the red, green, and blue components of *color*.

Argument *color* A color.

Example

```
? (color-values 8271466)
32256
13824
27136
```

`real-color-equal` [*Function*]

Syntax `real-color-equal` *color1 color2*

Description The `real-color-equal` function returns true if *color1* and *color2* are displayed as the same color on the current display device. Otherwise it returns false.

This function may return different results for the same arguments, depending on the current configuration of the computer running Macintosh Common Lisp. For information on the algorithm used to map RGB colors into Macintosh color-table entries, see *Inside Macintosh.*

Arguments

color1	A color.
color2	Another color.

Example

```
? (real-color-equal (make-color 32256 14000 27323)
                    (make-color 32333 14000 27323))
T
```

`color-to-rgb` [*Function*]

Syntax `color-to-rgb` *color* `&optional` *rgb*

Description The `color-to-rgb` function returns a Macintosh RGB record describing the same color as *color*. RGB records are allocated on the Macintosh heap and are therefore not subject to garbage collection. They must be explicitly deallocated with a call to `dispose-record` or `#_DisposPtr`. For this reason, it is recommended that the macro `with-rgb` be used instead whenever possible.

Most Color QuickDraw traps receive colors in the form of RGB records.

Arguments *color* A color.

rgb — A macptr to an RGB record. The record may be combined with *color* to produce the returned record. (For information on macptrs see Chapter 15: Low-Level OS Interface.)

Example

```
? (color-to-rgb 8271466)
#<A Mac Zone-Pointer  Size 6 #x611930>
? (print-record * :rgbcolor)
#<Record :RGBCOLOR :RED 32256 :GREEN 13824 :BLUE 27136>
```

But it is preferable to use `with-rgb`:

```
? (let ((color 8271466))
       (when *color-available*
         (with-rgb (rec color)
           (print-record rec :rgbcolor))))
#<Record :RGBCOLOR :RED 32256 :GREEN 13824 :BLUE 27136>
```

---

rgb-to-color [*Function*]

Syntax `rgb-to-color` *rgb-record*

Description Given an RGB record, the `rgb-to-color` function returns a corresponding color encoded as an integer.

Most Color QuickDraw traps receive colors in the form of RGB records.

Argument *rgb-record* — An RGB color record stored on the Macintosh heap.

Example

```
? (make-record :rgbcolor
   :red 1000
   :green 2000
   :blue 3000)
#<A Mac Zone-Pointer Size 6 #x611940>
? (rgb-to-color *)  ;*=the last value returned
198411
? (color-values *)
768
1792
2816
```

with-rgb [*Macro*]

Syntax `with-rgb` (*variable color*) {*form*}*

Description The `with-rgb` macro evaluates *form* with *variable* bound to an RGB record corresponding to the color *color*. When the body of the macro exits, the RGB record is automatically disposed of.

Most Color QuickDraw traps receive colors in the form of RGB records.

Arguments

variable	A symbol bound to an RGB record for the duration of the macro. This position in the macro call is not evaluated.
color	A color encoded as an integer. This position in the macro call is evaluated.
form	Zero or more forms to be executed with *variable* bound to an RGB record containing *color*.

Example

This macro is useful because it saves the trouble of having to allocate RGB records explicitly. (Remember, RGB records are allocated on the Macintosh heap, and so they are not subject to garbage collection.)

```
? (defmethod set-fore-color ((window window) color)
    (when *color-available*
      (with-rgb (rec color)
        (with-port (wptr window)
          (#_rgbforecolor :ptr rec)))))
#<Method SET-FORE-COLOR (WINDOW T)>
```

user-pick-color [*Function*]

Syntax `user-pick-color &key :color :prompt :position`

Description The `user-pick-color` function displays the standard Macintosh Color Picker at `:position`, set to color `:color`, with prompt `:prompt`. It returns the selected color if the user clicks OK or throws to the tag `:cancel` if the user clicks Cancel.

Arguments

`:color`	The default color to bring up in the dialog box. The default is `*black-color*`.
`:prompt`	The prompt to display in the dialog box. The default is `"Pick a color"`.
`:position`	The position of the Color Picker on screen. The default is calculated by Macintosh Common Lisp.

`*black-color*` [*Variable*]

`*white-color*` [*Variable*]

`*pink-color*` [*Variable*]

`*red-color*` [*Variable*]

`*orange-color*` [*Variable*]

`*yellow-color*` [*Variable*]

`*green-color*` [*Variable*]

`*dark-green-color*` [*Variable*]

`*light-blue-color*` [*Variable*]

`*blue-color*` [*Variable*]

`*purple-color*` [*Variable*]

`*brown-color*` [*Variable*]

`*tan-color*` [*Variable*]

`*light-gray-color*` [*Variable*]

`*gray-color*` [*Variable*]

`*dark-gray-color*` [*Variable*]

Description These variables contain colors corresponding to the 16 colors available by default on a Macintosh computer with a 16-color monitor.

`*black-rgb*` [*Variable*]

`*white-rgb*` [*Variable*]

Description These variables contain RGB records for black and white.

Operations on color windows

The following operations are used to set the foreground and background colors of windows. If the computer display does not support colors, the colors do not appear. However, they remain associated with the windows, and if the same window is moved to a color monitor, they appear in the proper colors.

Windows created with an omitted or null `:color-p` initarg can display only eight colors. Specify `:color-p` as true to use the full range of colors supported by the hardware.

set-fore-color [*Generic function*]

Syntax `set-fore-color` (*window* `window`) *color*

Description The `set-fore-color` generic function sets the foreground color of the window to *color* and returns `nil`. Future drawing in the window appears in this color; when the window is redrawn, all drawing appears in this color.

Arguments

window	A window.
color	A color.

Example

```
? (setq mywin (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4BEE99>
? (set-fore-color * *blue-color*)
NIL
```

set-back-color [*Generic function*]

Syntax `set-back-color` (*window* `window`) *color* `&optional` *redisplay-p*

Description The `set-back-color` generic function sets the background color of the window to *color* and returns `nil`.

Arguments

window	A window.
color	A color.
redisplay-p	If the value of this is true (the default), this function invalidates the window, forcing a redrawing. The displayed background color does not change unless the window is redrawn.

Example

```
? (set-back-color mywin *yellow-color* t)
NIL
```

---

with-fore-color [*Macro*]

Syntax `with-fore-color` *color* {*form*}*

Description The `with-fore-color` macro sets the foreground color of the window to *color* and executes *form*. When the body of the macro exits, the old foreground color is restored.

This macro should be used only with a port set. That is, it should be used within the dynamic extent of a call to `with-port` or `with-focused-view`.

If Color QuickDraw is not present or *color* is `nil`, the color is not set.

Arguments

color	A color.
form	Zero or more forms to be executed with the foreground color set.

Example

```
? (setq my-new-win (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4D1399>
? (defmethod type-in-color ((view view) color string)
    (with-focused-view view
      (with-fore-color color
        (princ (format nil "~s" string) view))))
#<STANDARD-METHOD TYPE-IN-COLOR (VIEW T T)>
? (type-in-color my-new-win *blue-color* "Hi there")
NIL
```

---

with-back-color [*Macro*]

Syntax `with-back-color` *color* {*form*}*

Description The `with-back-color` macro sets the background color of the window to *color* and executes *form*. When the body of the macro exits, the old background color is restored.

This macro should be used only with a port set. That is, it should be used within the dynamic extent of a call to `with-port` or `with-focused-view`.

If Color QuickDraw is not present or *color* is `nil`, the color is not set.

Arguments

color	A color.

form	Zero or more forms to be executed with the background color set.

Coloring user interface objects

Methods on the following functions are used for setting the colors of user interface objects such as windows, dialog items, menus, and menu items. This section assumes some familiarity with the use of these classes.

For each class, a set of keywords identifies the parts that can be colored. The keywords for the different classes are given in the next section, "Part Keywords."

If a user defines a new class of dialog items, the generic function `view-draw-contents` can be defined to use the colors of the parts of the dialog item.

`part-color` [*Generic function*]

Syntax `part-color` *object part*

Description The `part-color` generic function returns the color of the part of *object* indicated by *part.*

Arguments

object	A user interface object. Built-in methods are defined for `window`, `dialog-item`, `menubar`, `menu`, and `menu-item.`
part	A keyword associated with the class of object. The part keywords are described in the next section.

`set-part-color` [*Generic function*]

Syntax `set-part-color` *object part color*

Description The `set-part-color` generic function sets the part of *object* indicated by *part* to *color* and returns *color*, encoded as an integer. If *color* is `nil`, the default color is restored.

Arguments

object	A user interface object. Built-in methods are defined for `window`, `dialog-item`, `menubar`, `menu`, and `menu-item.`

	part	A keyword associated with the class of object. The part keywords are described in the next section.
	color	A color.

---

part-color-list [*Generic function*]

Syntax `part-color-list` *object*

Description The `part-color-list` generic function returns a property list of keywords and colors for all the colored components of *object*. The same keywords apply as for `part-color`. Components whose color has not been set are not included.

Argument *object* A user interface object. Built-in methods are defined for `window`, `dialog-item`, `menubar`, `menu`, and `menu-item`.

Example

Here is an example of the use of part keywords with these functions:

```
? (setf w (make-instance 'window))
#<WINDOW "Untitled" #x3E9229>
? (part-color w :content)
NIL
? (set-part-color w :content *blue-color*)
212
? (part-color w :content)
212
? (part-color-list w)
(:CONTENT 212)
```

Part keywords

You can perform color operations on six objects: menu bars, menus, menu items, windows, dialog items, and table dialog items. This section presents the keywords that identify which parts of certain objects can be colored.

Menu bar

To perform color operations on the menu bar, use the value of the variable `*menubar*`, which contains the one instance of the class `menubar`. You can color the menu bar's titles and its background using the following keywords:

`:default-menu-title`
: The default color used for the titles of menus in the menu bar.

`:default-menu-background`
: The default color used for the background of the menus in the menu bar.

`:default-menu-item-title`
: The default color used for the titles of menu items in the menu bar.

`:menubar`
: The background color of the menu bar.

Menus

You can color three parts of menus.

`:menu-title`
: The color used for the title of the menu.

`:menu-background`
: The color used for the background of the menu.

`:default-menu-item-title`
: The default color used for the titles of menu items in the menu.

Menu items

You can color three parts of menu items.

`:item-title`
: The color used for the title of the menu item.

`:item-key`
: The color used for the command key of the menu item.

`:item-mark`
: The color used for the check mark beside the menu item.

Windows

The window part keywords correspond to different features in different types of windows, because the Macintosh Toolbox uses window color records differently for different window types. You can color windows using these keywords.

`:content` The color used for the background of the window.

`:frame` The color used for the outline of the window and the title bar of :tool windows.

`:text` The color used for the title of :document windows.

`:hilite` The color used for the lines in the title bar of `:document` windows.

`:title-bar`
The color used for the background of the title bar in :document windows and the title in `:tool` windows.

Dialog items

These part keywords work for built-in dialog items (although not all dialog items have all of these features). You may wish to use the part colors in the `view-draw-contents` method for dialog item classes you define.

`:frame` The color used for the outline of the dialog item.

`:text` The color used for the text of the dialog item.

`:body` The color used for the body of the dialog item.

`:thumb` The color used for the scroll box of the dialog item. (Scroll bars are the only built-in dialog item that have a scroll box.)

Table dialog items

The color of individual table cells can be set and accessed. Simply use the cell coordinates as the part keyword. For example, `(set-part-color my-table #@(0 0) 212)` sets the cell in the upper-left corner of the table to blue (which is encoded as `212`).

These colors are used only by the default `draw-cell-contents` function. If you define your own `draw-cell-contents`, you must use `part-color` to access and install the color when you draw the cell.;

Chapter 7:

The Interface Toolkit

Contents

The Interface Toolkit is an application built on top of Macintosh Common Lisp. It is provided as source code in the Interface Tools folder distributed with Macintosh Common Lisp; you can examine and modify it for your own use. It is also useful for building interfaces, and that aspect of it is documented here.

The Interface Toolkit does two things: edits menus and menu bars, and creates and edits dialog boxes. In addition, it prints source code for everything it creates.

You do not need to be familiar with the MCL implementation of menus and dialog boxes before using the Interface Toolkit. However, you should read Chapter 3: Menus, Chapter 4: Views and Windows, and Chapter 5: Dialog Items and Dialogs before working with the source code generated by the interface toolkit.

The Interface Toolkit

The Interface Toolkit, built on top of Macintosh Common Lisp, is an example of a simple MCL application.

It does the following:

- It creates menu bars and populates them with menus.
- It creates and edits dialogs and dialog items.
- For everything it prototypes, it is able to print source code to a file. When you have developed something in the Interface Toolkit, you can save your work to a Fred file, then edit it.

The Interface Toolkit is supplied as source code in the Interface Tools folder. You are free to examine and modify this source code, to use this source code in developing your own applications, and to include it, as is or modified, within your applications.

Loading the Interface Toolkit

Perform these steps to load the Interface Toolkit.

1. **Open the file `make-ift.lisp` and execute its contents.**

   In the Listener, choose Open from the File menu.

   Select the file `make-ift.lisp` from the Interface Tools folder.

   Execute its contents by choosing Execute Buffer from the Lisp menu.

2. **Type the following to the Listener, or execute it in a Fred window:**

   ```
   (ift::load-ift)
   ```

   This function loads the files that make up the Interface Toolkit.

Now your menu bar has one additional menu, the Design menu (Figure 7-1).

■ **Figure 7-1** The Interface Toolkit menu on the menu bar

Editing menus with the Interface Toolkit

In the Interface Toolkit you can edit the default menu bar or another menu bar to contain any menus you want. You can add menus to a menu bar and remove them. In the same way, you can add menu items to a menu or remove menu items from a menu. You can use menu items from the menus on the standard menu bar or make your own menu items.

You edit both menus and menu items by double-clicking them and specifying their attributes in an edit window.

More than one menu bar may be active, and you may edit more than one menu bar at once. You can cut and paste menus among menu bars, including the default menu bar, just as you would cut and paste text from one buffer to another.

At any time, you can generate source code for a menu or for the entire menu bar.

Using the menu editing functionality

After you load the Interface Toolkit, choose Edit Menubar, the first menu item on the Design menu (Figure 7-2). With this menu item you will edit menus and the menu bar.

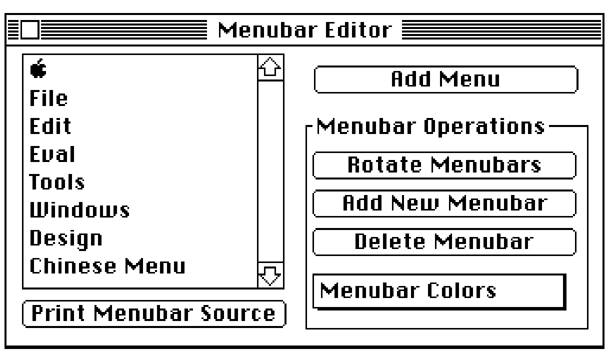
■ **Figure 7-2** Choosing Edit Menubar from the Design menu

When you choose Edit Menubar from the Design menu, the Interface Toolkit creates two windows, a small floating window and an editor window titled "Menubar Editor".

The floating window contains the standard editor commands Cut, Copy, Paste, and Clear. You can use this floating window to cut, copy, paste, and clear in situations where you don't have a working Edit menu.

The Menubar Editor window, shown in Figure 7-3, contains an editable list of the items in the current menu bar.

■ **Figure 7-3** The Menubar Editor window



The Menubar Editor window also contains the options listed in Table 7-1.

■ **Table 7-1** Menubar Editor window options

Option	Effect
Add Menu	Adds a new, empty menu named "Untitled" to the current menu bar (the one visible in the Menubar Editor's editable list and at the top of the screen).
Rotate Menubars	If more than one menu bar is active, makes the next menu bar the current menu bar. If only one menu bar is active, this command does nothing.
Add New Menubar	Adds a new, empty menu bar named "Untitled" to the active menu bars. The new menu bar initially contains only the Apple menu.
Delete Menubar	Deletes the current menu bar. The next active menu bar becomes the current menu bar.
Menubar Colors	Sets the colors of the menu bar.
Print Menubar Source	Creates a new Fred window containing the source code for the current menu bar.

Creating a new menu bar: Add New Menubar

To create a new menu bar, choose Add New Menubar from the Menubar Editor window. A new menu bar appears in the Menubar Editor window and at the top of the screen. This new menu bar initially contains only the Apple menu.

You can create any number of new menu bars.

Getting back to the default menu bar: Rotate Menubars

To get to another menu bar or back to the default menu bar, choose Rotate Menubars from the Menubar Editor window.

Deleting a menu bar: Delete Menubar

To delete a menu bar, choose Delete Menubar from the Menubar Editor window. This command deletes the currently installed menu bar and removes it from the rotation.

Creating and editing menus: Add Menu

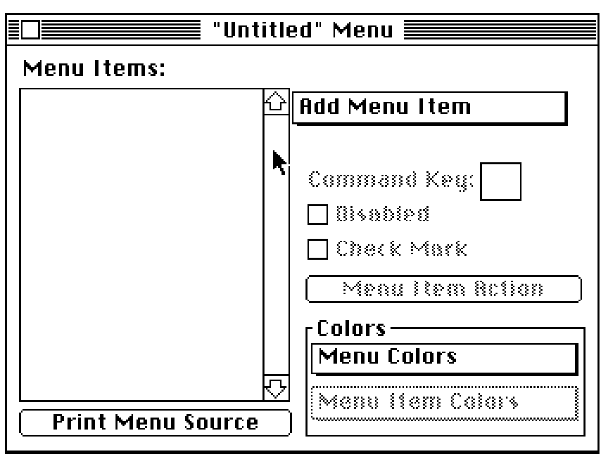
To create a menu, choose Add Menu from the Menubar Editor. The name of the new menu, “Untitled”, appears in the editable list and in the menu bar at the top of the screen.

You can change the name of any menu by choosing it and editing its text. To edit a menu, double-click its name in the list.

Creating menu items

Double-clicking the name of a menu creates a new Menu Editor window for menu items, as shown in Figure 7-4. This window contains an editable list of menu items, which will be empty if the menu is new, and the options listed in Table 7-2.

■ **Figure 7-4** A Menu Editor window showing a menu with no items



■ **Table 7-2** Menu editing options

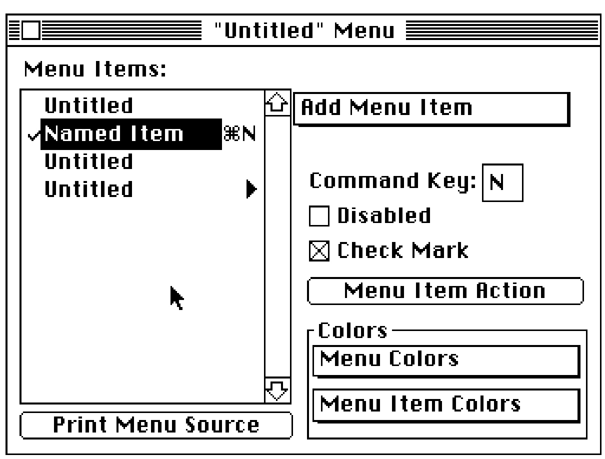
Option	Effect
Add Menu Item	Adds a new, empty menu item named "Untitled" to the current menu. There are three classes of menu items: `menu-item`, a menu item that represents a command; `menu`, a menu item that opens a menu; and `window-menu-item`, a window menu item. (See Chapter 3: Menus.) The `menu-item` class defaults to `menu-item`. To change it, edit the menu item source code. You can add further classes by editing the Interface Toolkit source code.
Menu Colors	Sets the colors for parts of the menu.
Print Menu Source	Opens a new Fred window and prints the source code for the menu to it.

Editing menu items

When you add menu items to a menu, you can edit them by double-clicking them, as in Figure 7-5.

Double-clicking a menu item lets you set the features listed in Table 7-3.

■ **Figure 7-5** Editing items in the Menu Editor



■ **Table 7-3** Menu item editing options

Option	Effect
Command key	Specifies the command key, if any, associated with the menu item.
Disabled	Specifies whether the menu item is disabled. The default is `nil`.
Check Mark	Specifies whether the menu item has a check mark beside it. The default is `nil`.
Menu Item Action	Brings up a Fred window in which you can write or edit code for the menu item action.
Menu Item Colors	Sets the menu item colors.

Saving a menu bar

When you are satisfied with your menu bar, choose Print Menu Source to create source code. Edit your source code as you like, then save it to a file for future use.

The definitions of some menu items in the standard menu bar must be edited. See the next section.

Editing menu bar source code

The Menu Editor is able to print source code for a menu item only if it has access to the source code of the action function of the menu item. If it doesn't, it puts `"Can't find definition"` in the place of the action function source code. You can then edit the code, putting in the real action function definition.

The source code for an action function is available if it was entered directly
from the menu editor or loaded from a source file with `*save-definitions*`
set to `t`.

It is not available if the menu was loaded from a `fasl` file unless the `fasl` file was compiled with a true value for the `:save-definitions` argument to `compile-file`.

The source code for the action functions of some of the built-in menu items is not available. For example, if you print the source code for the File menu, you need to edit the definition of the New menu item. The definition should make an instance of whatever kind of window you want New to use; for example, if New opens a Fred window, as it does in Macintosh Common Lisp, the definition you add is `(make-instance 'fred-window)`.

You should also delete `INTERFACE-TOOLS::W` from the argument list of the anonymous function.

If you are customizing your MCL menu bar, you may also need to edit the definitions in Table 7-4.

■ **Table 7-4** Menu items and corresponding MCL codes

Menu item	MCL code
New	Appropriate code to make an instance of the desired type of window.
Load File	`(load (choose-file-dialog))`
Compile File	`(compile-file (choose-file-dialog:button-string "Compile"))`
Break	`(break)`
Restarts	`(ccl::choose-restart)`
Edit Menubar	`(interface-tools::edit-menubar)`

Editing dialogs with the Interface Toolkit

The Interface Toolkit includes a quick interface designer for dialogs. With it you can create a blank dialog box with any set of attributes you want. Then, from a palette of buttons, radio buttons, checkboxes, editable-text dialog items, tables, and static text, you can drag in dialog items. You can edit them by double-clicking them. In an edit window you can specify the attributes of the dialog item, such as color, font, and associated action.

- *Note:* You can edit the palette to add your own items by editing its source code in the file `item-defs.lisp`, in the Interface Tools folder.

  At any time you can generate source code for the dialog box and its items.

- *Note:* When Design Dialogs is checked on the Interface Toolkit's special Design menu, *all* dialog boxes are editable, including the Search/Replace dialog box, the Environment dialog box, and so on. To use dialog boxes rather than edit them, choose Use Dialogs from the Design menu. (If you are in the middle of editing a dialog box, your edits will not disappear; the box will simply become usable.)

Using the dialog-designing functionality

First load the Interface Toolkit according to the directions in "Loading the Interface Toolkit" on page 264.

You see a new menu bar at the top of your screen, containing a Design menu. It should look like the one in Figure 7-2.

Dialog-designing menu items

The Interface Toolkit menu contains eight items, seven of which relate to dialog design (see Table 7-5).

■ **Table 7-5** Dialog design menu items

Option	Effect
Edit Menubar	Creates an editor window for the menu bar. This functionality is discussed in "Editing menus with the Interface Toolkit" on page 265.
Use Dialogs	Allows you to use dialog boxes in your MCL environment. Choosing this menu item automatically disables Design Dialogs, discussed next. These two menu items are the on/off stages of a single toggle. Turning on one turns off the other. When you first load the Dialog Designer, Use Dialogs is enabled. When you are using ordinary MCL dialogs, make sure Use Dialogs is enabled.
Design Dialogs	Allows you to design dialogs in your MCL environment. Choosing this menu item automatically disables Use Dialogs and makes all dialogs editable, but not usable. (As long as you are in the Interface Toolkit, you can switch back and forth between these modes at will.)
New Dialog...	Brings up a dialog box in which you can specify the type and attributes of a new dialog box. This menu item is discussed in the next section, "Creating Dialog Boxes."
Add Horizontal Guide	Adds a dotted horizontal guideline to the dialog box. This guideline becomes invisible when you choose Use Dialogs. This menu item is enabled only when you are creating or editing a dialog box.
Add Vertical Guide	Adds a dotted vertical guideline to the dialog window. This guideline becomes invisible when you choose Use Dialogs. This menu item is enabled only when you are creating or editing a dialog box.
Edit Dialog	Allows you to specify the title and position of the window that contains the dialog items. This menu item is enabled only when you are creating or editing a dialog box.

Creating dialog boxes

To create a dialog box, first make sure that a check appears next to Design Dialogs. Then choose New Dialog from the Design menu. The system displays a dialog box (Figure 7-6) in which you select the type and attributes of the dialog box you want to create.

■ **Figure 7-6** New Dialog dialog box

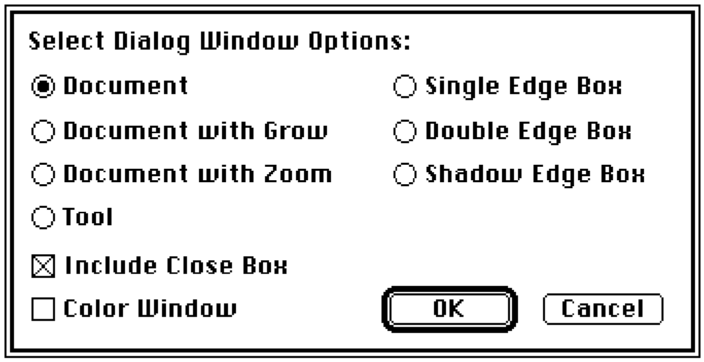


Table 7-6 lists the seven possible types of dialog.

■ **Table 7-6** Seven types of dialog

Option	Effect
Document	This is the default. Creates a dialog box with square corners and the title “Untitled Dialog.” By default, a document dialog box includes a close box.
Document with Grow	Creates a document dialog box with a size box.
Document with Zoom	Creates a document dialog box with a size box and a zoom box.
Tool	Creates a dialog box with rounded edges, a solid title bar, and the title “Untitled Dialog.” By default, it also includes a close box.
Single Edge Box	Creates a box with square corners, no title, and no close box. (You must put a close button within a dialog of this type.) Its edge is a single line.
Double Edge Box	Creates a box with square corners, no title, and no close box. Its edge is a double line.
Shadow Edge Box	Creates a box with square corners, no title, and no close box. Its edge is shadowed.

Two attributes are available (see Table 7-7).

■ **Table 7-7** Two attributes of dialog boxes

Option	Effect
Include Close Box	Includes a close box in your dialog window. The default value is true.
Color Window	Builds your dialog on top of a Macintosh `CWindowRecord` record.

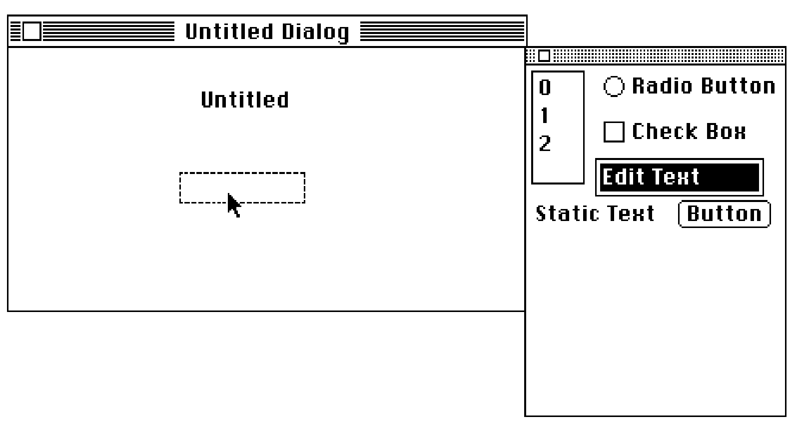
Adding dialog items

Whenever you change from the Use Dialogs menu item to the Design Dialogs menu item, you open a palette of dialog items. If you don't see this palette, choose Use Dialogs, then choose Design Dialogs again. The palette will appear.

The palette contains one of each type of dialog item: a table, a radio button, a checkbox, a field of editable text, some static text, and a button. In Figure 7-7, the palette appears to the right of the new dialog box.

Add dialog items to your dialog box by dragging them from the palette. The original dialog item will remain on the palette, and a copy with the title "Untitled" will appear in your dialog box. Figure 7-7 shows an editable-text dialog item being dragged from the palette to the dialog.

■ **Figure 7-7** Dragging an editable-text dialog item into an untitled dialog box



Place dialog items in the dialog box by dragging them. If you want to move the item only vertically or only horizontally, hold down Shift when you drag the box.

To help you place the dialog items, you can add vertical or horizontal guidelines to your dialog box. Click Add Vertical Guide or Add Horizontal Guide in the Design menu. You can select and drag a guide to place it. If you place a dialog item with an edge near a guide, it automatically aligns with the guide.

To resize the display space of any item, first click the item once. Handles (small black boxes) appear around the item. Click the pointer on any of these handles, then drag the item by its handle until you are satisfied with the size.

Editing dialog items

Edit a dialog item by double-clicking it. A dialog box opens. The dialog box varies with the kind of dialog item being edited. Figure 7-8 shows a typical example.

■ **Figure 7-8** Edit Dialog Items dialog box

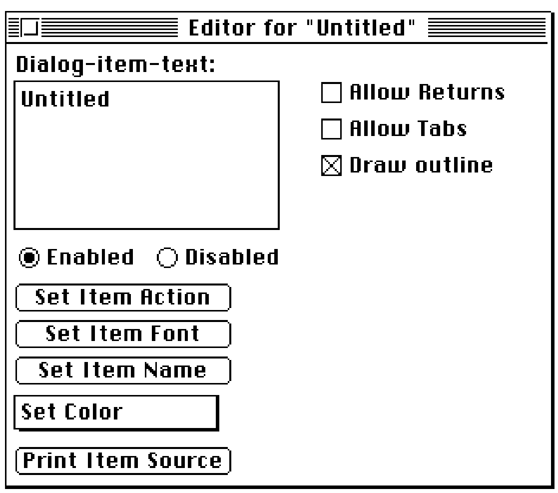


Table 7-8 lists the options available for editing dialog items.

■ **Table 7-8** Editable options in dialog items

Option	Effect
Dialog-item-text	Indicates `dialog-item-text`, the label or text the user sees. After you edit this text, you may have to change the size of the dialog item.
Enabled/Disabled	Sets whether the item is enabled or disabled. The default is enabled.
Set Item Action	Sets the code for the action performed by the dialog item.
Set Item Font	Sets the item font. The default is Chicago 12.
Set Item Name	Associates a nickname with the item.
Set Color	Colors one or more parts of the dialog item. You can color the frame, text, body, and thumb.
Print Item Source	Prints the dialog item source code to a new Fred window.

Most dialog item subclasses also allow you to edit special parameters associated with the subclass (see Table 7-9).

■ **Table 7-9** Editable options in subclasses of dialog items

Subclass and option	Effect
Radio buttons	
Radio Button Pushed	Indicates whether or not the radio button is selected when the dialog box is first displayed. The default is `nil`.
Set Item cluster	Allows you to move the radio button to a new cluster. Radio button clusters are numbered sequentially, starting with 0. To set the button's cluster, enter a new number.
Buttons	
Default Button	Indicates whether this is the default button. The default value is `nil`.
Edit-text dialog items	
Allow Returns	Indicates whether carriage returns are allowed in the Edit Text field. The default value is `nil`.
Allow Tabs	Indicates whether pressing the Tab key inserts a tab in the buffer or selects the next key handler in the dialog box. The default, `nil`, selects the next key handler.
Draw Outline	Indicates whether an outline is drawn around the dialog item.
Checkboxes	
Checkbox Checked	Indicates whether or not the checkbox is checked. The default value is `nil`.
Tables	
Set Cell Size	Allows you to set a new default size for table cells.
Horizontal Scroll Bar	Adds a horizontal scroll bar to the table.
Vertical Scroll Bar	Adds a vertical scroll bar to the table.
Set Table Sequence	Sets the sequence in which items appear in the table.
Set Wrap Length	Sets the maximum length a line of text can attain before wrapping to the next line occurs. The default value is `nil`; that is, lines are not wrapped.
Orientation	Determines whether the orientation of the table is vertical or horizontal. The default is vertical.

Chapter 8:

File System Interface

Contents

This chapter describes filename specification and the functions for manipulating the Macintosh File System. It does not document all Common Lisp file system features, but refers to *Common Lisp: The Language* where appropriate.

You should read this chapter to familiarize yourself with the specification of filenames in Macintosh Common Lisp. It is particularly important if you will deal with other file systems and must translate between them and the file system of Macintosh Common Lisp.

You should be familiar with Chapter 23 of the second edition of *Common Lisp: The Language,* which discusses the Common Lisp file system features.

Filenames, physical pathnames, logical pathnames, and namestrings

The file system interface provides a way of dealing with references to file systems when code may be running on multiple platforms. MCL code must deal with the file system requirements of the Macintosh Operating System and, if the code is meant to be ported, with those of any other operating system on which it is intended to run. Macintosh Common Lisp specifies filenames by means of pathnames, which can be specified as namestrings.

A **filename** is a means of specifying a particular file or directory in a file system. You can represent a filename as either a Lisp object (a pathname) or a string (a namestring). Internally, Macintosh Common Lisp always uses pathnames and converts namestrings to pathnames before using them.

- A **pathname** is a structured Lisp object. It represents a filename as a set of components that can be manipulated in an implementation-independent way. A pathname is not necessarily the name of a file; it is a specification, perhaps partial, of how to access a file.

  A single filename may be represented by two or more quite different pathnames, and the existence of a pathname does not guarantee that the file it specifies exists.
- There are two kinds of pathnames:

  A **physical pathname** indicates the physical components of the pathname.

  A **logical pathname** structure has one or more logical components. Logical components may be translated to their physical counterparts.
- A **namestring** is a string that names a file in any one of three syntaxes: Macintosh physical syntax, Common Lisp logical pathname syntax, or MCL logical directory syntax. (MCL logical directory syntax is now deprecated and is likely to disappear in a future release.)

The following sections discuss Macintosh physical syntax and Common Lisp logical pathname syntax. MCL logical directory syntax is described in "Logical directory names" on page 310.

Changes from earlier versions of Macintosh Common Lisp

If you have used versions of Macintosh Common Lisp prior to version 2.0, you should note an important change in the implementation of the file system.

As of version 2.0, Macintosh Common Lisp version uses logical hosts, bringing it into compliance with the file system interface design described in Chapter 23 of *Common Lisp: The Language*. This can be somewhat confusing, since the old MCL-specific system of logical directories is very similar to the new Common Lisp system of logical hosts. Under earlier versions of Macintosh Common Lisp, `"CCL"` (for example) was defined as a logical directory, and you could test for the presence of a file like this:

```
? (probe-file "ccl;MCL help")
```

In Macintosh Common Lisp version 2, `"CCL"` is defined as a logical host, and the syntax is very slightly different:

```
? (probe-file "ccl:MCL help")
```

If your application requires it, you can reproduce the old behavior by defining the logical directory yourself:

```
? (def-logical-directory "ccl"
      (full-pathname "ccl:"))
```

◆ *Note:* The MCL functionality previously called “logical pathnames” refers to the MCL-specific system of logical directories and is now called “logical directory names.” The logical pathname functionality discussed in this chapter refers to the file system interface design described in Chapter 23 of *Common Lisp: The Language.*

Printing and reading pathnames

Common Lisp now specifies that pathnames be printed and read using the `#P` syntax.

In Macintosh Common Lisp, pathnames are printed using the Common Lisp `#P` reader macro (see *Common Lisp: The Language,* pages 537 and 556), as shown in this example:

```
? (make-pathname :directory "hd" :name "foo")
#P"hd:foo"
```

Macintosh Common Lisp also has a numeric argument that specifies one of four possible unusual conditions in the pathname. .

`#1P` means that the type is `:unspecific`.

`#2P` means that the name is `""`.

`#3P` means that the type is `:unspecific` and the name is `""`.

`#4P` means that the namestring represents a logical pathname.

All other numeric arguments are illegal.

With this convention, Macintosh Common Lisp avoids the potential loss of information when converting between a pathname and a namestring:

```
(make-pathname :name "foo" :type "lisp")
#P"foo.lisp"
(make-pathname :name "foo" :type nil)
#P"foo"
(make-pathname :name "foo" :type :unspecific)
#1P"foo"
(make-pathname :name nil :type "lisp")
#P".lisp"
(make-pathname :name "" :type "lisp")
#2P".lisp"
(make-pathname :name nil :type nil)
#P""
(make-pathname :name nil :type :unspecific)
#1P""
(make-pathname :name "" :type nil)
#2P""
(make-pathname :name "" :type :unspecific)
#3P""
```

◆ *Note:* The numeric argument #*n*`P` is not a part of Common Lisp and may be removed in future releases of Macintosh Common Lisp.

Pathname structure

Common Lisp pathnames (Lisp data objects of type `pathname`) have six components: `host`, `device`, `directory`, `name`, `type`, and `version`.

Macintosh physical pathnames

On a Macintosh computer, filenames have only three components: `directory`, `filename`, and an optional `type`. Macintosh filenames can be translated into Common Lisp pathname structures; when they are, the `host`, `device`, and `version` components of the pathname are `:unspecific`.

The Macintosh physical pathname syntax has the following components:

`[:] {`*directory*`:}* [`*name*`] [.`*type*`]`

A Macintosh physical pathname may have multiple colons. The component of the string preceding its first delimiter does not name a logical host.

```
? (make-pathname :directory "Style&Design:Glossary:"
                 :name "frontmatter")
#P"Style&Design:Glossary:frontmatter"
```

Common Lisp logical pathnames

Common Lisp logical pathname syntax has the following components:

`[`*host*`:] [;] {`*directory*`;}* [`*name*`] [.`*type* `[.`*version*`]]`

In logical pathname syntax, the *host* and *directory* components are indicated by the characters to the left of the last colon or semicolon. Logical pathnames can be distinguished from physical pathnames by the following tests:

- The first delimiter between components is a colon.
- The first delimiter is the only colon.
- The string preceding the first delimiter names a defined logical host.

For example, the following is a Common Lisp logical pathname because the first delimiter between pathname components is a colon, it is the only colon, and `"CCL"`, the string preceding the first delimiter, names a defined logical host:

```
"CCL:Interface Tools;My Menus;custom-menu.lisp"
```

Defining logical hosts

By defining logical hosts, Macintosh Common Lisp is able to exchange logical pathnames conveniently and portably. When a logical host is a different file system, for example, one in which the length of filenames is restricted, logical hosts and logical pathname translations provide a necessary layer of abstraction. Logical hosts are also useful when moving software from one machine to another.

Macintosh Common Lisp will recognize a logical host only after it has been defined. To define a logical host, you create and execute a `setf` form to set `logical-pathname-translations` for the relevant host. You should do this for every file system with which you will interact. Here is a very simple example:

```
? (setf (logical-pathname-translations "home")
          `(("**;*.*" ,(merge-pathnames ":**:*.*"
                        (mac-default-directory)))))
```

When Macintosh Common Lisp is run, two logical hosts are set up automatically:

- The host `"ccl"` is set to the directory holding the MCL application.
- The host `"home"` is set to the directory holding the document that was launched with Macintosh Common Lisp.

After you define a logical host, you can inspect it by clicking Inspect on the Tools menu, then clicking Logical Hosts. This displays a list of all the logical hosts used and generated by Macintosh Common Lisp.

◆ *Note:* For a full discussion of logical pathname namestrings and their syntax, see *Common Lisp: The Language*, pages 628–629. For information on the philosophy and use of `logical-pathname-translations`, see pages 636–637.

Ambiguities in physical and logical pathnames

In Macintosh Common Lisp, the colon is both the host delimiter in logical pathname syntax and the device/directory delimiter in physical pathname syntax. This can cause ambiguity. For example, in the namestring `"bar:foo.lisp"`, `"bar"` can be either a logical host or a top-level physical directory.

If you have both a top-level physical directory and a logical host with the same name, there is a possibility of ambiguity. For this reason it is advisable not to give a physical device and a logical host the same name.

If you have a name conflict, you should do one of the following:

- Rename one.
- Use the special escape character, `#\∂` (Option-D) to quote the colon after the directory name of the physical pathname; this indicates that the pathname is physical. The escape character is documented in "The pathname escape character" on page 289.
- Create the physical pathname with the function

  `(make-pathname :directory '(:absolute` *namestring*`))`
- where *namestring* is the namestring of the physical directory.

More on namestrings and pathnames

Types may be specified as part of the filename; for instance, you generally specify the type of an uncompiled file of Lisp source code by giving it the type `.lisp`, and compiled source code by giving it the type `.fasl`.

All functions that accept pathnames as arguments also accept namestrings, converting them to pathnames before using them. It is seldom necessary to use (`pathname "hd:foo"`). Instead, you can use `"hd:foo"`. However, if the pathname is going to be parsed repeatedly, you should use the `pathname` syntax; that is, the value of `*default-pathname-defaults*` should be a pathname, not a string. (See the documentation of `*default-pathname-defaults*` in *Common Lisp: The Language.*)

The Common Lisp function `parse-namestring` converts a namestring to a pathname. The Common Lisp function `namestring` converts a pathname to a string. You can create a pathname directly by specifying its components using the Common Lisp function `make-pathname`.

Creating and testing pathnames

Common Lisp provides several functions to create pathnames and to test whether an object is a pathname. You can create a pathname directly, merge a pathname with default components, and retrieve components of a pathname.

Full documentation of most of these functions appears in Chapter 23, "File System Interface," of *Common Lisp: The Language*, and they are not redocumented here. Only the following function shows special behavior in Macintosh Common Lisp.

`make-pathname` [*Function*]

Syntax

```
make-pathname &key :host :device :directory :name
      :type :version :defaults :case
```

Description

The Common Lisp function `make-pathname` constructs and returns a pathname. After the components specified by the `:host`, `:device`, `:directory`, `:name`, `:type`, and `:version` arguments are filled in, missing components are taken from the `:defaults` argument. The Macintosh Operating System does not support hosts, devices, or versions, so Macintosh Common Lisp recognizes only logical hosts. In Common Lisp, a logical host is a string that has been defined as a logical pathname host using `setf` and `logical-pathname-translations`. (See page 632 of *Common Lisp: The Language* for a discussion of how this is done.)

Arguments

`:host` Specifies the host component. The `:host` argument determines whether a pathname is physical or logical. If the `:host` argument is `:unspecific`, or if it is omitted and the `:defaults` argument is a physical pathname, then a physical pathname is created. Otherwise the `:host` argument must be `nil` or a string, and a logical pathname is created.

`:device` Specifies the device component. Because the Macintosh computer does not support devices, this argument is ignored and `pathname-device` always returns `:unspecific`.

`:directory` Specifies the directory component. The value of the `:directory` argument is `nil`, `:wild`, `:wild-inferiors`, *string*, or *list*.

- nil Specifies that the directory component should be taken from the defaults.
- `:wild` Specifies the wildcard `"*"`.
- `:wild-inferiors` Specifies the wildcard `"**"`.
- *string* A string, which may be a wildcard or empty, and which may end in a colon or semicolon. Unless the `:host` argument is a logical host, Macintosh Common Lisp interprets a string argument with colons or semicolons as a Macintosh-syntax directory namestring.
- *list* A list beginning with either `:absolute` or `:relative` followed by the individual directory component strings.

`:name`	Specifies the name component. The value of the `:name` argument is `nil`, `:wild`, or *string*.
`nil`	Specifies that the name component should be taken from the defaults.
`:wild`	The wildcard `"*"`.
string	A string, which may be a wildcard or empty. Quoted colons are allowed in the `:name` component, but they cause an error when they are passed to the Macintosh File System.
`:type`	Specifies the type component. Its value is `nil`, `:wild`, or *string*.
`nil`	Specifies that the type component should be taken from the defaults.
`:wild`	The wildcard `"*"`.
string	A string, which may be a wildcard or empty. Quoted colons are allowed in the `:type` component, but they cause an error when they are passed to the Macintosh File System.
`:version`	Ignored unless the `:host` argument is a logical host. For logical pathnames, the value of the `:version` argument may be `nil`, `:unspecific`, `:wild`, `:newest`, or a positive integer.
`nil`	Specifies that the version component should be taken from the default.
`:unspecific`	Indicates whether the version number is unspecified.
`:wild`	The wildcard `"*"`.
`:newest`	The newest version.
integer	A positive integer representing the version number. Currently Macintosh Common Lisp allows only `0`.
`:defaults`	Specifies which defaults to use. The default value of the `:defaults` argument is a pathname whose host component is the same as the host component of `*default-pathname-defaults*` and whose other components are all `nil`.
`:case`	Determines how character case is treated. The value of `:case` may be `:common` or `:local`. A full description of `:case` is given in *Common Lisp: The Language*, starting on page 617.

Full documentation of `make-pathname` is given in *Common Lisp: The Language*, on page 643.

Parsing namestrings into pathnames

The MCL pathname parser uses the following rules to break namestrings into their components.

- Unspecified components are given the value `nil`. Neither defaults nor logical directory names are merged at parse time, with the exception of the `:host` component of `*default-pathname-defaults*`. The function `merge–pathnames` merges one pathname with another by replacing `nil` components of its first argument with corresponding components of its second argument. The function `full-pathname` performs the logical-to-physical pathname translation.
- The `:directory` component is identified as the characters from the end of the host component to the last colon or semicolon. The colon is the standard Macintosh separator character for directories. The semicolon is the separator for logical directory names. A directory name that begins with a colon is relative to the Macintosh default directory.
- The `:name` component is identified as the characters that follow the directory component until either the end of the string or the beginning of the type component. The period between the name and the type component is only a separator and is not part of the `:name` component. To make a name containing a period, use the escape character (see the next section, "The Pathname Escape Character"). To specify a file that has an empty string as its name, use a single period after the directory separator character.
- The `:type` component is composed of the characters from the name component to either the version component or the end of the string.
- The `:version` component, if present, is always either `.newest` or `0`. It is the last component before the end of the string.

Table 8-1 contains some examples of namestring-to-pathname parsing.

■ **Table 8-1** Some namestrings parsed into pathnames

		Pathname components		
Namestring	**Host**	**Directory**	**Name**	**Type**
"hd:foo.lisp"		(:absolute "hd")	"foo"	"lisp"
"hd:"		(:absolute "hd")	nil	nil
"hd:."		(:absolute "hd")	""	nil
":foo"		(:relative)	"foo"	nil
"foo"		nil	"foo"	nil
"foo."		nil	"foo"	nil
"foo.fasl"		nil	"foo"	"fasl"
"hd:sub-dir:foo.text"		(:absolute "hd" "sub-dir")	"foo"	"text"
"sys:bar;foo.lisp"	"sys"	(:absolute "bar")	"foo"	"lisp"

The pathname escape character

If you need to use a colon, semicolon, period, or asterisk as part of a pathname, quote it with the special escape character, `#\∂` (Option-d). This escape character works very much like the backslash character in strings. Any character preceded by a ∂ loses any special meaning.

◆ *Note:* Asterisks must be quoted in physical pathnames, because Common Lisp mandates that functions such as `truename` and `open` must signal an error if given a wild pathname.

Table 8-2 illustrates the quoting mechanism in pathnames.

■ **Table 8-2** Effect of escape characters

	Pathname components		
Namestring	**Directory**	**Name**	**Type**
`"hd:foo.lisp"`	`(:absolute "hd")`	`"foo"`	`"lisp"`
`"hd:foo∂.lisp"`	`(:absolute "hd")`	`"foo∂.lisp"`	`nil`
`":fo∂o∂."`	`(:relative)`	`"foo∂."`	`nil`
`";ccl∂;foo"`	`(:relative)`	`"ccl∂;foo"`	`nil`
`"ccl;fo∂∂o"`	`(:absolute (:logical "ccl))`	`"fo∂∂o"`	`nil`
`"hd:fo\"o.lisp"`	`(:absolute "hd")`	`"fo\"o"`	`"lisp"`

Only the *needed* escape characters are retained (for example, the "∂" before the "o" in the third line is removed, but the "∂" before the period is retained). Of course, this mechanism is meant to work only for the MCL additions; you can specify a filename that includes a colon, but you cannot open such a file, because Macintosh computers do not accept filenames that contain colons.

◆ *Note:* The escape characters are not part of the true name. They are included only in the Lisp representation of the pathname, not in the Macintosh system's representation of the pathname.

The `make-pathname` function attempts to insert the appropriate escape characters in components that need them. The user need only insert escape characters in front of semicolons that are part of directory components, and in front of the character ∂. Here are some examples of the use of `make-pathname`.

```
? (make-pathname :directory "Hd:" :name "foo" :type "lisp")
#P"Hd:foo.lisp"
? (make-pathname :directory nil
                 :name "foo"
                 :type "fasl")
#P"foo.fasl"
? (make-pathname :directory nil :name "foo."
                 :type "fasl")
#P"foo∂..fasl"
? (make-pathname :directory "hd;"
                 :name "foo."
                 :type (pathname-type *.lisp-pathname*))
#P"hd;foo∂..lisp"
```

Loading files

The following functions and variables govern the loading of files. For Common Lisp functions governing the loading of files, see Section 23.4, "Loading Files," starting on page 657 of *Common Lisp: The Language*.

`*.lisp-pathname*` [*Variable*]

Description The `*.lisp-pathname*` variable contains the file type for MCL source code files. The initial value of this variable is `#P".lisp"`.

`*.fasl-pathname*` [*Variable*]

Description The `*.fasl-pathname*` variable contains the file type for MCL compiled files. The initial value of this variable is `#P".fasl"`.

`*pathname-translations-pathname*` [*Variable*]

Description The `*pathname-translations-pathname*` variable contains a pathname whose host is `:ccl` and whose type is the string `"pathname-translations"`.

`require` [*Function*]

Syntax `require` *module* `&optional` *pathname*

Description The `require` function was once a Common Lisp function but is now specific to Macintosh Common Lisp. It attempts to load the files in *module* if they have not already been loaded.

Arguments

module The name of the module.

pathname A pathname or list of pathnames indicating the files contained in the module.

There are three ways to tell `require` how to look for a module:

- If *pathname* is given, it should be a pathname or a list of pathnames whose files should be loaded in order, left to right.

- If *pathname* is not given, `require` first looks in the variable `*module-file-alist*`, which is bound to an association list. In this association list, the `car` of each element should be a module name, and the `cdr` of each element should be a pathname or list of pathnames making up the module. The `require` function loads all the files listed. Initially, `*module-file-alist*` is empty. Here is how to add something to `*module-file-alist*`.

```
? (push '(my-system . ("my-sys;definitions.fasl"
                       "my-sys;actions.fasl"))
       *module-file-alist*)
```

- If the module is not registered in `*module-file-alist*`, `require` looks for a file with the same name as the module name in the locations specified by the variable `*module-search-path*`. The `*module-search-path*` variable should be bound to a list of pathnames, each specifying the directory and possibly a file type (the name component is ignored and replaced by the name of the module). If no file type is given, both `*.lisp-pathname*` and `*.fasl-pathname*` are looked for, and the more recent file is used.

For example, `(push "ccl:misc;" *module-search-path*)` causes `(require 'tools)` to look for the file `ccl:misc;tools.fasl` or `ccl:misc;tools.lisp`, whereas `(push "ccl:misc;.fasl" *module-search-path*)` causes `(require 'tools)` to look for `ccl:misc;tools.fasl` before searching for other versions of the tools file. The initial value of `*module-search-path*` is `(#4P"ccl:" #4P"home:" #4P"ccl:library;" #4P"ccl:examples;")`.

Macintosh Common Lisp keeps a list of files currently being loaded. This helps ensure that files requiring each other do not cause infinitely recursive calls to `require`.

For documentation of the state of `require`, see *Common Lisp: The Language*, pages 277–278.

---

provide [*Function*]

Syntax `provide` *module*

Description The `provide` function was once part of Common Lisp but is now specific to Macintosh Common Lisp. It adds a new module name to the list of modules maintained in the variable `*modules*`, indicating that the module *module* has been provided.

For documentation of the state of `provide`, see *Common Lisp: The Language*, pages 277–278.

Argument *module* The name of the module.

Macintosh default directories

The Macintosh Operating System maintains a default directory of its own. Any namestring that begins with a colon or semicolon is relative. The directory component of a relative pathname is appended to the directory component of `*default-pathname-defaults*` before accessing the file system. If the resulting pathname is still relative, then the value of `mac-default-directory` is used.

◆ *Note:* Desk accessories and other background processes may change the default directory without notice. If you must access the Macintosh default directory, you should set it just before accessing it, or (preferably) specify a directory explicitly in file system calls.

The Macintosh default directory is initially the directory containing Macintosh Common Lisp.

mac-default-directory [*Function*]

Syntax `mac-default-directory`

Description The function `mac-default-directory` returns the Macintosh default directory.

Example

```
? (mac-default-directory)
#P"BigTowel:CCL:"
```

set-mac-default-directory [*Function*]

Syntax `set-mac-default-directory` *pathname*

Description The function `set-mac-default-directory` sets the Macintosh default directory to the directory component of *pathname.*

If the directory component of a pathname is empty, the Macintosh computer looks for the directory in the Macintosh default directory. To ensure that the Macintosh default directory is *not* used, specify the directory component of the pathname. (One way to do this is by specifying a merge with some other default.)

The default directory returned by `mac-default-directory` can change at any time; set it explicitly just before using it, or (preferably) specify a directory explicitly in file system calls.

Argument *pathname* A pathname, string, or stream associated with a file. If the directory specified by the pathname exists, `set-mac-default-directory` sets the Macintosh default directory to the directory component of *pathname*. If it does not exist, `set-mac-default-directory` returns `nil` and the Macintosh default directory is not changed.

Example

```
? (set-mac-default-directory #P"BigTowel:CCL Test:")
#P"BigTowel:CCL Test:"
```

---

mac-namestring [*Function*]

Syntax `mac-namestring` *pathname*

Description The `mac-namestring` function translates *pathname* from a logical to a physical pathname. If *pathname* is a logical pathname or a string describing a logical pathname, it is translated to a physical pathname. If *pathname* contains MCL logical directories, they are expanded. The function returns the physical pathname as a namestring. The function then prepares *pathname* for passing to the Macintosh File Manager by verifying that the namestring contains no wildcards or quoted colons and by removing all quoting. If *pathname* contains wildcards or quoted colons, an error is signaled.

Argument *pathname*A pathname or a string.

Example

```
? (mac-namestring "ccl:examples;dialog-editor.lisp")
"hd:myccl:examples:dialog-editor.lisp"
```

---

mac-directory-namestring [*Function*]

Syntax `mac-directory-namestring` *pathname*

Description The function `mac-directory-namestring` turns *pathname* into a namestring, expands all logical directories into physical directories, then prepares it for passing to the Macintosh File Manager by verifying that the namestring contains no wildcards or quoted colons and by removing all quoting. If *pathname* contains wildcards or quoted colons, an error is signaled. It returns only the directory component of the pathname as a string.

Argument *pathname*A pathname, string, or stream.

---

mac-file-namestring [*Function*]

Syntax `mac-file-namestring` *pathname*

Description The function `mac-file-namestring` turns *pathname* into a namestring, then prepares it for passing to the Macintosh File Manager by verifying that the namestring contains no wildcards or quoted colons and by removing all quoting. If *pathname* contains wildcards or quoted colons, an error is signaled. It returns only the part of the string excluding the directory specification (that is, the filename and file type).

Argument *pathname*A pathname, string, or stream.

Structured directories

Common Lisp provides a portable format for specifying directories, discussed in *Common Lisp: The Language*, starting on page 620. Macintosh Common Lisp follows that format, with the exception that the symbols `:up` and `:back` are equivalent in the current Macintosh File System.

The following function extends the Common Lisp function `directory`.

---

directory [*Function*]

Syntax `directory` *pathname* `&key :directories :files :directory-pathnames :test :resolve-aliases`

Description The `directory` function takes a pathname as its argument and returns a list of pathnames, one for each file in the file system that matches the given pathname.

You can use `directory` with any of the wildcards described in the next section. When you use wildcards, this function returns a list of the true names of all matching files in all matching directories. If no files match the specified pathname, `directory` returns `nil`.

Arguments

pathname A value. If the directory specified by the pathname exists, `directory` returns a list of pathnames of files included in that directory. If it does not, `directory` returns `nil`.

`:directories` An argument specifying whether to include directories in the returned list. The default value is `nil`.

`:files` An argument specifying whether to include files in the returned list. The default value is true.

`:directory-pathnames` An argument specifying whether to represent directory pathnames in the returned list as directories or files (`foo:baz:` or `foo:baz`). The default value is true, which means that they are represented as directories.

`:test` A test function to be applied to each matching pathname. The `:test` argument is called only if all the other conditions are satisfied.

`:resolve-aliases` An argument specifying whether to resolve aliases. If the value of `:resolve-aliases` is `:show-alias`, then aliases are resolved but the pathname returned contains the name of the alias rather than the name of the target. Any other non-`nil` value causes aliases to be resolved and the pathname returned to be that of the target. The default value is `nil`.

directoryp [*Function*]

Syntax `directoryp` *pathname*

Description The `directoryp` function returns the true name of the directory if *pathname* names a directory, `nil` if it names an ordinary file; otherwise it signals an error. (For true names, see *Common Lisp: The Language* under the function `truename`.)

Argument *pathname* A pathname or string.

full-pathname [*Function*]

Syntax `full-pathname` *pathname-or-namestring* `&key :no-error`

Description The `full-pathname` function returns a pathname whose logical components are all translated into physical components. If the function is called on a namestring, the namestring is first converted into a Lisp pathname. It can translate both Common Lisp logical pathnames and MCL logical directories (described in “Logical directory names” on page 310). The pathname is merged with `*default-pathname-defaults*`.

This function was formerly called `expand-logical-namestring`.

Arguments *pathname-or-namestring*
A pathname or namestring.

`:no-error` If the value of `:no-error` is `t` (the default) and there is no physical directory for a logical directory in *pathname*, Macintosh Common Lisp returns `nil`. If the value of `:no-error` is `nil`, Macintosh Common Lisp signals an error.

Example

This example creates a logical-to-physical mapping and gets its full pathname.

```
;Create the logical to physical mapping:
? (setf (logical-pathname-translations "misc")
        '(("**;**" "hd:ccl-misc:**.*.*")))
NIL
;Load the file "hd:ccl-misc:hacks.lisp":
? (load "misc:hacks.lisp")
;Loading "hd:ccl-misc:hacks.lisp"...
#P"hd:ccl-misc:hacks.lisp"

? (full-pathname "misc:hacks.lisp")
"hd:ccl-misc:hacks.lisp"
? (full-pathname "MISC:hacks.lisp")
"hd:ccl-misc:hacks.lisp"          ;Note case insensitivity.
```

directory-pathname-p [*Function*]

Syntax `directory-pathname-p` *pathname*

Description The `directory-pathname-p` function returns a Boolean value: `t` if *pathname* is a pathname specifying a directory, `nil` if it is not. A pathname is a directory pathname if its name is `nil` or the empty string and its type is `nil` or `:unspecified`.

Argument *pathname* A pathname, string, or stream.

Example

```
? (directory-pathname-p "ccl:foo;")
T
? (directory-pathname-p "ccl:foo")
NIL
? (directory-pathname-p "hd:ccl:")
T
? (directory-pathname-p "hd:ccl:init.lisp")
NIL
```

Wildcards

Macintosh Common Lisp supports two forms of wildcards. One is **extended wildcards** as specified in *Common Lisp: The Language*, pages 623–627. Extended wildcards do not depend on a specific wildcard syntax. If you plan to port your code over multiple file systems, use the Common Lisp extended wildcards.

You can also use the simpler wildcard system described here, which is compatible with previous versions of Macintosh Common Lisp.

The wildcards are used in the following ways:

- One asterisk matches zero or more characters in a component.
- One asterisk in place of a directory component matches one directory level.
- Two asterisks used in place of a directory match zero or more subdirectories at all levels of the parent directory.
- Two asterisks used in place of the filename components match any number of components that are left.

The following examples assume the existence of a mounted disk with the name `"hd"`.

- `(directory "hd:*:" :files nil :directories t)` returns a list of all subdirectories directly under `"hd:"`.
- `(directory "hd:**")` returns a list of files under `"hd:"`.

- `(directory "**:**:" :directories t :files nil)` returns a list of all the subdirectories at all levels in all the devices known to the machine.
- `(directory "**:**")` returns a list of all the files at the top level in all the devices known to the machine.
- `(directory "hd:*.lisp")` returns a list of all the files in the top level of `"hd:"` that are of type `"lisp"`.
- `(directory "**:ccl:*:*:prin*12.**")` returns a list of all the files in any device that start with the letters `"prin"` and end in `"12"` and are two levels below a directory named `"ccl:"`.

File and directory manipulation

The functions in this section operate on both directories and files. A directory operation is performed if the filename component is empty (that is, if the pathname ends in a colon or semicolon); otherwise, a file operation is performed.

The functions operate on Lisp pathnames, strings, and streams.

delete-file [*Function*]

Syntax `delete-file` *pathname* `&key :if-does-not-exist`

Description This extension of the Common Lisp function `delete-file` deletes the specified *pathname*.

Arguments *pathname* A pathname.

`:if-does-not-exist`
A keyword that can take the value `nil` or `:error`. If *pathname* does not exist and the value of `:if-does-not-exist` is `nil` (the default), Macintosh Common Lisp returns `nil`. If it is `:error`, Macintosh Common Lisp signals an error.

create-file [*Function*]

Syntax `create-file` *pathname* `&key :if-exists :mac-file-type :mac-file-creator`

Description The `create-file` function creates an empty file or a directory named *pathname* and returns the `truename` of the created file or directory. If necessary, `create-file` creates missing intermediate directories.

The `:mac-file-type` and `:mac-file-creator` keywords are case sensitive. The values of these keywords must be os-types. An os-type is a four-character string or keyword that is case sensitive.

Arguments

pathname A pathname.

`:if-exists` A keyword that determines what to do if the file already exists. If *pathname* already exists and the value of `:if-exists` is `:error` (the default), Macintosh Common Lisp signals an error. If its value is `nil`, Macintosh Common Lisp does nothing and returns `nil`. If it is `:overwrite` or `:supersede`, then Macintosh Common Lisp overwrites or replaces the previous file and returns the new file.

`:mac-file-type`
The os-type of the new file. The default is `:TEXT`. Directories do not have Macintosh types.

`:mac-file-creator`
The creator of the new file. The default is `:CCL2`. Directories do not have Macintosh creators.

---

open [*Function*]

Syntax `open` *filename* `&key :direction :element-type :if-exists :if-does-not-exist :external-format :mac-file-creator :fork`

Description The Common Lisp function `open` opens a stream to the file specified by *filename*, which may be a string, a pathname, a logical pathname, or a stream. Two new keywords, `:mac-file-creator` and `:fork`, distinguish the MCL implementation from Common Lisp's; the keyword arguments `:direction` and `:if-exists` can each take an additional value. The additional MCL keywords and values are documented next.

Arguments

`:direction` A pathname or string. This keyword can now take the value `:shared` in addition to `:input`, `:output`, `:io`, and `:probe`. The value `:shared` is the same as `:io` except that more than one stream can be open to a file at the same time. It defaults to `:input`.

`:if-exists`	The action to take when the direction is `:output` or `:io` and the file already exists. This argument can take the value `:dialog` in addition to the values `:append`, `:error`, `:new-version`, `:rename`, `:rename-and-delete`, `:overwrite`, `:supersede`, and `nil`. The default is `:error`. The values `:dialog`, `:rename`, and `:new-version` cause a dialog box to request confirmation if the file already exists.
`:external-format`	A four-character string to store as the Macintosh file type. Its value defaults to `:default`, in which case the Macintosh file type is `:TEXT`.
`:mac-file-creator`	The Macintosh file creator. It defaults to `:CCL2`.
`:fork`	An argument specifying whether to open the data fork or the resource fork. It may have the value `:data` (the default) or `:resource`.

---

rename-file [*Function*]

Syntax `rename-file` *old-pathname new-pathname* `&key :if-exists`

Description The Common Lisp function `rename-file` renames the specified *old-pathname.* The new name is the result of merging *new-pathname* with *old-pathname.* Both arguments may be a string, stream, or Lisp pathname. If *new-pathname* is an open stream associated with a file, then the stream itself and the file associated with it are affected.

If successful, the `rename-file` function returns three values. The first value is the renamed *old-pathname*. The second value is the true name of the *old-pathname* before it was renamed. The third value is the true name of the *old-pathname* after it was renamed. An error is signaled if the renaming operation is not successful.

Arguments

old-pathname	The old pathname of the file or directory.
new-pathname	The new pathname of the file or directory.
`:if-exists`	A keyword that determines what to do if the file already exists. If *new-pathname* already exists and the value of `:if-exists` is `:error` (the default), Macintosh Common Lisp signals an error. If its value is `nil`, Macintosh Common Lisp returns `nil`. If it is `:overwrite` or `:supersede`, then Macintosh Common Lisp overwrites or replaces the previous file and returns the new file.

Example

```
? (rename-file "hd:doc:file system notes"
      "BigTowel:misc:renamed notes")
#P"BigTowel:misc:renamed file system notes"
#1P"hd:doc:file system notes"
#1P"BigTowel:misc:renamed file system notes"
```

`file-create-date` [*Function*]

`file-write-date` [*Function*]

`set-file-create-date` [*Function*]

`set-file-write-date` [*Function*]

Syntax

`file-create-date` *pathname*
`file-write-date` *pathname*
`set-file-create-date` *pathname time*
`set-file-write-date` *pathname time*

Description

These functions report on or modify the creation and modification dates of files. The `file-create-date` function returns the time when the volume, directory, or file specified by *pathname* was created. The `file-write-date` function returns the time when the volume, directory, or file specified by *pathname* was last modified. The corresponding `set-` functions change these parameters.

Arguments

pathname	A pathname, string, or stream.
time	A time, given in the Common Lisp universal time format. (The Common Lisp universal time format is described in *Common Lisp: The Language*, on page 703.)

File operations

The following functions operate on files only. These functions, in conjunction with the `directory` function, provide the needed flexibility for operating on directories.

`copy-file` [*Function*]

Syntax

`copy-file` *old-pathname new-pathname* `&key :if-exists :fork`

Description The `copy-file` function copies the file to a file corresponding to the pathname specified by merging *new-pathname* with *old-pathname*. Arguments may be either strings, Lisp pathnames, or streams. If *new-pathname* does not have a filename component, then the filename of *old-pathname* is used.

If successful, the `copy-file` function returns three values. The first value is the new pathname with the filename component filled in. The second value is the true name of the file before it was copied. The third value is the true name of the copied file. An error is signaled if the copying operation is not successful.

Arguments

old-pathname	The old pathname of the file.
new-pathname	The new pathname of the file.
`:if-exists`	If *new-pathname* already exists and the value of `:if-exists` is `:error` (the default), Macintosh Common Lisp signals an error. If its value is `nil`, Macintosh Common Lisp returns `nil`. If it is `:overwrite` or `:supersede`, then Macintosh Common Lisp overwrites or replaces the previous file and returns the new file.
`:fork`	The type of fork. This value can be `:both`, `:data`, or `:resource`. The default is `:both`.

Example

```
? (copy-file "BigTowel:misc:renamed notes"
     "BigTowel:CCL Doc:copy")
#P"BigTowel:CCL Doc:copy"
#1P"BigTowel:misc:renamed notes"
#1P"BigTowel:CCL Doc:copy"
```

`lock-file` [*Function*]

`unlock-file` [*Function*]

`file-locked-p` [*Function*]

Syntax

`lock-file` *pathname*
`unlock-file` *pathname*
`file-locked-p` *pathname*

Description These functions allow you to manipulate the software lock that prevents modifications to a particular file. The `file-locked-p` function returns `nil` if the file is not locked.

If a file is locked, opening it creates a read-only buffer. You can look at the file but you cannot modify it.

Argument *pathname* A pathname, string, or stream.

mac-file-type [*Function*]

mac-file-creator [*Function*]

set-mac-file-type [*Function*]

set-mac-file-creator [*Function*]

Syntax

`mac-file-type` *pathname*
`mac-file-creator` *pathname*
`set-mac-file-type` *pathname os-type*
`set-mac-file-creator` *pathname os-type*

Description

Every Macintosh file has two parameters specifying the type of the file and the application that created the file. These parameters, called *os-type*s, are specified by four-character keywords or symbols that are case sensitive.

The `mac-file-type` and `mac-file-creator` functions return keywords indicating the type and creator parameters of *pathname*.

The `set-mac-file-type` and `set-mac-file-creator` functions destructively modify the type or creator of *pathname*. The new type or creator is returned as a keyword.

Arguments

pathname A pathname, string, or stream.

os-type The parameters specifying the type of the file and the application that created it. The *os-type* parameter may be a string of four characters or a four-character keyword. Files created by Macintosh Common Lisp have the creator `:CCL2` and the type `:TEXT` or `:FASL`. The *os-type* arguments are case sensitive and may contain spaces.

open-file-streams [*Variable*]

Description

The `*open-file-streams*` variable is bound to a list of all streams open to disk files. The user should not change this variable. It is updated automatically by file stream operations.

file-resource-size [*Function*]

Syntax

`file-resource-size` *path*

Description

Returns the size in bytes of the resource fork of the file whose pathname is *path*.

`file-data-size` [*Function*]

Syntax `file-data-size` *path*

Description Returns the size in bytes of the data fork of the file whose pathname is *path.*

`file-allocated-resource-size` [*Function*]

Syntax `file-allocated-resource-size` *path*

Description Returns the number of bytes allocated for the resource fork of the file whose pathname is *path*.

`file-allocated-data-size` [*Function*]

Syntax `file-allocated-data-size` *path*

Description Returns the number of bytes allocated for the data fork of the file whose pathname is *path.*

`file-info` [*Function*]

Syntax `file-info` *path*

Description Returns six values for the file whose pathname is *path*: create-date, modify-date, resource length, data length, allocated resource length, allocated data length.

Volume operations

Volume operations take as an argument either an integer (the volume number) or a pathname or string. If the argument is a pathname or string, only the volume component (the root directory) is used. Volume numbers are unique negative integers assigned to each mounted volume. Volumes numbers change from session to session and may change if a volume is unmounted and remounted. Within these limits, volume numbers allow a program to distinguish between multiple volumes with the same name. The volume number 0 is used to specify the default volume. If a string is used to specify a volume, it must contain a colon.

Drive numbers are positive integers denoting physical devices.

The following functions signal an error if the number or pathname given as an argument does not correspond to a mounted volume.

`volume-number` [*Function*]

Syntax `volume-number` *volume*

Description The `volume-number` function returns the volume reference number of *volume*. If *volume* is a valid volume number, it is simply returned.

Argument *volume* An integer, pathname, or string representing a volume.

Example

See the example under `drive-name`.

`eject-disk` [*Function]*

Syntax `eject-disk` *volume*

Description The `eject-disk` function ejects *volume* if possible. It is not possible to eject hard disks. If successful, `eject-disk` returns the true name of *volume;* otherwise, it signals an error. It does not unmount the volume.

Argument *volume* A volume number, drive number, pathname, or string representing a volume.

`eject&unmount-disk` [*Function*]

Syntax `eject&unmount-disk` *volume*

Description The function `eject&unmount-disk` ejects and unmounts *volume* if possible. If successful, `eject&unmount-disk` returns the true name of *volume;* otherwise, it signals an error. It is not possible to eject hard disks.

Argument *volume* A volume number, drive number, pathname, or string representing a volume.

`disk-ejected-p` [*Function*]

Syntax `disk-ejected-p` *volume*

Description The `disk-ejected-p` function returns `t` if the volume is ejected and `nil` otherwise. It signals an error if the specified *volume* is not mounted. The `probe–file` function can be used to check whether a volume is mounted.

Argument *volume* A volume number, drive number, pathname, or string representing a volume.

`hfs-volume-p` [*Function*]

Syntax `hfs-volume-p` *volume*

Description The `hfs-volume-p` function returns `t` if *volume* uses the Hierarchical File System (HFS) and `nil` if it uses the Macintosh File System (MFS). Most current Macintosh computers use only HFS devices, with the exception of floppy disks.

The HFS and MFS file systems are described in *Inside Macintosh.*

Argument *volume* A pathname or string representing a volume.

`flush-volume` [*Function*]

Syntax `flush-volume` *volume*

Description Some file system manipulations are buffered for execution at a later time. The `flush-volume` function ensures that all buffered file manipulations to a specified volume are performed. The `flush-volume` function returns the name of the volume affected.

Argument *volume* A pathname or string representing a volume.

`drive-name` [*Function*]

Syntax `drive-name` *number*

Description The `drive-name` function returns the name of the drive whose drive number or volume number is *number*.

Argument *number* A fixnum. A positive number is a drive number; a negative number, a volume number.

Example

```
? (volume-number #P"Dr. Johnson:")
-1
? (volume-number -1)
-1
? (drive-name -1)
#P"Dr. Johnson:"
```

`drive-number` [*Function*]

Syntax `drive-number` *pathname*

Description The `drive-number` function returns the drive number of the drive indicated by *pathname*.

Argument *pathname* A pathname or string.

User interface

The following functions let the user choose or set a pathname to a file or directory.

choose-file-dialog

[*Function*]

Syntax

```
choose-file-dialog &key :mac-file-type :directory
:button-string
```

Description

The `choose-file-dialog` function displays the standard Macintosh `SFGetFile` dialog box, allowing you to select a file for reading. Unless the dialog is canceled, this function returns a pathname.

Arguments

`:mac-file-type`
: An os-type parameter or list of os-type parameters. If specified, only files with the given Macintosh file type are displayed in the dialog box. Os-types are case sensitive.

`:directory`
: A pathname or string. Specifies the directory shown when the dialog box first appears. It defaults to the last directory shown by the Choose File dialog box or Choose New File dialog box.

`:button-string`
: A string. Specifies the text that appears in the button that opens the chosen file. The default is `Open`.

choose-new-file-dialog

[*Function*]

Syntax

```
choose-new-file-dialog &key :directory :prompt :button-
string
```

Description

The `choose-new-file-dialog` function displays the standard Macintosh `SFPutFile` dialog box, allowing you to specify a destination file for writing. An alert dialog box requests confirmation if an existing file is chosen. Unless canceled, it returns a pathname.

Arguments

`:directory`
: Specifies the directory shown when the dialog box first appears. It defaults to the last directory shown by the Choose File dialog box or Choose New File dialog box.The filename component of `:directory` is used as the default filename in the editable-text item of the dialog box.

`:prompt`
: Specifies the text to display above the area in which the user types the filename. If supplied, `:prompt` should be a string. The default prompt is As….

`:button-string`
: Specifies the text that appears in the button that opens the file. The default is Save.

`choose-directory-dialog` [*Function*]

Syntax `choose-directory-dialog &key :directory`

Description The function `choose-directory-dialog` displays a variation of the standard Macintosh `SfGetFile` dialog box. Unless canceled, it returns a directory pathname.

Argument `:directory` Specifies the directory shown when the dialog box first appears. It defaults to the last directory shown by the `choose-file-dialog`, `choose-new-file-dialog`, or `choose-directory-dialog` dialog box.

`choose-file-default-directory` [*Function*]

Syntax `choose-file-default-directory`

Description The function `choose-file-default-directory` returns the namestring of the last directory selected by the `choose-file-dialog`, `choose-new-file-dialog`, or `choose-directory-dialog` dialog box. Initially, this is the directory that is the translation of `"home:"`.

`set-choose-file-default-directory` [*Function*]

Syntax `set-choose-file-default-directory` *pathname*

Description The function `set-choose-file-default-directory` sets the default directory used by the `choose-file-dialog`, `choose-new-file-dialog`, or `choose-directory-dialog` dialog box to *pathname*. It returns *pathname*.

Argument *pathname* A pathname or string.

Logical directory names

If you are new to Macintosh Common Lisp, you do not need to read this section.

Previous versions of Macintosh Common Lisp provided a facility, called logical pathnames, that is now called **logical directory names.** It is not connected with the new Common Lisp logical pathname facility. You can still use logical directory names; however, they will probably go away in future releases of Macintosh Common Lisp. For your new code, you should use Common Lisp logical pathnames.

Logical directory names serve as variables in a pathname string. Their goal is to allow code with embedded pathname information to run under different directory hierarchies.

Unlike physical directories, which end with colons, logical directory names end with semicolons.

Because of the use of a semicolon as the directory delimiter in MCL logical directories, a namestring containing semicolons but no host will not parse to a Common Lisp logical pathname. However, if it is merged with a logical pathname, the result is a logical pathname.

```
? (ccl::logical-pathname-p (pathname "blotz;blitz;"))
NIL
? (ccl::logical-pathname-p
   (merge-pathnames
    (pathname "blotz;blitz;")
    (pathname "ccl:")))
T
```

The following MCL functions and variables govern logical directory names.

`*logical-directory-alist*` [*Variable*]

Description The `*logical-directory-alist*` variable contains an association list that maps between logical and physical pathnames.

This variable was formerly called `*logical-pathname-alist*`.

`def-logical-directory` [*Function*]

Syntax `def-logical-directory` *logical-directory-name physical-pathname*

Description The function `def-logical-directory` defines a new logical directory name and adds it to `*logical–directory–alist*`. It returns the new value of `*logical–directory–alist*.`

To remove a logical pathname from the environment, call `def-logical-directory` with a *physical-pathname* of `nil`.

This function was formerly called `def-logical-pathname`.

Arguments *logical-directory-name*

A logical directory name.

physical-pathname

The physical pathname associated with *logical-directory-name*. It may contain logical components.

Chapter 9:

Debugging and Error Handling

Contents

This chapter discusses debugging tools in Macintosh Common Lisp. These tools include compiler options, Fred commands, debugging functions, error-signaling functions, functions to break or cancel operations, backtrace, facilities to step through a program, trace functions, and an advise function. In addition, any part of any MCL object can be inspected and, when appropriate, edited within the Inspector.

You should read this chapter to familiarize yourself with the debugging environment in Macintosh Common Lisp.

Debugging tools in Macintosh Common Lisp

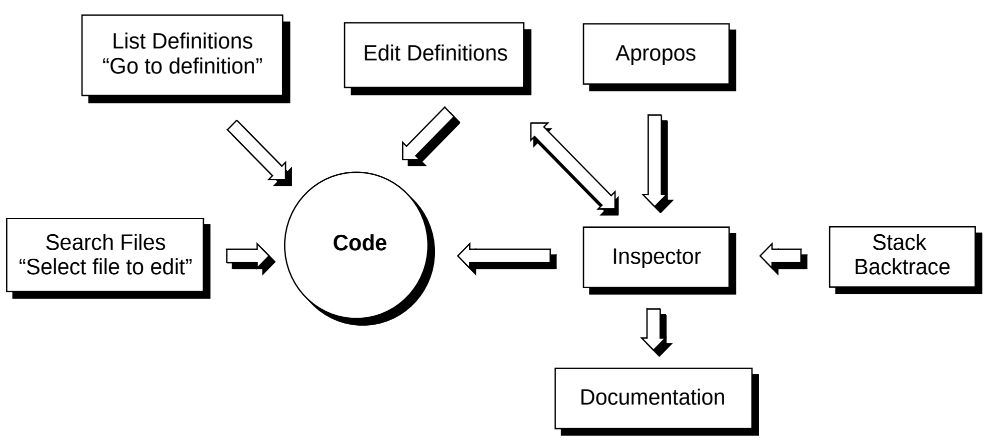
Macintosh Common Lisp provides several tools to help programmers examine and debug functions, source code, and environments:

- compiler options to retain information useful for later programming
- a set of Fred commands
- debugging functions
- a set of functions for signaling errors and aborting operations (these functions may optionally enter a break loop)
- a break-loop facility, which interrupts a program and allows you to look at the stack and examine dynamic values before returning
- a Stack Backtrace
- a single-expression stepper
- a trace function
- an Inspector

The Tools menu contains most of these tools and the Fred Commands window; the other tools are available through MCL expressions.

MCL debugging tools form an integrated whole, allowing you to look at your code from a variety of perspectives. Figure 9-1 shows the MCL debugging tools and their relationships. From each of the listed windows you can examine code in the windows they point to.

■ **Figure 9-1** MCL debugging tools



Here is what the various components of Figure 9-1 do.

- The Apropos window accepts one or two strings and a number of options and finds all definitions containing the strings and matching the options.
- The Stack Backtrace window examines the state of the stack during a break loop.
- The Documentation window brings up documentation for Common Lisp and MCL symbols.
- The Inspector window allows you to examine all the components of any data object.
- The Edit Definitions window accepts the name of a definition and finds its source code.
- The List Definitions window lists all definitions in the current buffer and allows you to pick one for editing.
- Search Files lets you search files for the presence of a string.

When available, code is always the best documentation. Two keyboard commands are often used to examine code.

- Pressing Meta-period when the insertion point is within or next to an expression in code allows you to examine its source code. You can examine the source code of many MCL expressions.
- Pressing Control-Meta and clicking an expression acts like pressing Meta-period but also allows you to examine expressions within Inspector windows.

Compiler options

The MCL compiler can optionally retain information useful for later programming. It can also provide useful debugging information at compile time. The behavior of the compiler is regulated by the global variables listed in Table 9-1.

■ **Table 9-1** Compiler options

Variable	Purpose
`*fasl-save-definitions*`	Provides a default value for the :save-definitions keyword argument to compile-file; determines whether lambda expressions are saved in the compiled file. *Default is* `nil`; lambda expressions are not saved in the compiled file and are not available when the file is loaded. If true, lambda expressions are saved. Compiled functions without lambda expressions cannot be stepped.
`*fasl-save-doc-strings*`	Provides a default value for the `:save-doc-strings` keyword argument to `compile-file`; determines whether documentation strings are saved in the compiled file. *Default is* `nil`; documentation strings are not saved and are not available when the file is loaded. If true, documentation strings are saved in the compiled file and are available through the Inspector and the `documentation` function (bound to the keyboard command Control-X Control-D).
`*fasl-save-local-symbols*`	Provides a default value for the `:save-local-symbols` keyword argument to `compile-file`. *Default is* `nil`; local symbols are not saved in the compiled file. If true, local symbols are saved in the compiled file and are available when the file is loaded. Generally increases `.fasl` file size by about 15–20 percent.
`*record-source-file*`	Determines whether compiler records source file of definitions. The definition contains a pointer to the source file. You can retrieve the definition by pressing Meta-period when the insertion point is next to the symbol name. *Default is true;* compiler records source file of all definitions. Meta-period retrieves source code. If `nil`, no record is kept, and Meta-period cannot retrieve source code.
`*save-definitions*`	Determines whether compiled functions can be uncompiled. *Default is* `nil`; lambda expressions are not retained; functions cannot be stepped through. If true, lambda expressions are retained; functions can be stepped through.

(continued)

■ **Table 9-1** Compiler options (continued)

Variable	Purpose
`*save-doc-strings*`	Determines whether documentation strings are retained. *Default is* `nil`; documentation strings are discarded. This can save memory. If true, documentation strings are retained.
`*save-local-symbols*`	Determines whether names of arguments and local variables are saved when functions are compiled. *Default is* `nil`; information is discarded. If true, information is retained, and `*arglist-on-space*` and `backtrace` will show actual argument names.
`*warn-if-redefine*`	Helps prevent accidental redefinition of a function defined somewhere else. *Default is true;* compiler issues a warning whenever a function, macro, or variable is redefined from a new file. If `nil`, compiler does not issue warnings when user functions are redefined (but does when built-in functions are redefined).
`*warn-if-redefine-kernel*`	Helps prevent accidental redefinition of a built-in function. *Default is true;* compiler signals a continuable error whenever a built-in function is redefined. If `nil`, compiler does not signal an error when built-in functions are redefined. Use with caution.

Fred debugging and informational commands

Several Fred command keystrokes help the programmer get information about MCL expressions and the MCL environment.

Remember that you access Meta commands by pressing the Option key. You access Control commands by pressing the Control key (if your keyboard has one) or by pressing Command or Command-Shift.

Several of these commands are on the Tools menu; those menu items are listed in Table 9-2.

■ **Table 9-2** Fred debugging and informational commands

Purpose	Keystroke/menu item	Effect
Display Fred commands	Control-?, Fred Commands on Tools menu	Brings up the Fred Commands window. This window contains a list of all Fred keyboard commands available in the global command table. The list is regenerated each time the window is created. The Fred Commands window may be searched, saved, and printed.
Edit definition	Meta-period, Control-Meta-click, Edit Definition on Tools menu	Attempts to bring up the source code definition for the symbol surrounding the insertion point. If the symbol is defined in more than one source file, the user is given a choice of definitions. If the symbol is defined as a slot in a `defclass`, Meta-period finds the *approximate* location of the symbol. Search backward with Control-R to find the location at which the symbol is defined. This function works for most forms that are defined with `*record-source-file*` set to `t`.
Get argument list information	Control-X Control-A	Prints the argument list of the function bound to the symbol surrounding the insertion point. Argument list is displayed in the minibuffer if the value of `*mini-buffer-help-output*` is `t`; otherwise, it is displayed in the `*standard-output*` stream. The `ed-arglist` function works for built-in functions and macros, and for most functions and macros defined with `*save-local-symbols*` or `*fasl-save-local-symbols*` set to `t`.

(continued)

■ **Table 9-2** Fred debugging and informational commands (continued)

Purpose	Keystroke/menu item	Effect
Get documentation for current expression	Control-X Control-D, Documentation on Tools menu	Opens a dialog box displaying the symbol surrounding the insertion point and the documentation string of the function bound to that symbol. If no documentation string is available, displays "No documentation available." This function works for built-in functions and macros and for most forms defined with *save-doc-strings* set to true.
Inspect current expression	Control-X Control-I	Inspects the current symbolic expression.
Macroexpand current expression	Control-X Control-M	Macroexpands the current expression with `macroexpand` and pretty-prints the result to `*standard-output*`.
Macroexpand current expression repeatedly	Control-M	Macroexpands the current expression repeatedly with `macroexpand-1` until the result is no longer a macro call and pretty-prints the result to `*standard-output*`.
Print information about active window	Control–=	Prints information about the current Fred window to `*standard-output*`.
Read current expression	Control-X Control-R	Prints the result of reading the current symbolic expression. This is useful for tracking read-time bugs, particularly in expressions containing backquotes.

Here are some examples of using these Fred keyboard equivalents.

To perform macroexpansion with Control-X Control-M:

```
? (defmacro foo (x y)
  `(+ ,x ,y))
FOO
? (defmacro bar (z)
  `(foo ,z ,z))
BAR
? (foo 10 20);Control-X Control-M
(+ 10 20)
```

```
? (bar 10);Control-X Control-M
(+ 10 10)
```

To perform macroexpansion with Control-M:

```
? (foo 10 20);Control-M
(+ 10 20)

? (bar 10);Control-M
(foo 10 10)
(+ 10 10)
```

To read the current expression with Control-X Control-R:

```
(print `(2  ,(+ 3 4) 6));<c-x c-r>
(print (cons 2 (cons (+ 3 4) '(6))))

#@(2 2);<c-x c-r>
131074
```

Debugging functions

The following functions and variables are useful when programming. They provide information on the MCL programming environment and aid in testing and tracking functions.

apropos [*Function*]

Syntax `apropos` *string-or-symbol* `&optional` *package*

Description The `apropos` function finds all interned symbols whose print names contain *string* as a substring and prints the name, function definition, and global value of each symbol. The value `nil` is returned. The result is printed to `*standard-output*`.

The `apropos` function is not case sensitive.

The functionality of `apropos` is also available through Apropos on the Tools menu. In the Apropos dialog box, you can type a symbol name or part of a symbol name. The Apropos dialog box displays a scrollable list of symbol names. Double-clicking one brings up an Inspector window for that symbol.

Arguments

string-or-symbol
: Any string or symbol.

package
: A package within which to search for *string-or-symbol.* When *package* is `nil`, all packages are searched.

Example

```
? (apropos 'bitmap)
BITMAP
$BITMAP.TOPLEFT,  Value: 6
$BITMAP.TOP,  Value: 6
$BITMAP.LEFT,  Value: 8
_SCRNBITMAP, Def: MACRO FUNCTION,  Value: 43059
$AFPBITMAPERR,  Value: -5004
$ICONBITMAP,  Value: 2574
:BITMAP,  Value: :BITMAP
```

◆ *Note:* If a symbol is given, it is interned (that is, a symbol is created and installed in the current package) and therefore the symbol always appears in the output of `apropos`. So, for example, typing `(apropos 'i-just-made-this-up)` retrieves `(i-just-made-this-up)`. This can confuse new programmers who are using `apropos` to check on the existence of a symbol. As you would expect, the Apropos dialog box does not intern strings typed into it as symbols; however, after a previously nonexistent symbol is interned with `apropos`, the Apropos dialog box will find it.

---

apropos-list [*Function*]

Syntax `apropos-list` *string-or-symbol* `&optional` *package*

Description The `apropos-list` function returns a list of all symbols whose print names contain *string-or-symbol* as a substring.

If a symbol is given, it is interned and therefore always appears in the list returned by `apropos-list`. So, for example, typing `(apropos-list 'i-made-this-up-too)` retrieves `(i-made-this-up-too)`.

The `apropos-list` function is not case sensitive.

Arguments

string-or-symbol
: Any string or symbol.

package
: A package within which to search for *string-or-symbol.* When *package* is `nil`, all packages are searched.

Example

```
? (apropos-list 'bitmap)
```

```
(:BITMAP $ICONBITMAP $AFPBITMAPERR _SCRNBITMAP $BITMAP.LEFT
$BITMAP.TOP $BITMAP.TOPLEFT BITMAP)
? (setq make-syms (apropos-list 'bitmap))
(:BITMAP $ICONBITMAP $AFPBITMAPERR _SCRNBITMAP $BITMAP.LEFT
$BITMAP.TOP $BITMAP.TOPLEFT BITMAP)
? (setq make-syms (sort make-syms #'string<
                        :key #'symbol-name))
($AFPBITMAPERR $BITMAP.LEFT $BITMAP.TOP $BITMAP.TOPLEFT
$ICONBITMAP :BITMAP BITMAP _SCRNBITMAP)
? (pprint make-syms)
($AFPBITMAPERR
$BITMAP.LEFT
$BITMAP.TOP
$BITMAP.TOPLEFT
$ICONBITMAP
:BITMAP
BITMAP
_SCRNBITMAP)
```

`arglist` [*Function*]

Syntax `arglist` *symbol* `&optional` *include-bindings use-help-file*

Description The `arglist` function returns two values, the argument list of *symbol* and how the list was computed. The second value can be one of `:definition`, `:declaration`, `:analysis`, `:unknown`, or `nil`. The value `:definition` means that `*save-definitions*` was true when the function was compiled; the value `:declaration` means that either the argument list was found in the MCL Help file or you declared the argument list with `(setf (arglist` *symbol*`)` *arglist*`)`. The value `:analysis` means that the argument list was computed from information stored with the function; `:unknown` means that the symbol was bound to a function, but no argument list information could be determined; and `nil` means that the symbol was not bound to a function.

Arguments

symbol A symbol.

include-bindings
A value. If this value is specified and true, then the default values of optional and keyword parameters are included, if known.

use-help-file A Boolean value. If true (the default), the argument list is taken from the MCL Help file. If `nil`, the argument list is computed directly from information stored within the function. (This parameter is useful if you suspect that the MCL Help file may be incorrect.)

`documentation` [*Generic function*]

Syntax `documentation` (*x* `thing`) `&optional` *doc-type*

Description The generic function `documentation` returns the documentation string of *doc-type* for *x*. If *x* is a method object, a class object, a generic function object, a method combination object, or a slot-description object, *doc-type* may not be supplied, or an error is signaled. If *x* is a symbol or a list of the form `(setf` *symbol*`)`, *doc-type* must be supplied. See Table 9-3 for the documentation type that should be supplied for various MCL constructs.

Documentation strings can be changed with the Common Lisp generic function `(setf documentation)`, documented on page 842 of *Common Lisp: The Language.*

Documentation strings are retained only if the value of `*save-doc-strings*` is true when the definition occurs. If no documentation string is available, `documentation` returns `nil`.

Arguments

x A method object, class object, generic function object, method combination object, slot-description object, symbol, or list of the form `(setf` *symbol*`)`.

doc-type One of the symbols `variable`, `function`, `structure`, `type`, or `setf`.

Example

```
? (documentation 'view-draw-contents 'function)
"The event system calls this generic function whenever a view
needs to redraw any portion of its contents.  For a view, the
function is applied focused on the view; for a simple view,
it is focused on the view's container."
? (documentation 'window 'type)
"The window class, from which all window objects inherit.
Windows in turn inherit from view. All windows are streams."
```

Table 9-3 lists the values of *doc-type* that should be supplied with various MCL constructs.

■ **Table 9-3** Constructs and their documentation types

Construct	Documentation type
Function	`function`
Generic function	`function`
Special form	`function`
Macro	`function`
Variable	`variable`
Constant	`variable`
`defstruct` structure	`structure`
Class object	`type`
Type specifier	`type`
`defsetf` definition	`setf`
`define-setf-method` definition	`setf`
Method combination	`method-combination`

edit-definition-p [*Function*]

Syntax `edit-definition-p` *name* `&optional` *type specializers qualifiers*

Description The function `edit-definition-p` returns source file information for a symbol, method, or function.

It returns five values: a list of definition types and source file names where the definition occurs (the first file in the list is the one containing the most recent definition); the name of *name;* the definition type found (one of `function`, `method`, `structure`, `class`, and so on); a list of its method qualifiers, such as `(:before)`, `(:after)`, or `(:around)`, and a list of the method specializer classes. If *name* is not the name of a method, the two last values are `nil` and `t`.

Arguments

name A symbol, method, or function.

type The type of definition desired. Allowable values are any data type that can have a source file: for example, `function`, `method`, `structure`, `class`, or `variable.` The default value, `t`, finds whatever exists.

specializers A list of specializer classes for a method. Giving this argument a non-`nil` value forces the value of the *type* argument to be `'method`.

qualifiers A list of qualifiers for a method, for example, `(:before)`, `(:after)`, or `(:around)`. The default value is `t`, which finds a method with any qualifier.

Example

```
? (edit-definition-p 'pop-up-menu)
((CLASS . "ccl:library;pop-up-menu.lisp"))
POP-UP-MENU
T
NIL
T
? (edit-definition-p 'view-draw-contents 'method '(basic-
editable-text-dialog-item) :after)
NIL
VIEW-DRAW-CONTENTS
METHOD
(BASIC-EDITABLE-TEXT-DIALOG-ITEM)
:AFTER
```

`*help-output*` [*Variable*]

Description The `*help-output*` variable specifies the stream to which documentation string and argument list information is printed when accessed through Fred keyboard commands or the Inspector. This variable is initially bound to `*standard-output*`.

`print-call-history` [*Function*]

Syntax `print-call-history`

Description The function `print-call-history` writes a full Stack Backtrace to `*debug-io*`.

`select-backtrace` [*Function*]

Syntax `select-backtrace`

Description The function `select-backtrace` opens a Stack Backtrace window if it is meaningful to backtrace. If there is no context for backtracing, the function signals an error.

`room` [*Function*]

Syntax `room &optional` *detailed-p*

Description The `room` function prints information on the amount of space available in the Lisp operating system.

Argument *detailed-p* A value indicating how much information to print. If this value is `nil`—the default—minimal information is printed. If it is non-`nil`, more detailed information is printed.

Example

```
? (room)
There are at least 1356752 bytes of available RAM.

                 Total Size          Free               Used
Mac Heap:      540576 (527K)      132600 (129K)      407976
(399K)
Lisp Heap:    2097152 (2048K)    1224152 (1195K)     849016
(829K)
 (Static):      458752 (448K)
Stacks:         218100 (212K)
? (room t)
There are at least 1344548 bytes of available RAM.

                 Total Size          Free               Used
Mac Heap:      540576 (527K)      132604 (129K)      407972
(399K)
Lisp Heap:    2097152 (2048K)    1211944 (1183K)     860736
(840K)
 (Static):      458752 (448K)
Stacks:         218100 (212K)
Markable objects:      777112 (758K) dynamic, 212776 (207K)
static.
Immediate objects:      83624 (81K) dynamic, 242992 (237K)
static.
```

---

inspect [*Function*]

Syntax `inspect` *thing*

Description The `inspect` function inspects *thing*.

Macintosh Common Lisp supports the Common Lisp `inspect` function with a window-based Inspector. In addition to calling the `inspect` function, there are two other ways of invoking the Inspector directly: choosing Inspect from the Tools menu, or giving the keyboard equivalent, Control-X Control-I. In addition, double-clicking a symbol name from the Apropos dialog box, or choosing a symbol and clicking the Inspect button, invokes the Inspector on that symbol.

Argument *thing* Any Lisp data object.

Example

```
? (inspect 'windows)
#<INSPECTOR-WINDOW "WINDOWS" #x467281>
```

---

top-inspect-form [*Function*]

Syntax `top-inspect-form`

Description The `top-inspect-form` function returns the form being inspected by the active Inspector window.

Example

```
? (top-inspect-form)
WINDOWS
```

For full details on the Inspector, see “The Inspector” on page 348.

Error handling

Macintosh Common Lisp uses the Common Lisp condition system, which reconceptualizes and adds to Common Lisp’s previous error-detection and error-handling capabilities.

A *condition* is an interesting situation that has been detected and announced within a program. An *error* is a condition from which the program cannot continue normally, but requires some sort of intervention, either by program control or from the user.

Most MCL error-handling functions now follow the definitions of those functions given in *Common Lisp: The Language*, Chapter 24, “Errors,” and Chapter 29, “Conditions.” (Note that pages 886–887 of *Common Lisp: The Language* supersede the earlier discussion of `error` and `cerror` in Chapter 24 of the same book.) MCL extensions to those functions are described next.

Functions extending Common Lisp error handling

The following functions extend the Common Lisp condition system.

`abort-break` [*Function*]

Syntax `abort-break`

Description If the current read loop is waiting for input, the Common Lisp function `abort` calls the non–Common Lisp function `abort-break`, which decrements the abort level by 1. If there is input in the current read loop, the Common Lisp function `abort` calls the `abort` restart.

`cancel` [*Function*]

Syntax `cancel`

Description The `cancel` function throws to the nearest *catch-cancel*. (Described in “Simple turnkey dialog boxes” on page 239.) This function is generally called when the user clicks Cancel in a modal dialog box.

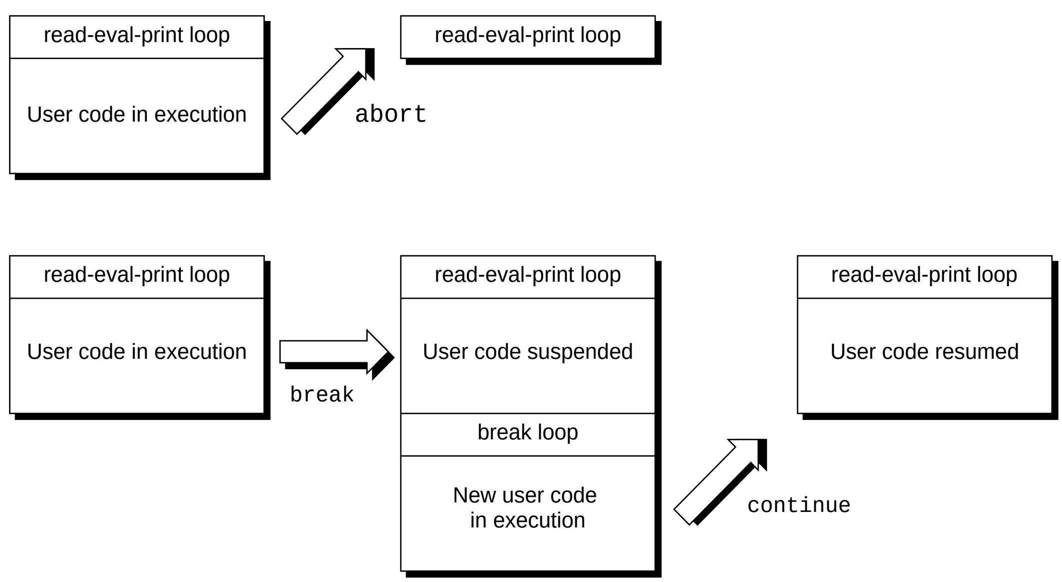
For information on the syntax of Common Lisp `throw` and `catch`, see *Common Lisp: The Language,* in particular page 192.

Break loops and error handling

At any point during an MCL program, program execution may be suspended and control passed to a break loop. A break loop behaves like the top-level read-eval-print loop. However, when you enter a break loop you do not exit your program and return control to the top level (as `abort` does). Instead, a break loop suspends your program and allows interaction on top of your program. From a break loop, you can resume the program or return to the top level.

Figure 9-2 shows the execution stack of Macintosh Common Lisp. Newer items are added to the bottom. The diagrams show that break loops add new areas to the stack, but `abort` and `continue` remove areas from the stack. New items are added to the bottom.

■ **Figure 9-2** Effects on the stack of `break`, `abort`, and `continue`



Within a break loop, the MCL question mark prompt is replaced by a number and an angle bracket. Expressions can be executed, just as they are in the normal Listener loop. Because the break loop runs on top of the interrupted program, all global variables have the values they had when the interrupted program was suspended, as the following code shows.

```
? *print-case*
:downcase
? *load-verbose*
t
? (defun show-specials ()
         (let ((*print-case* :upcase)
               (*load-verbose* nil))
           (break)
           (print "Now we have continued.")
           t))
show-specials
? (show-specials)
>Break:
> While executing: SHOW-SPECIALS
> Type Command-/ to continue, Command-. to abort.
> If continued: Return from BREAK.
See the Restarts… menu item for further choices.
1 > *print-case*
:UPCASE
1 > *load-verbose*
NIL
1 > (continue)
Continuing...
"Now we have continued."
t
? *print-case*
:downcase
```

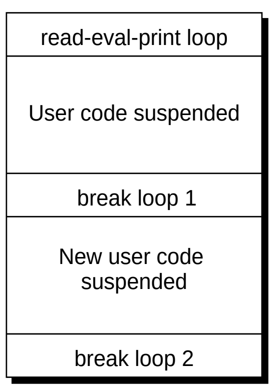
Break loops retain the dynamic environment of the interrupted program (that is, the values of global variables), but they do not retain the lexical environment of the interrupted program. For this reason, forms that you type into the break loop do not have access to the lexical variables of the interrupted program, as shown in the following code. (You can look at the lexical variables with the Stack Backtrace, described in "Stack Backtrace" on page 334.)

```
? (defun double (num)
    (unless (numberp num)
      (break))
    (+ num num))
DOUBLE
? (double 5)
10
? (double 'ten)
>Break:
> While executing: DOUBLE
```

```
> Type Command-/ to continue, Command-. to abort.
> If continued: Return from BREAK.
See the Restarts… menu item for further choices.
1 > num
> Error: Unbound variable: NUM
> While executing: SYMBOL-VALUE
> Type Command-/ to continue, Command-. to abort.
> If continued: Retry getting the value of NUM.
See the Restarts… menu item for further choices.
2 > (abort-break)
Aborted
1 > (abort-break)
Aborted
```

Break loops may be nested; that is, you can enter a break loop from a break loop, and so on. The current level is indicated by the number in the Listener prompt (see Figure 9-3).

■ **Figure 9-3** Nesting of break loops




```
? (break)
> Break:

> Type Command-/ to continue, Command-. to abort.
1 > (break)
> Break:

> Type Command-/ to continue, Command-. to abort.
2 >
```

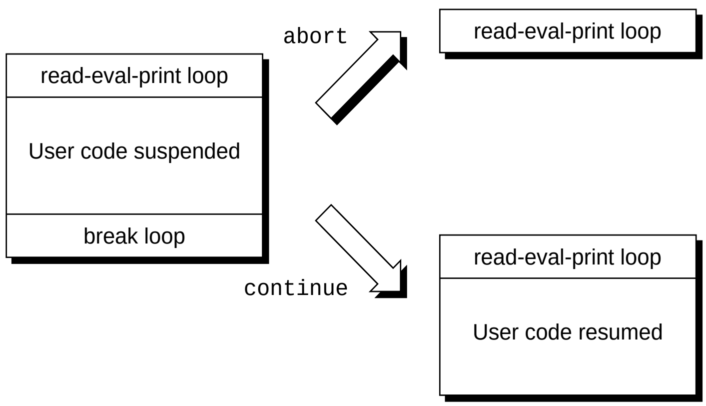
You can enter a break loop explicitly by calling the function `break` or `cerror`. In addition, if the value of `*break-on-errors*` is true, Macintosh Common Lisp enters a break loop whenever an error is signaled. If the value of `*break-on-warnings*` is true, Macintosh Common Lisp enters a break loop whenever `warn` is called. These functions and variables are described in “Functions and variables for break loops and error handling” on page 332.

Break is also available as a command on the Lisp menu.

There are two ways to leave a break loop: by calling `continue` and by calling `abort` (see Figure 9-4). Calling `continue` resumes the program from the point at which it was interrupted. Calling `abort` returns to the previous read loop. This may be the top-level loop or a prior break loop. In the case of `abort`, the suspended program is not resumed.

Abort and Continue are available as commands on the Lisp menu. You can also invoke `abort` at any point by pressing Command-period. Within a break loop, you can invoke `continue` by pressing Command-slash (Command-/).

■ **Figure 9-4** Two ways to leave a break loop



Functions and variables for break loops and error handling

The following functions and variables control break loops and error handling.

`break` [*Function*]

Syntax `break &optional` *format-string* `&rest` *arguments*

Description The `break` function prints the message specified by *format-string* and *arguments* and enters a break loop. It returns `nil` when continued.

The `break` function can also be invoked through the Lisp menu. This provides a convenient method for suspending a program at any point of execution.

If `break` is called during the dynamic extent of a call to `without-interrupts`, no action is taken.

Arguments

format-string	A `format` control string used to construct the break message.
arguments	Zero or more `format` arguments used to construct the break message.

---

continue [*Function*]

Syntax `continue &optional` *condition*

Description The `continue` function resumes execution of the code suspended by the most recent call to `break` or `cerror`. If there have been no calls to `break` or `cerror`, `continue` simply returns to the top level. If `condition` is present, the restart for *condition* is invoked.

Argument

condition	A condition.

---

break-on-errors [*Variable*]

Description The `*break-on-errors*` variable determines whether Macintosh Common Lisp enters a break loop when an error is signaled. The default value is true.

If the value of this variable is true, then Macintosh Common Lisp enters a break loop when an error is signaled.

If the value of this variable is `nil`, *then* errors simply cause a return to the read-eval-print loop.

---

break-on-warnings [*Variable*]

Description The `*break-on-warnings*` variable determines whether Macintosh Common Lisp enters a break loop when a warning is issued. The default value is `nil`.

If the value of this variable is true, then Macintosh Common Lisp enters a break loop when a warning is issued.

If the value of this variable is `nil`, *then* warnings do not interrupt program flow.

`*backtrace-on-break*` [*Variable*]

Description The `*backtrace-on-break*` variable determines whether Macintosh Common Lisp displays the Stack Backtrace whenever it enters a break loop. The default value is `nil`.

If the value of this variable is true, then Macintosh Common Lisp displays the Stack Backtrace window.

If the value of this variable is `nil`, *then* you must choose Backtrace from the Tools menu to see the Stack Backtrace dialog box.

`*error-print-circle*` [*Variable*]

Description In break or error loops, `*print-circle*` is set to the value of `*error-print-circle*`. The initial value is `t`.

Stack Backtrace

Beyond `print-call-history`, which prints a backtrace to `*debug-io*`, Macintosh Common Lisp provides a Stack Backtrace dialog box.

When inside a break loop, the Stack Backtrace command lets you examine the state of the suspended program. To see the Stack Backtrace dialog box, choose Backtrace from the Tools menu when you are in a break loop.

The Stack Backtrace shows the functions awaiting return values as well as the local variables of these functions (if the functions were compiled with `*save-local-symbols*` set to true). You can easily access and set the values in a stack frame. Finally, information on the program counter and stack frame address is given.

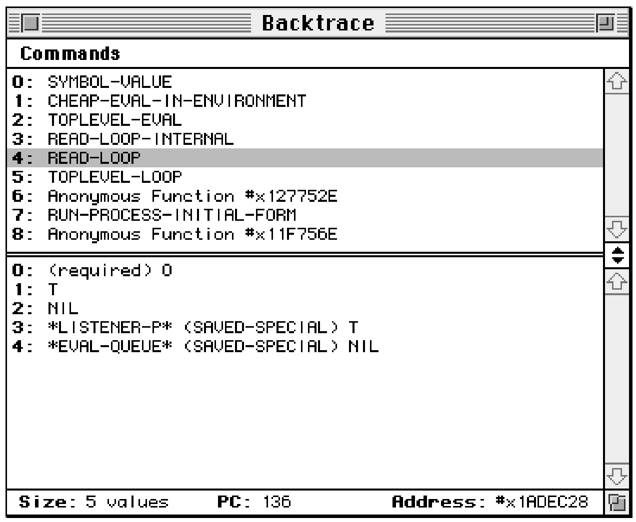
Certain internal functions are not shown in the Stack Backtrace by default. You can control this behavior, and the set of functions considered internal.

The Stack Backtrace dialog box shows two tables. The upper table shows the functions that are pending on the stack. The lower table is initially blank. When you single-click any function in the upper table, the lower table displays that function's stack frames. You can edit values in the lower table (but do so with caution).

Below the tables are three pieces of information about the frame: the number of values in the frame, the memory address of the frame, and the program counter within the function where execution has been suspended. The memory address is useful for low-level system debugging. You can use the program counter with `disassemble` to locate the point of a break within a function.

Figure 9-5 and the code that follows it show editing in the Stack Backtrace dialog box.

■ **Figure 9-5** A Stack Backtrace dialog box



Here is the code that produced the Stack Backtrace dialog box in Figure 9-5.

```
? (defun foo (x y)
   (let ((z 10))
     (break)
     (+ x y z)))
foo
? (foo 10 20)
> Break:
> Type Command-/ to continue,
Command-. to abort.
```

Single-click `foo`, then edit within the Stack Backtrace window:

```
1 > (local 1)
20
1 > (setf (local 1) 50)
50
1 > (continue)
Continuing...
70
```

This only works if the value of `*compile-definitions*` was true when `foo` was compiled. Otherwise, the result is still 40.

Double-clicking a function in the top table causes the function object to be inspected, giving you access to `edit-definition`, `documentation`, `arglist`, `disassemble`, and `uncompile-function`.

Because Macintosh Common Lisp supports tail recursion, any function that makes a tail-recursive call will not appear in the backtrace. To ease debugging, you can disable tail recursion with compiler declarations.

The stack frame in the lower table shows the names of local variables, if these were retained at compile time. If these were not retained, the parameters are listed as required, optional, keyword, or rest. You can use the `local` macro to access the values of these frames. In addition, you can double-click a value to inspect it.

`local` [*Macro*]

Syntax `local` *indicator*

Description The macro `local` returns the value in the current stack frame of the slot given by *indicator*. This macro can be used only when a Stack Backtrace dialog box is visible and when a frame is selected.

Argument *indicator* A symbol or number indicating a slot in the stack frame. A symbol can be used if the frame includes local symbol names and if the symbol is unique in the frame. Otherwise, a number giving the position in the frame should be used.

`set-local` [*Macro*]

Syntax `set-local` *indicator new-value*

Description The `set-local` macro changes the value at the specified *indicator* to *new-value*.

You can use `set-local` (or `local` with `setf`) to modify a value in a stack frame. Modify these values with caution, however, because the compiler may have made assumptions based on the initial values.

Arguments

indicator — A symbol or number indicating a slot in the stack frame. A symbol can be used if the frame includes local symbol names and if the symbol is unique in the frame. Otherwise, a number giving the position in the frame should be used.

new-value — The new value of the indicator.

`inspector::*backtrace-hide-internal-functions-p*` [*Variable*]

`inspector::*backtrace-internal-functions*` [*Variable*]

Description

If `inspector::*backtrace-hide-internal-functions-p*` is true (the default), internal stack frames are not shown in the Backtrace.

`inspector::*backtrace-internal-functions*` contains a list of functions considered to be "internal". You may add functions to and remove them from this list.

Single-expression stepper

The single-expression stepper allows you to examine a single form, expression by expression.

The `step` macro can be used on compiled functions only if their uncompiled definitions have been retained. If there is no uncompiled definition for a function, it is treated as an atomic unit as it is evaluated. A compiled function call is executed as a whole rather than being evaluated form by form. (This is how the `step` macro treats built-in functions.)

Function definitions are retained if the function is compiled with the `*save-definitions*` variable set to `t` or if a file is compiled with the `*fasl-save-definitions*` variable set to `t`. If the function was compiled with `*save-definitions*` set to `nil`, it must be recompiled or reloaded with `*compile-definitions*` set to `nil` before it can be evaluated.

Because evaluation occurs in a null lexical environment, `step` is usually called only from the top level. If it is called from within a function, it does not have access to the local environment in which it was called. However, internal stepping can be invoked through the `trace` macro, described in “Tracing” on page 338.

It is not generally possible to step through code that requires the use of `without-interrupts` or code that uses the Macintosh graphics interface.

`step` [*Macro*]

Syntax `step` *form*

Description The `step` macro evaluates *form* expression by expression, under user control.

Argument *form* Any Lisp form.

`*step-print-level*` [*Variable*]

`*step-print-length*` [*Variable*]

Description The `*step-print-level*` and `*step-print-length*` variables are used to set the values of `*print-level*` and `*print-length*` during step evaluation.

Tracing

Tracing is useful when you want to find out why a function behaves in an unexpected manner, perhaps because incorrect arguments are being passed.

Tracing causes actions to be taken when a function is called and when it returns. The default tracing actions print the function name and arguments when the function is called and print the values returned when the function returns.

Other actions can be specified. These include entering a break loop when a function is entered or exited, or stepping the function. Trace actions may be conditional.

Several functions can be traced at one time.

When a traced function is traced again, the new trace actions replace the former ones. When a traced function is redefined by evaluation in a buffer, the trace actions are transferred from the old definition to the new definition. When a traced function is redefined while loading a file, the function is untraced and a warning is issued.

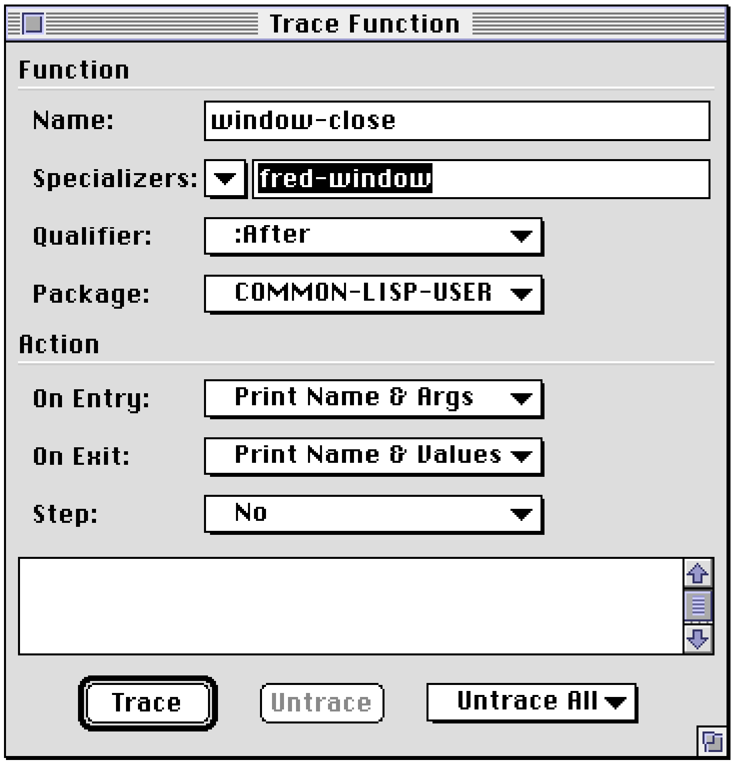
Macros and special forms cannot be traced. Functions that are compiled inline cannot be traced (see *Common Lisp: The Language*, pages 229–230). Note that, by default, self-recursive calls are compiled as inline branches. To effectively trace a function with self-recursive calls, you should declare it not inline.

Tracing is available both through the Trace menu-item on the Tools menu, and through a number of Lisp macros and functions.

The Trace tool

The Trace tool is an interactive interface to the MCL trace mechanism. This tool calls the `trace` macro. The argument to the `trace` function is the string in the Name text edit field. The following figure shows the dialog box for the Trace tool.

■ **Figure 9-6** The Trace dialog box



The Specializers type-in pop-up menu specifies a parameter specializer for the function; the Qualifier pop-up menu specifies an auxiliary method qualifier, which is one of `None`, `:before`, `:after`, or `:around`; the Package pop-up menu specifies the package that defines the function.

The On Entry and On Exit pop-up menus specify different courses of actions. Items in the On Entry menu are Print Name and Args, Break, or No Action; items in the On Exit menu are Print Name and Values, Break, or No Action. The Step pop-up menu specifies whether the function should be stepped, or simply executed.

Untrace removes the trace from the most recently traced function.

Untrace All pops up a list of all functions currently being traced and the item `Untrace All.` You may select the function from which to remove the trace, or remove all traces.

Expressions used for tracing

The following macros, functions and variables are used to invoke and control tracing.

`trace` [*Macro*]

Syntax `trace` {*spec* `|` `(`*spec* {*option modifier*}`)`}

Description The `trace` macro encapsulates the function or method specified by each *spec*, causing the actions specified by the options. When no options are specified, the default actions print arguments on entry and values on exit.

Invoking `(trace)` without arguments returns a list of the functions currently being traced. If no functions are currently being traced, `(trace)` returns `nil`.

Arguments

spec The specification of the function to be traced. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* {*qualifiers*} `(`*specializer* {*specializer*}`*))`.

option An option that specifies an action to be performed. The following options and their modifiers are supported:

`:before` Specifies the action to be taken before the traced function is called. The `:before` keyword must be followed by a modifier:

`:print` Prints the name and arguments to the function before the function is called.

`:break` Prints the name and arguments to the function and enters a break loop before the function is called. You can examine the Stack Backtrace, perform operations in the Listener, and continue if desired.

lisp-function
If the `:before` option is a function, it is called before the traced function is called. The arguments to *lisp-function* are the name of the traced function and the arguments passed to the traced function.

`:after` Specifies the action to be taken after the traced function returns. This keyword must be followed by a modifier, which should be one of the following:

`:print` Prints the name of the function and the returned values.

`:break` Prints the name of the function and the returned values, and enters a break loop. You can examine the Stack Backtrace, perform operations in the Listener, and continue if desired.

lisp-function

If the `:after` option is a function, it is called after the traced function returns. The arguments are the name of the traced function and the values returned by the traced function.

`:step` Specifies whether the traced function should be stepped when it is run. For this option to be effective, the function needs to have been compiled with the variable `*save-definitions*` set to `t` or loaded with `*compile-definitions*` set to `nil`. In addition, stepping will not work if the most recent definition comes from a `.fasl` file unless the file was compiled with `*fasl-save-definitions*` set to true.

The `:step` keyword must be followed by a modifier, which is either `t` or a function whose arguments are the name of the traced function and the arguments passed to the traced function. If it is `t` or if the function returns non-`nil`, then the traced function is stepped; otherwise it is run without stepping.

Examples

Here is an example of tracing the function `fact`.

```
? (defun fact (num)
         (declare (notinline fact))
         (if (= num 0)
           1
           (* num (fact (- num 1)))))
FACT
? (trace fact)
NIL
? (fact 5)
 Calling (FACT 5)
  Calling (FACT 4)
   Calling (FACT 3)
    Calling (FACT 2)
     Calling (FACT 1)
      Calling (FACT 0)
      FACT returned 1
     FACT returned 1
    FACT returned 2
   FACT returned 6
```

```
  FACT returned 24
 FACT returned 120
120
```

Here are some examples of the syntax of `trace` and their results. This prints before but not after.

```
? (trace (fact :before :print))
? (fact 5)
 Calling (FACT 5)
  Calling (FACT 4)
   Calling (FACT 3)
    Calling (FACT 2)
     Calling (FACT 1)
      Calling (FACT 0)
120
```

This example breaks before and prints after.

```
? (trace (fact :before :break
               :after :print))
```

This example breaks on entry with an odd argument.

```
? (trace (fact :before
               #'(lambda (func &rest args)
               "only break if number is odd"
               (if (evenp (car args))
                 (format t "~&Calling ~s~%"
                         (cons func args))
                 (break "on calling ~s"
                         (cons func args))))))
```

This example breaks before an instance of the class `foo` is initialized.

```
? (trace ((:method initialize (foo)) :before :break))
```

This example steps through the function.

```
? (trace (fact :step t))
```

This example steps through even invocations of the function.

```
? (trace (fact :step
             (lambda (name &rest args)
             (declare (ignore name))
             (evenp (car args)))))
```

---

untrace [*Macro*]

Syntax `untrace` {*spec*}

Description The `untrace` macro stops each *spec* from being traced. Notices will not be printed when the function enters or returns. The macro returns a list of the functions that are no longer being traced.

If no *spec*s are specified, all traced functions are untraced.

If you untrace a function that wasn't traced in the first place, no action is taken.

Argument *spec* The specification of the function to be untraced. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* {*qualifiers*} `(`*specializer* {*specializer*}`))`.

`*trace-print-level*` [*Variable*]

`*trace-print-length*` [*Variable*]

Description The `*trace-print-level*` and `*trace-print-length*` variables are used to set the values of `*print-level*` and `*print-length*` during trace operations.

`*trace-level*` [*Variable*]

Description The `*trace-level*` variable specifies the depth of calls to the traced function. Each time the traced function is called, this number is incremented. Each time the traced function returns, it is decremented.

Example

This example begins stepping `fact` after the first five calls.

```
? (trace (fact :step
            (lambda (number &rest args)
              (declare (ignore number args))
              (> *trace-level* 5))))
FACT
```

`*trace-max-indent*` [*Variable*]

Description The `*trace-max-indent*` variable specifies the maximum number of spaces to indent trace output. (Normally, trace output is indented one space for each level of nesting.) The default value is 40.

trace-tab [*Function*]

Syntax `trace-tab`

Description The `trace-tab` function outputs the appropriate number of spaces and vertical bars in `*trace-output*`, given the current value of `*trace-level*`.

trace-bar-frequency [*Variable*]

Description The `*trace-bar-frequency*` variable determines whether and how often vertical bars are printed in trace output. If the value of `*trace-bar-frequency*` is `nil` (the default value), no vertical bars are printed.

Example

```
? (trace fact)
nil
? (setq *trace-bar-frequency* 2)
2
? (fact 5)
Calling (fact 5)
|Calling (fact 4)
| Calling (fact 3)
| |Calling (fact 2)
| | Calling (fact 1)
| | |Calling (fact 0)
| | |fact returned 1
| | fact returned 1
| |fact returned 2
| fact returned 6
|fact returned 24
fact returned 120
120
? (setq *trace-bar-frequency* nil)
nil
? (fact 3)
Calling (fact 3)
 Calling (fact 2)
  Calling (fact 1)
   Calling (fact 0)
   fact returned 1
  fact returned 1
 fact returned 2
fact returned 6
6
```

Advising

The `advise` macro can be thought of as a more general version of `trace`. It allows code that you specify to run before, after, or around a given function, for the purpose of changing the behavior of the function. Each piece of added code is called a piece of advice. Each piece of advice has a unique name, so that you can have multiple pieces of advice on the same function, including multiple `:before`, `:after`, and `:around` pieces of advice.

The unique `:name` and the `:when` keyword serve to identify the piece of advice. A later call to `advise` with the same values for the `:name` and `:when` keywords replaces the existing piece of advice, but a call with different values does not.

advise [*Macro*]

Syntax `advise` *spec form* `&key` *when name define-if-undefined*

Description The `advise` macro adds a piece of advice to the function or method specified by *spec* according to *form.*

Arguments

spec The specification of the function on which to put the advice. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* `{`*qualifiers*`}` `(`*specializer* `{`*specializer*`}))`.

form A form to execute before, after, or around the advised function. The *form* can refer to the variable *arglist* that is bound to the arguments with which the advised function was called. You can exit from *form* with `(return)`.

name A unique name that identifies the piece of advice.

when An argument that specifies when the piece of advice is run. There are three allowable values. The default is `:before`, which specifies that *form* is executed before the advised function is called. Other possible values are `:after`, which specifies that *form* is executed after the advised function is called, and `:around`, which specifies that *form* is executed around the call to the advised function. You should use `(:do-it)` in *form* to indicate invocation of the original definition.

define-if-undefined
An argument that determines whether to define the advised function if it is undefined. The default is `nil`, in which case an error is signaled if the function is undefined.

Examples

Here are some examples of the use of `advise`.

The function `foo`, already defined, does something with a list of numbers. The following code uses a piece of advice to make `foo` return zero if any of its arguments is not a number. Using `:around` advice, you can do the following:

```
(advise foo (if (some #'(lambda (n)
                           (not (numberp n)))
                       arglist)
                 0
                 (:do-it))
        :when :around :name :zero-if-not-nums)
```

To do the same thing using a `:before` piece of advice:

```
(advise foo (if (some #'(lambda (n)
                           (not (numberp n)))
                       arglist)
                (return 0))
        :when :before :name :zero-if-not-nums)
```

---

unadvise [*Macro*]

Syntax `unadvise` *spec* `&key` *when name*

Description The `unadvise` macro removes the piece or pieces of advice for everything matching *spec, when*, and *name*. When the value of *spec* is `t` and the values of *when* and *name* are `nil`, `unadvise` removes every piece of advice; when *spec* is `t`, *when* is `nil`, and *name* is non-`nil`, `unadvise` removes all pieces of advice with the given name.

Arguments

spec — The specification of the function for which pieces of advice are to be removed. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* {*qualifiers*} `(`*specializer* {*specializer*}`))`.

when — The specification of the when value for the piece of advice to be removed. The allowable values are the same as those for `advise`.

name The unique name of the piece of advice to be removed.

---

advisedp [*Macro*]

Syntax `advisedp` *spec* `&key` *when name*

Description The `advisedp` macro returns a list of existing pieces of advice that match *spec*, *when*, and *name*. When the value of *spec* is `t` and the values of *when* and *name* are `nil`, `advisedp` returns all existing pieces of advice.

Arguments

spec The specification of the function to check for pieces of advice. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* `{`*qualifiers*`} (`*specializer* `{`*specializer*`}))`.

when The specification of the when value for the piece of advice to be removed. The allowable values are the same as those for `advise`.

name A unique name that identifies the piece of advice.

The Inspector

Macintosh Common Lisp supports the Common Lisp `inspect` function with a window-based Inspector.

The Inspector lets you look quickly at any component of one or more data objects. For instance, you can use it to look at the current state of the system data. Double-click any form or component of a form in an Inspector window to bring up a window with a definition of the form or component; double-click any item in that window to bring up its definition, and so on.

Because objects are editable in Inspector windows, you can change the state of system data and other components on the fly. You should be careful about doing so, however; it is generally safe to change the value of a global variable in the Inspector, but you should use the standard interface functions to change the values associated with object keywords.

To see the Inspector, choose Inspect from the Tools menu.You can also call `inspect` on a Lisp object or use the keystroke command Control-X Control-I. If you have an extended keyboard, you can also press the Help key. When you choose Apropos from the Tools menu and double-click a symbol name, Macintosh Common Lisp creates an Inspector window containing information about that symbol.

The Inspector menu

The Inspector menu-item on the Tools menu has a number of submenus. These submenus and their actions are described in the following table.

■ **Table 9-4** Options in Inspector Central

Inspector option	Effect
Record Types	Displays a window that lists all record types.
Record Field Types	Displays a window that lists all record field types.
Inspector Help	Displays a window giving brief help on Inspector commands.
Inspector History	Lists all Lisp objects that have been inspected. Double-clicking one of them creates an Inspector window showing its definition. To begin keeping a history, evaluate the form shown in the initial window.
Disk Devices	Displays a window listing the names of all currently active devices.
Logical Hosts	Displays an Inspector window listing all logical hosts and their physical equivalents.
Packages	Displays an Inspector window that inspects the list returned by the Common Lisp function `(list-all-packages)`.
`*package*`	Displays a window that inspects the value of the Common Lisp variable `*package*`.
`*readtable*`	Displays a window that inspects the value of the Common Lisp variable `*readtable*`.

Inspector functions

The following functions are used with the Inspector.

inspect [*Function*]

Syntax `inspect` *thing*

Description The `inspect` function inspects *thing*.

Argument *thing* Any Lisp data object.

Example

```
? (defun foo (x y)
    (let ((z 10))
      (break)
      (+ x y z)))
FOO
? (inspect 'foo)
#<INSPECT-DIALOG "Symbol: FOO" #x5DE9F9>
```

top-inspect-form [*Function*]

Syntax `top-inspect-form`

Description The `top-inspect-form` function returns the form being inspected by the active Inspector window, or `nil` if there are no active Inspector windows.

inspector-disassembly [*Variable*]

Description The `*inspector-disassembly*` variable specifies whether the Inspector displays a disassembly when you inspect a function.

If the value of this variable is true, the Inspector displays a disassembly.

If the value of this variable is `nil` (the default), no disassembly is displayed.

`@` *[Variable]*

Description The `@` variable is bound to the last object that was cut or copied. It is used primarily to communicate values between an Inspector window and the Listener.

The Apropos tool

The Apropos tool performs `apropos` on a user specified string. The Name scrolling-list displays all symbols of a specified type that `apropos` found containing the string.

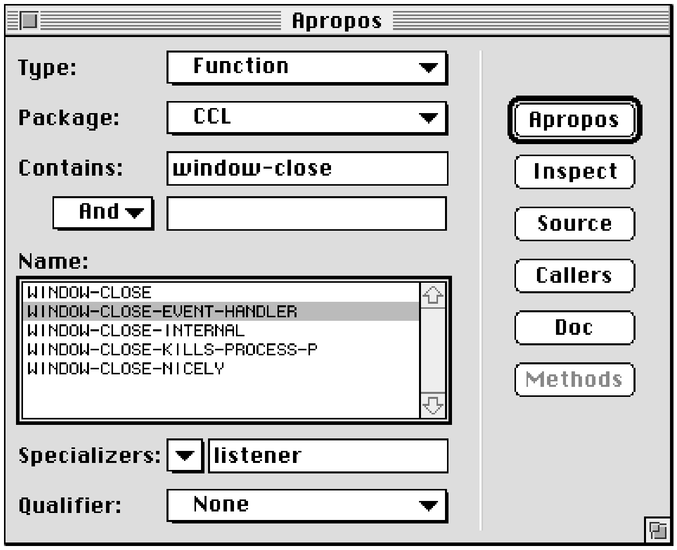
The Type pop-up menu specifies the symbol's type. Items in the Type pop-up menu are Function, Variable, Class, Macro, and All. Only symbols with values of that type will be shown.

Items in the Package pop-up menu limit your request to symbols in particular packages.

The boolean operators And, Or, and Not allow you to display symbols which contain the specificed combinaton of two strings.

The Specializers type-in pop-up menu specifies a parameter specializer for the symbol. The Qualifier pop-up menu specifies an auxiliary method qualifier with options `None`, `:before`, `:after`, and `:around`.

■ **Figure 9-7** The Apropos dialog box



The buttons in the Apropos dialog box have the following functions:

Apropos Performs `apropos` on a string in the Contains text edit field.

Inspect Displays an Inspector window for the symbol highlighted in the Name scrolling-list.

Source Attempts to find the source code for the definition of the symbol highlighted in the Name scrolling-list. If the symbol was defined when the value of the variable `*record-source-file*` was true, the source code file is known.

Callers Displays a list of functions that call the symbol highlighted in the Name scrolling-list, and allows you to select and edit a caller.

Doc Displays the documentation string for the symbol highlighted in the Name scrolling-list, if a documentation string is available. Documentation strings are available if the symbol was defined when the value of `*save-doc-strings*` was true and if the symbol definition contains a documentation string. Documentation strings are also available for all the external symbols in the `COMMON-LISP` and `CCL` packages if the MCL Help file is present in the folder containing the MCL application.

Methods Displays a list of methods that specialize on the class selected in the name scrolling-list. This button is enabled only when a class is selected. A dialog box contains the list of methods and a Find It button. Double-clicking on a method in the list or pressing the Find It button opens a Fred window containing the source code for the method.

The Get Info tool

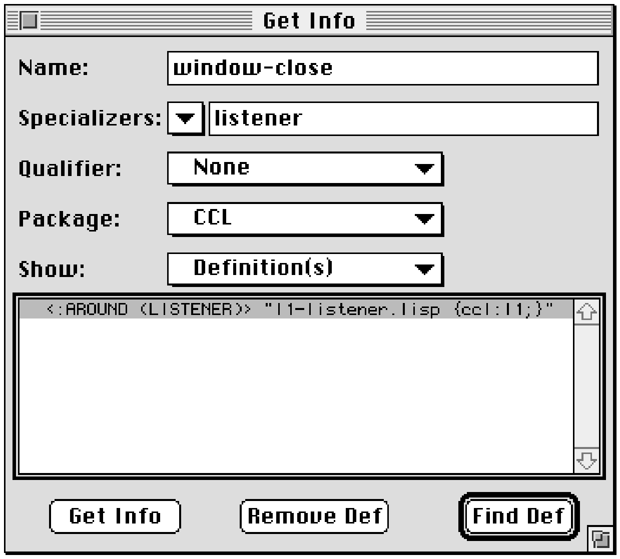
The Get Info tool shows information about a symbol. The information shown depends on the item chosen in the Show pop-up menu.

A symbol is entered in the Name text edit field. The Package pop-up menu limits the search for the symbol to particular packages. The Specializers type-in pop-up menu shows the classes on which the symbol is defined and specifies parameter specializers for the symbol. The Qualifier pop-up menu specifies an auxiliary method qualifier using the option `None`, `:before`, `:after`, or `:around.`

The Show pop-up menu allows you to choose exactly what information you want to see about the symbol, Definition(s), Applicable Methods, Callers, Documentation, and Inspector. The items Definition(s), Applicable Methods, and Callers display the relevant source code in a Fred window if you double-click on an item in the list or press the Find It button. The Documentation and Inspector items display a documentation string and an Inspector window, respectively.

The following figure shows the Get Info dialog box.

■ **Figure 9-8** The Get Info dialog box



The buttons on the bottom have the following functions:

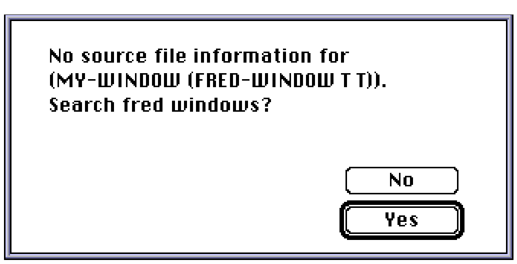
Get Info Displays the information requested about the symbol.

Remove Def Removes the binding from the symbol in memory. The symbol's name in the display is marked with an X to indicate that it is now unbound. The definition of the symbol in the source file is not affected.

Find Def Displays the definition of the symbol from the source file.

The Get Info tool also finds user-defined symbols in Fred windows. If Get Info cannot find the symbol, it asks if you want to search your Fred windows for the symbol, as shown in the following figure.

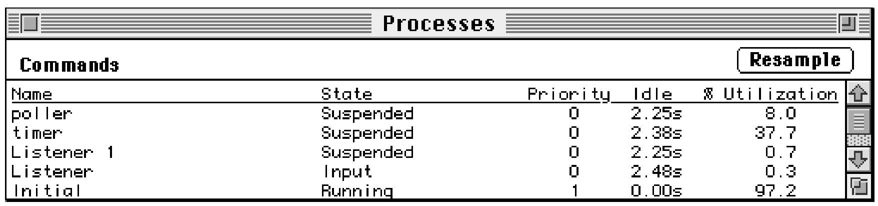
■ **Figure 9-9** The Get Info modal dialog box



The Processes tool

The Processes tool displays information about all existing Macintosh Common Lisp processes. After selecting this item, an Inspector window appears on your screen. The Inspector window lists the name, state, priority, idle status, and utilization of each process.

■ **Figure 9-10** The Processes Inspector window



The `% Utilization` column shows cumulative values since process run times were cleared. The `Clear run times` item in the Inspector's Commands menu resets the values in the `% Utilization` column. The Initial process includes time spent in other applications.

For more information on multiple processes in Macintosh Common Lisp, see Chapter 12: Processes.

Miscellaneous Debugging Macros

The following macros are useful for testing and optimizing code and for tracing program flow.

`time` [*Macro*]

Syntax `time` *form*

Description The `time` macro executes *form*, prints the duration of execution (with a special note on garbage collection time, if any), and returns the value returned by *form*. The `time` macro is useful for testing and optimizing code.

Argument *form* Any Lisp form. The form should not be quoted.

Example

```
? (defun make-numlist (positive-number &aux result)
    "returns a list of numbers between 0 and
     positive-number - 1"
    (dotimes (x positive-number)
      (setq result (append result (list x))))
       ;APPEND is inefficient here.
    result)
MAKE-NUMLIST

? (time (make-numlist 100))
(MAKE-NUMLIST 100) took 449 ticks (7.483 seconds) to run.
Of that, 444 ticks (7.400 seconds) was spent in GC.
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99)
? (defun make-faster-numlist (positive-number &aux result)
    "returns the same list more quickly"
    (dotimes (x positive-number)
      (setq result (cons x result)))
    ;This is more efficient.
      (nreverse result))
MAKE-FASTER-NUMLIST
? (time (make-faster-numlist 100))
(make-faster-numlist 100) took 0 ticks (0.000 seconds) to run.
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99)
```

`print-db` [*Macro*]

Syntax `print-db` {*form*}*

Description The `print-db` macro is equivalent to `progn`, except that each form is printed as it is evaluated. The *form* itself and the result of evaluating *form* are both printed (unless *form* is a string, in which case it is printed only once). The value of the last *form* is returned.

If multiple forms are given, they are printed on separate lines. Printed output is sent to `*error-output*`, which makes the Listener the active window before printing. Like `progn`, `print-db` returns the value of the last form.

The `print-db` macro is useful for tracing program flow and for checking the values of variables at various points in a program.

Argument	*form*	Any Lisp form.

Chapter 10:

Events

Contents

This chapter explains how Macintosh Common Lisp processes events. It describes built-in handlers and functions that give event-related information. It discusses the MCL event management architecture. Finally, it describes two processes involved in event management: updating the cursor and accessing the Macintosh Clipboard.

You should read this chapter to understand or program events and event handlers in Macintosh Common Lisp.

If you are creating handlers for Apple events, you should read this chapter and then read Chapter 11: Apple Events.

Implementation of events in Macintosh Common Lisp

Users generate events as a way of directing program flow. Typical events are keystrokes and mouse clicks. Events interrupt a program and often require a response. Whenever possible, Macintosh programs should be event driven.

Macintosh Common Lisp automatically handles events in a separate process. When a user generates an event, the current program is interrupted and an event handler handles the event. Program execution does not resume until the event-handling function returns. Further event processing is also deferred until the event-handling function returns. For this reason, the computer may not respond to user actions until the event handling is finished.

Many user programs do not need to handle events explicitly. For those programs that do, several different event-handling methods are available. In order of increasing complexity these are

- defining methods associated with specific types of events in a view
- defining all methods associated with a view
- defining a hook procedure that has first priority in processing all events
- disabling all background event processing, and handling events with an event loop

Most programming languages for the Macintosh computer support only the last, and most difficult, method of event handling. MCL programs rarely need to do anything more complex than the first method.

Programs can be initiated from within an event handler; create a separate process or use the function `eval-enqueue`, which lets an event initiate a process with event processing enabled.

How an event is handled

The MCL event system gets each event from the Macintosh Operating System in turn and binds `*current-event*` to it. The event system then determines the type of the event and calls the appropriate event-handling function on the relevant view. If the event is a mouse click, the relevant view is the view in which the click occurred. If the event is a keystroke, the relevant view is the active (frontmost) window.

Functions that end with "–event–handler" should be called only by the event system.

Many of the default event-handling methods do nothing, although they are called whenever an event of the appropriate type is processed. These handlers exist so that they may be shadowed by any subclass of `view` that needs to process events of that type.

Some event handlers defined on views do nothing more than invoke the same event handler on each subview. In this way nested views and subviews are processed.

Event-handling functions assume that the `*current–event*` variable is bound to a valid event record (see "Chapter 16: OS Entry Points and Records"). They may also call the current-event information functions (listed in "Event information functions" on page 372), which depend on `*current-event*` being bound.

MCL built-in event handlers

The following are standard event handlers in Macintosh Common Lisp.

view-click-event-handler [*Generic function*]

Syntax

`view-click-event-handler` (*view* `simple-view`) *where*
`view-click-event-handler` (*view* `view`) *where*
`view-click-event-handler` (*window-or-item* `fred-mixin`) *where*
`view-click-event-handler` (*item* `dialog-item`) *where*
`view-click-event-handler` (*item* `table-dialog-item`) *where*
`view-click-event-handler` (*item* `scroll-bar-dialog-item`) *where*
`view-click-event-handler` (*menu* `pop-up-menu`) *where*

Description

The generic function `view-click-event-handler` is called by the event system on the window containing the view whenever the user clicks a view or subview.

The `view-click-event-handler` function is not called when the user clicks the title bar, close box, zoom box, or size box of a window. The method for `simple-view` does nothing. Specialized windows provided by the system, such as Fred windows, have special behavior.

If you define any window methods, they must call `call-next-method`.

Arguments

view	A simple view, view, or subview, such as a window or dialog item.
window-or-item	A Fred window or Fred dialog item.
item	A dialog item, table dialog item, or scroll-bar dialog item.
menu	A pop-up menu.
where	The cursor position when the user clicks, expressed in local window coordinates.

Example

The following code displays the cursor coordinates whenever the user clicks `my-window`. (As a subclass of `view`, `window` inherits the `view-click-event-handler` method for `view`.)

```
? (defclass my-window (window)())
#<STANDARD-CLASS MY-WINDOW>

? (defmethod view-click-event-handler
((window my-window) where)
    (print (point-string where)))
#<Method VIEW-CLICK-EVENT-HANDLER (MY-WINDOW T)>

? (make-instance 'my-window)
#<MY-WINDOW "Untitled" #x410891>
```

view-key-event-handler [*Generic function*]

Syntax

`view-key-event-handler` (*view* `simple-view`) *char*
`view-key-event-handler` (*window* `window`) *char*
`view-key-event-handler` (*item* `fred-dialog-item`) *char*
`view-key-event-handler` (*window-or-item* `fred-mixin`) `*current-character*`

Description

The methods of the generic function `view-key-event-handler` examine the current keystroke and determine what is to be done with it.

The method for `simple-view` calls `ed-beep`. The method for `window` determines whether the key indicates the selection of a default button or indicates a change of the current key handler, then selects the button or passes the keystroke to the appropriate key handler. The method for `fred-mixin` binds the `*current-keystroke*` variable to the keystroke of the current event and runs the Fred command associated with the keystroke. The method for `fred-dialog-item` calls `call-next-method` inside `with-focused-view` and `with-fore-color`.

Arguments *view* A simple view.

window	A window or Fred window.
item	A Fred dialog item.
char	The current keystroke character.
window-or-item	A Fred window or Fred dialog item.

view-activate-event-handler [*Generic function*]

Syntax

`view-activate-event-handler` (*view* `simple-view`)
`view-activate-event-handler` (*view* `view`)
`view-activate-event-handler` (*window* `window`)
`view-activate-event-handler` (*window-or-item* `fred-mixin`)
`view-activate-event-handler` (*item* `table-dialog-item`)
`view-activate-event-handler` (*item* `scroll-bar-dialog-item`)

Description

The generic function `view-activate-event-handler` is called by the event system when the window containing the view is made active. The method for `view` calls `view-activate-event-handler` on each subview. The method for `simple-view` does nothing.

The method for `window` includes highlighting the window and drawing the size box (if there is one).

Arguments

view	A simple view, view, or subview such as a dialog item.
window	A window or Fred window.
item	A Fred dialog item, table dialog item, or scroll-bar dialog item.
window-or-item	A Fred window or Fred dialog item.

view-deactivate-event-handler [*Generic function*]

Syntax

`view-deactivate-event-handler` (*view* `simple-view`)
`view-deactivate-event-handler` (*view* `view`)
`view-deactivate-event-handler` (*window* `window`)
`view-deactivate-event-handler` (*window-or-item* `fred-mixin`)
`view-deactivate-event-handler` (*item* `table-dialog-item`)
`view-deactivate-event-handler` (*item* `scroll-bar-dialog-item`)

Description

The generic function `view-deactivate-event-handler` is called by the event system to deactivate a view. It is called when the window containing the view is active and a different window is made active.

The method for `view` simply calls `view-deactivate-event-handler` on each subview. The method for `window` includes removing the highlight and erasing the size box (if there is one).

Arguments

view	A simple view, view, or subview such as a window or dialog item.
window	A window or Fred window.
item	A Fred dialog item, table dialog item, or scroll-bar dialog item.

`key-handler-mixin` [*Class name*]

Description The class `key-handler-mixin` should be mixed into any class that handles key events. The class `fred-dialog-item` includes `key-handler-mixin`.

`key-handler-list` [*Generic function*]

Syntax `key-handler-list` (*view* `simple-view`)

Description The `key-handler-list` generic function returns the list of key handlers associated with *view*.

Argument *view* A simple view or dialog item.

`current-key-handler` [*Generic function*]

Syntax `current-key-handler` (*window* `window`)

Description The `current-key-handler` generic function returns the current key handler of *window*.

Argument *window* A window.

`set-current-key-handler` [*Generic function*]

Syntax `set-current-key-handler` (*window* `window`) *item* `&optional` *select-all*

Description The generic function `set-current-key-handler` sets the current key handler of *window* to *item*. If *item* is not already the current key handler and *select-all* is true, `set-current-key-handler` selects all of the window.

Arguments

window	A window.
item	A key handler. If *item* is not a key handler, the function signals an error.
select-all	This variable determines whether the entire text of the key handler is highlighted when it is first selected. The default is `t`; that is, all the text is highlighted and can be manipulated at once.

`add-key-handler` [*Generic function*]

Syntax `add-key-handler` (*view* `simple-view`) `&optional` *window*

Description The generic function `add-key-handler` adds a key handler to *view*. It is called by `install-view-in-window` when the view installed is a subclass of `key-handler-mixin`. If *window* has no current key handler, *view* becomes the current key handler.

Arguments

view	A simple view or dialog item.
window	A window to which to add the key handler. The default value is `(view-window` *view*`)`.

`remove-key-handler` [*Generic function*]

Syntax `remove-key-handler` (*view* `simple-view`) `&optional` *window*

Description The generic function `remove-key-handler` removes a key handler from a window. It is called by the method of `remove-view-from-window` for `key-handler-mixin`.

Arguments

view	A simple view or dialog item.
window	A window from which to remove the key handler. The default value is `(view-window` *view*`)`.

`change-key-handler` [*Generic function*]

Syntax `change-key-handler` (*view* `view`)

Description The generic function `change-key-handler` changes the key handler of *view* to the next key handler on `key-handler-list` of *view*.

Argument *view* A simple view or dialog item.

`key-handler-p` [*Generic function*]

Syntax `key-handler-p` (*item* `dialog-item`)
`key-handler-p` (*key-handler* `key-handler-mixin`)

Description The `key-handler-p` generic function checks to see whether *item* is a key handler. When `key-handler-p` is called on an instance of a class one of whose superclasses is `key-handler-mixin`, the function returns `t` unless the key handler is disabled. The method for `dialog-item` returns `nil`.

Arguments

item	A dialog item.
key-handler	A key handler.

`key-handler-idle` [*Generic function*]

Syntax `key-handler-idle` (*view* `simple-view`) `&optional` *dialog*
`key-handler-idle` (*item* `fred-dialog-item`) `&optional` *dialog*

Description The `key-handler-idle` generic function is called periodically via the default `window-null-event-handler` function to allow a key handler to blink a cursor or perform other periodic activities.

The method for `fred-dialog-item` blinks the insertion point and matches parentheses. The method for `simple-view` does nothing.

Arguments

view	A simple view.
item	A Fred dialog item.
dialog	An argument allowing a dialog to be specified. In system-supplied methods, this argument is ignored.

`window-null-event-handler` [*Generic function*]

Syntax `window-null-event-handler` (*window* `window`)
`window-null-event-handler` (*window* `t`)

Description The generic function `window-null-event-handler` is called on the top window (if there is one) whenever the system is idle. It updates the cursor, runs system tasks, and forces output from `*terminal-io*`. If there is no top window, the unspecialized method simply updates the cursor..

Argument *window* A window.

`window-select-event-handler` [*Generic function*]

Syntax `window-select-event-handler` (*window* `window`)

Description The generic function `window-select-event-handler` is called whenever the user clicks an inactive window. The `window-select-event-handler` function may be specialized, for example, to make a window unselectable.

Argument *window* A window.

`window-key-up-event-handler` [*Generic function*]

Syntax `window-key-up-event-handler` (*window* `window`)

Description The generic function `window-key-up-event-handler` is called whenever a key is released after being pressed. The method for `window` does nothing.

Every key pressed by the user actually generates two events: one when the key is pressed and another when the key is released.

The default Macintosh event mask filters out key-up events. To allow key-up events, call `#_SetEventMask` with an appropriate mask. Note that you must reset the event mask before exiting Lisp. For details on event masks, see Macintosh Technical Note 202 and *Inside Macintosh*.

Argument *window* A window.

`window-mouse-up-event-handler` [*Generic function*]

Syntax `window-mouse-up-event-handler` (*window* `window`)

Description The `window-mouse-up-event-handler` generic function is called whenever the user releases the mouse button. The method for `window` does nothing.

Argument *window* A window.

`window-grow-event-handler` [*Generic function*]

Syntax `window-grow-event-handler` (*window* `window`) *where*

Description The generic function `window-grow-event-handler` is called by the event system whenever the user clicks a window's grow box. The method for `window` calls `#_GrowWindow`, then calls `set-view-size` on the window and the new size.

Arguments

window	A window.
where	The position in screen coordinates of the cursor when the mouse button was pressed down.

`window-drag-event-handler` [*Generic function*]

Syntax `window-drag-event-handler` (*window* `window`) *where*

Description The generic function `window-drag-event-handler` is called by the event system whenever a window needs to be dragged. It calls `#_SetClip` and `#_ClipAbove` on the region of the window, copies the contents of the region to the new location of *window*, and calls `set-view-position` on the window and the new position of the upper-left corner of the window.

Arguments

window	A window.
where	The position in screen coordinates of the cursor when the mouse button was pressed down.

`window-zoom-event-handler` [*Generic function*]

Syntax `window-zoom-event-handler` (*window* `window`) *message*

Description The generic function `window-zoom-event-handler` is called by the event system when the user clicks the window's zoom box. It executes the Toolbox calls to zoom the window, then calls `window-size-parts`.

The function `window-size-parts` should be specialized if you want to change the contents of a window whenever the window changes size.

Arguments

window	A window.
message	An integer, `#$inZoomOut` if the window should move to the window's zoom position and size and `#$inZoomIn` if the window should move to the position and size it had before zooming out.

`window-close-event-handler` [*Generic function*]

Syntax `window-close-event-handler` (*window* `window`)

Description The generic function `window-close-event-handler` is called by the event system whenever a window needs to be closed. In the method for `window`, if the Meta key was pressed when the command was given, the command closes all windows in the class of *window*. If the Control key was pressed, *window* is hidden. Otherwise, `window-close` is called on *window*.

Argument *window* A window.

`window-do-first-click` [*Generic function*]

Syntax `window-do-first-click` (*window* `window`)

Description The generic function `window-do-first-click` determines whether the click that selects a window is also passed to `view-click-event-handler`. The default value is `nil`, meaning that the click that selects a window generates no further action.

You can give a window instance or subclass of `window` its own value for `window-do-first-click`.

Argument *window* A window.

Functions for redrawing windows

Whenever a window is created or uncovered, an update event is posted for the window. The next time events are processed, Macintosh Common Lisp recognizes the update event and calls `window-update-event-handler`.

The following functions relate to redrawing windows.

`window-update-event-handler` [*Generic function*]

Syntax `window-update-event-handler` (*window* `window`)

Description The generic function `window-update-event-handler` is called by the event system whenever any portion of the window needs to be redrawn. The window version calls `#_BeginUpdate` to make the `VisRgn` field of the GrafPort the portion that needs to be redrawn, calls `view-draw-contents`, and then calls `#_EndUpdate` to restore the GrafPort `VisRgn` field.

Because event processing occurs asynchronously, `window-update-event-handler` may not be called until a moment after a window is created or uncovered. (In the default environment, this may take up to one-third of a second; see `event-ticks` in "The event management system" on page 375.) This means that anything drawn in the window immediately after it is created or uncovered may be erased when `window-update-event-handler` is first called.

To fix this problem, simply call `event-dispatch` before drawing in the window. The function `event-dispatch` forces the processing of any pending events. Note that it is necessary to call `event-dispatch` only when drawing occurs soon after a window is created or uncovered.

You should not specialize this function except to note that the window has been updated. To get special drawing behavior, you should instead specialize `view-draw-contents`.

Argument *window* A window.

`view-draw-contents` [*Generic function*]

Syntax
`view-draw-contents` (*view* `simple-view`)
`view-draw-contents` (*view* `view`)
`view-draw-contents` (*window-or-item* `fred-mixin`)
`view-draw-contents` (*item* `fred-dialog-item`)
`view-draw-contents` (*item* `table-dialog-item`)
`view-draw-contents` (*item* `scroll-bar-dialog-item`)
`view-draw-contents` (*item* `static-text-dialog-item`)
`view-draw-contents` (*menu* `pop-up-menu`)

Description The generic function `view-draw-contents` is called whenever a view needs to redraw any portion of its contents. The `view` method for `view-draw-contents` erases the area in the window's erase region (for new windows, this is the entire content area) and then calls `view-draw-contents` on each subview. You can specialize this function so that a user-defined view can be redrawn when portions of it are covered and uncovered.

When `view-draw-contents` is called by the event system, the view's clip region is set so that drawing occurs only in the portions that need to be updated. This normally includes areas that have been covered by other windows and then uncovered.

Arguments

view	A view or a simple view.
window	A window.
item	A dialog item.
window-or-item	A Fred window or Fred dialog item.
menu	A pop-up menu.

Examples

The following code creates a window that always has a circle drawn in it:

```
? (require 'quickdraw)
"QUICKDRAW"
? (setq foo (make-instance 'window))
#<WINDOW "Untitled" #x4A3BD9>
? (defmethod view-draw-contents ((window (eql foo)))
  (paint-oval window 10 10 100 100))
VIEW-DRAW-CONTENTS
```

(Note that the circle is drawn only after the first time the window is covered
and uncovered.)

To add an area (rectangle or region) to the invalid region, call the trap `#_InvalRect` or `#_InvalRgn`. Calling these traps forces the posting of an update event for the window. For this reason, calling these traps from inside `view-draw-contents` or `window-update-event-handler` can lead to an infinite loop.

If you want to invalidate several areas before the update is performed, surround the calls to `#_InvalRect` and `#_InvalRgn` with the special form `without-interrupts`, which temporarily suspends updates.

The following call will force the redrawing of the entire window. It doesn't need `without-interrupts` because there is only one call to `#_InvalRect`. If there were several calls to `#_InvalRect`, `without-interrupts` would postpone updating until the end.

```
(with-port wptr
  (#_invalrect :ptr (rref wptr window.portrect)))
```

The `view-draw-contents` function is not strictly an event handler, since it may be called at any time, not only during event processing. For example, you can use `view-draw-contents` to implement the redrawing that occurs during scrolling, or you can use it to implement a generalized printing mechanism. (For an example, see the file `scrolling-windows.lisp` in your Examples folder.)

`window-draw-grow-icon` [*Generic function*]

Syntax `window-draw-grow-icon` (*window* `window`)

Description The generic function `window-draw-grow-icon` is called when the size box in the lower-right corner of a window must be redrawn. You may need to call this function explicitly if you draw over the size box.

When a window is inactive (that is, not the frontmost window), `window-draw-grow-icon` erases the inside of the size box.

Argument *window* A window.

Event information functions

The following functions give event-related information. To bypass these functions, programs can simply examine `*current-event*` during event handling. (See "Chapter 16: OS Entry Points and Records," for techniques used in examining records.)

`view-mouse-position` [*Generic function*]

Syntax `view-mouse-position` (*view* `simple-view`)
`view-mouse-position` (*view* `null`)

Description The generic function `view-mouse-position` returns the cursor position as a point expressed in the view's local coordinates. The point is returned as an integer (for a description of points, see "Chapter 2: Points and Fonts"). This function may be called at any time, not just during event processing. The coordinates may be negative, or outside of the view's PortRect, depending on the position of the cursor.

The function `(view-mouse-position nil)` returns the cursor position expressed in screen coordinates.

Argument *view* A simple view.

Example

See the example under `mouse-down-p`.

mouse-down-p

[*Function*]

Syntax `mouse-down-p`

Description The `mouse-down-p` function returns `t` if the mouse button is pressed and `nil` otherwise. This function may be called at any time, not only during event processing.

Examples

The following example prints the mouse position in window coordinates until the mouse is clicked.

```
(do () ((mouse-down-p))
  (print (point-string (view-mouse-position (front-
window)))))
```

The following example prints the mouse position in screen coordinates until the mouse is clicked.

```
(do () ((mouse-down-p))
  (print (point-string (view-mouse-position nil))))
```

double-click-p

[*Function*]

Syntax `double-click-p`

Description The `double-click-p` function returns `t` if the click currently being processed was the second half of a double-click. Double-clicks take into account the timing as well as the spacing of consecutive clicks.

The `double-click-p` function always returns `nil` if called from outside event processing. It also returns false if the first click activated the window and `window-do-first-click` is false.

multi-click-count

[*Variable*]

Description The `*multi-click-count*` variable is incremented during event processing if the current event is part of a series of multiple clicks. It is reset to 1 when there is a mouse click that is not part of a series.

Determination of whether a click is part of a series is done as for `double-click-p`.

double-click-spacing-p [*Function*]

Syntax `double-click-spacing-p` *point1 point2*

Description The function `double-click-spacing-p` is called by `double-click-p` to see whether two clicks should count as a double-click. It is also used to determine whether to increment `*multi-click-count*`.

Macintosh guidelines specify that if the cursor is moved excessively between clicks, the clicks do not count as a double-click.

The function `double-click-spacing-p` returns false if *point1* and *point2* are separated by more than 4 pixels, horizontally or vertically. If they are within 4 pixels of each other, both horizontally and vertically, the function returns true.

Arguments

point1	The cursor position during the first click.
point2	The cursor position during the second click.

command-key-p [*Function*]

control-key-p [*Function*]

option-key-p [*Function*]

shift-key-p [*Function*]

caps-lock-key-p [*Function*]

Syntax

```
command-key-p
control-key-p
option-key-p
shift-key-p
caps-lock-key-p
```

Description Each of these functions has two meanings, depending on whether they are called during event processing or outside it.

If called during event processing, they return true if the corresponding key was pressed during the event; otherwise, they return false. If called outside of event processing, they return true if the key is currently pressed; otherwise, they return false.

Note that some Macintosh keyboards do not have a Control key.

The event management system

This section describes the overall architecture used for implementing event handling in Macintosh Common Lisp.

`event-dispatch` [*Function*]

Syntax `event-dispatch &optional` *idle*

Description The `event-dispatch` function is called periodically as a background process. The `event-dispatch` function calls `#_WaitNextEvent` and binds the value of `*current-event*` for the duration of the event processing. It then calls `*eventhook*` if `*eventhook*` is not `nil`. If `*eventhook*` returns true, the processing of the event stops. If `*eventhook*` returns false, the event is passed to the system event handlers. Finally, `event-dispatch` checks for deferred Apple events.

If you create a program with a loop that checks for events, you should probably include a call to `event-dispatch` inside the loop. This improves the response time when events occur.

Argument *idle* An argument representing whether the main Lisp process is idle. The default is the value of `*idle*`, which is true when the main Lisp process is idle and `nil` otherwise. The function `event-dispatch` calls `get-next-event` with an event and the value of *idle*.

`get-next-event` [*Function*]

Syntax `get-next-event` *event* `&optional` *idle mask sleep-ticks*

Description The `get-next-event` function calls `#_WaitNextEvent` to get an event. It disables and reenables the clock sampled by `get-internal-run-time`. (MultiFinder may do a context switch.)

After `#_WaitNextEvent` returns, the function reschedules the `event-dispatch` task, which is the usual caller of `get-next-event`.

Arguments

event An event record allocated on the stack or the heap.

idle Used to determine the default value of *sleep-ticks.* The default value is `*idle*`, which is true if `get-next-event` is called via `event-dispatch` from the top-level loop when the Listener is waiting for input.

mask	This is the `EventMask` argument for `#_WaitNextEvent`, a fixnum. The default is `#$everyEvent`.
sleep-ticks	This is the `Sleep` argument to `#_WaitNextEvent`. It determines how many ticks are given to other applications under MultiFinder if no event is pending. The default is determined by the values of the *idle* argument and the global variables `*idle-sleep-ticks*`, `*foreground-sleep-ticks*`, and `*background-sleep-ticks*`. If Macintosh Common Lisp is running in the foreground, then the default is `*idle-sleep-ticks*` if the value of *idle* is true; otherwise, the default is `*foreground-sleep-ticks*`. If Macintosh Common Lisp is running in the background, then the default is `*background-sleep-ticks*` unless that value is `nil`, in which case the default is the same as when Macintosh Common Lisp is running in the foreground.

`*current-event*` [*Variable*]

Description

The `*current-event*` variable holds the event record currently being processed. This is bound by `event-dispatch` and is valid only during event processing. The fields of `*current-event*` may be accessed using `rref` (for details see “Chapter 16: OS Entry Points and Records,” and *Inside Macintosh*).

The definition of the event record type is

```
(defrecord Event
  (what integer)
  (message longint)
  (when longint)
  (where point)
  (modifiers integer))
```

`*eventhook*` [*Variable*]

Description

The `*eventhook*` variable provides a user hook into the event system. A program can store a function of no arguments in this global variable. The stored function is given the first opportunity to handle all event processing. If the function returns true, the event system assumes the event has been handled and no further processing is done. If the function returns false, the event system assumes the event hasn’t been handled and the normal event handlers are invoked.

If *eventhook* is a list of functions with no arguments, they will be called sequentially until either one of them returns true or the list is exhausted. In the latter case, normal event processing occurs.

An *eventhook* function can be used to perform periodic tasks (because it is called whenever there is an event, including a null event).

Note that a slow *eventhook* function can significantly slow down Macintosh Common Lisp.

---

idle [*Variable*]

Description The *idle* variable signals the event system that the main Lisp process is idle. This changes the sleep time that event dispatch gives to the trap #_WaitNextEvent. This variable is normally bound to true by the read loop as the loop waits for input, and by modal-dialog.

---

idle-sleep-ticks [*Variable*]

Description The *idle-sleep-ticks* variable holds the value of the sleep time given to #_WaitNextEvent when Macintosh Common Lisp is idle. The initial value is 5.

---

foreground-sleep-ticks [*Variable*]

Description The *foreground-sleep-ticks* variable holds the value of the sleep time given to #_WaitNextEvent when Macintosh Common Lisp is running. The initial value is 0.

---

background-sleep-ticks [*Variable*]

Description The *background-sleep-ticks* variable holds the value of the sleep time given to #_WaitNextEvent when Macintosh Common Lisp is in the background. The initial value is 5. .

---

event-ticks [*Function*]

Syntax event-ticks

Description The `event-ticks` function returns the number of ticks (sixtieths of a second) between calls to `event-dispatch`. This number is applicable when code is running. When Lisp is idling in the main read-eval-print loop, `event-dispatch` is called as close to continuously as possible.

This value is reset on every suspend and resume event, according to the values in `*foreground-event-ticks*` and `*background-event-ticks*`.

---

set-event-ticks [*Function*]

Syntax `set-event-ticks` *n*

Description The `set-event-ticks` function sets the number of ticks between calls to `event-dispatch` to *n*.

If *n* is too low, `event-dispatch` is called too often, and the system may get bogged down by event processing. If it is too high, the system may not respond smoothly to events. To keep the insertion bar blinking smoothly, for example, a sleep time of 12 to 20 ticks is recommended. This will yield 3 to 5 idle events per second.

This function is called on every suspend and resume event, with the argument `*foreground-event-ticks*` or `*background-event-ticks*`.

Argument *n* An integer determining the number of ticks.

---

foreground-event-ticks [*Variable*]

Description The `*foreground-event-ticks*` variable holds the appropriate value for `event-ticks` when Lisp is the foreground application. The initial value is 20.

---

background-event-ticks [*Variable*]

Description The `*background-event-ticks*` variable holds the appropriate value for `event-ticks` when Lisp is a background application. The initial value is 5.

---

window-event [*Generic function*]

Syntax `window-event` (*window* `window`)

Description The `window-event` generic function is called by `event-dispatch` to get a window to handle an event. This function is called only when the event system determines the appropriate window. The method of `window-event` for `window` checks the type of the event and calls the appropriate event handler. The `window-event` function should be specialized in windows that need to do something in addition to or different from the default behavior for many types of events.

Argument *window* A window.

`without-interrupts` [*Special form*]

Syntax `without-interrupts` {*form*}*

Description The `without-interrupts` special form executes *form* with all event processing disabled, including `abort`.

You should use `without-interrupts` sparingly because anything executed dynamically within it cannot be aborted or easily debugged.

However, you must often use `without-interrupts` in code that causes a window to be redisplayed. If you need to invalidate a number of regions in a window, do it inside a `without-interrupts` form to prevent multiple redisplays.

Argument *form* Zero or more Lisp forms.

`*break-loop-when-uninterruptable*` [`Variable`]

Description Controls the interaction of break loops and `without-interrupts`. If set to true, a break loop can occur during a `without-interrupts`. The `without-interrupts` is suspended for the duration of the break loop. The default value is `t`.

The cursor and the event system

The *cursor* is the screen image whose motion is controlled by the mouse. As the user moves the mouse on the desktop, the cursor moves correspondingly on the screen.

The cursor often changes shape as it moves over different areas of the screen. For example, when it is on the menu bar or scroll bars, it is shaped like an arrow; when inside a text window, the cursor is shaped like an I-beam.

A program can control the appearance of the cursor in four ways:

- You can define methods for `view-mouse-enter-event-handler` and `view-mouse-leave-event-handler`, specialized on a subclass of `simple-view`. These functions are called when the mouse cursor enters and leaves the area of the view. A possible side effect may be to change the shape of the cursor, for example, from an arrow to an I-beam.
- A view may have a method for the `view-cursor` generic function. The event system sets the cursor according to this method whenever the cursor is over the view.
- The `with-cursor` macro may surround a series of forms. The cursor assumes a given shape for the duration of the macro.
- The variable `*cursorhook*` may be bound to a function or cursor, giving you complete control over the appearance of the cursor.

`view-cursor` [*Generic function*]

Syntax

`view-cursor` (*view* `simple-view`) *point*
`view-cursor` (*view* `view`) *point*
`view-cursor` (*window* `window`) *point*
`view-cursor` (*item* `basic-editable-text-dialog-item`) *point*
`view-cursor` (*window* `fred-window`) *point*

Description

The `view-cursor` generic function determines the cursor shape whenever the window containing the view is active and the cursor is over it. The `view-cursor` function is called by `window-update-cursor`.

Arguments

view	A view or simple view.
window	A window or Fred window.
item	A dialog item.
point	The position of the cursor, expressed as a point.

`window-update-cursor` [*Generic function*]

Syntax

`window-update-cursor` (*window* `null`) *point*
`window-update-cursor` (*window* `window`) *point*

Description

The generic function `window-update-cursor` is called by `update-cursor` whenever the cursor is over the window.

When the mouse is over the front window or any floating window, the `window-update-cursor` method for the `window` class sets the variable `*mouse-view*` to the view containing the mouse, using `find-clicked-subview`. The `window-null-event-handler` method for the `window` class calls `update-cursor`, which calls `*cursorhook*`. The function that is the initial value of `*cursorhook*` calls `window-update-cursor`, which sets the cursor using the value returned by `view-cursor`.

The method for `window` simply sets the cursor to the result of calling the generic function `view-cursor` on the clicked subview of *window* if there is one; otherwise it sets the cursor to the result of calling `window-cursor` on the window.

The `null` method sets the cursor to the value of `*arrow-cursor*`.

The `window-update-cursor` function should be shadowed if the cursor must change according to what part of the window it is over.

Arguments

window	A window or Fred window.
point	The position of the cursor, given in the window's local coordinates.

---

view-mouse-enter-event-handler [*Generic function*]

view-mouse-leave-event-handler [*Generic function*]

Syntax

`view-mouse-enter-event-handler` (*view* `simple-view`)
`view-mouse-leave-event-handler` (*view* `simple-view`)

Description

The methods of these generic functions for `simple-view` do nothing. You specialize them to create mouse-sensitive items.

Argument

view	A simple view.

---

with-cursor [*Macro*]

Syntax

`with-cursor` *cursor* {*form*}*

Description

The `with-cursor` macro executes zero or more forms with `*cursorhook*` bound to *cursor*.

Arguments

cursor	A cursor record or a `'CURS'` resource ID (see *Inside Macintosh* for details on `'CURS'` resource IDs).
form	Zero or more forms to be executed within the body of the macro.

`*cursorhook*` [*Variable*]

Description The `*cursorhook*` variable may be bound to a function or cursor, giving you complete control over the appearance of the cursor. This variable is bound by `with-cursor`.

If the value of this variable is non-`nil`, then no other cursor functions are called. If the value of `*cursorhook*` is a function, it is called repeatedly in the background and has complete control over the state of the cursor at all times. If it is not a function, it should be a cursor record or a `'CURS'` resource ID.

Its initial value is an internal function that allows events to alter the cursor normally.

`update-cursor` [*Function*]

Syntax `update-cursor &optional` *hook*

Description The `update-cursor` function does the actual work of cursor handling. If *hook* is a function or symbol, it is called with no arguments; otherwise, `set-cursor` is called with *hook.*

The `update-cursor` function is called periodically by the global event-handling system. It is not usually necessary to call this function directly, but it may be called to make sure that the cursor is correct at a particular time.

Argument *hook* A function, symbol, or cursor. The default value is `*cursorhook*`.

`set-cursor` [*Function*]

Syntax `set-cursor` *cursor*

Description The `set-cursor` function sets the cursor to *cursor*.

Argument *cursor* A cursor record or a `'CURS'` resource ID.

◆ *Note:* If `set-cursor` is called from anywhere except within a `window-update-cursor` function, a function that is the value of `*cursorhook*`, or a `without-interrupts` special form, the event system's background cursor handling immediately resets the cursor to some other shape. If *cursor* is not of an acceptable type, then no action is taken. To prevent the system from hanging at cursor update time, no error is signaled.

`*arrow-cursor*` [*Variable*]

Description The `*arrow-cursor*` variable specifies the standard north-northwest-arrow cursor shape.

`*watch-cursor*` [*Variable*]

Description The `*watch-cursor*` variable specifies the watch-face shape shown during time-consuming operations, when event processing is disabled.

`*i-beam-cursor*` [*Variable*]

Description The `*i-beam-cursor*` variable specifies the I-beam shape used when the cursor is over an area of editable text.

Event handlers for the Macintosh Clipboard

Data that can be cut and pasted comes in different forms, for example, ASCII text, PICT format graphics, and stylized text. Macintosh Common Lisp provides a simple model for accessing the Macintosh scrap, the structure that supports the Macintosh Clipboard. The Clipboard is accessed through a simple handler, the scrap handler.

Macintosh Common Lisp uses scrap handlers, with one handler for each type of scrap data. The scrap handlers are stored in an association list of the form `(`*scrap-type-keyword . scrap-handler*`)`.

The scrap-type keyword should have a four-character print name. This name is used as an `OStype` data type when the scrap type is communicated to the Macintosh Operating System. (For full details on `OStype` data types, see *Inside Macintosh*.)

In the initial MCL environment, scrap handlers are defined for simple text, formatted Fred text, and Lisp code. You can add handlers for other data types.

When defining new handlers, you should look at the file `pict-scrap.lisp` in the MCL Examples folder to learn how to define a scrap handler for `PICT`s.

For full details on the operation of the Clipboard, you should also consult *Inside Macintosh.*

MCL expressions relating to scrap handlers and scrap types

The following MCL expressions relate to defining scrap handlers and scrap types.

`get-scrap` [*Function*]

Syntax `get-scrap` *scrap-type*

Description The `get-scrap` function returns two values. The first value is the current scrap of *scrap-type*. The second value is `t` if some scrap is found or `nil` if no scrap is found.

The `get-scrap` function looks up the scrap handler for *scrap-type* and calls `get-internal-scrap` with the handler.

Before calling `get-internal-scrap`, `get-scrap` checks to see whether data needs to be imported from the external Macintosh system scrap.

Argument *scrap-type* A scrap type. In the initial MCL environment, the three predefined scrap types are `:text`, `:fred`, and `:lisp`. The file `pict-scrap.lisp` in your Examples folder adds the `:pict` type.

Example

Here is an example of using `get-scrap` to get some text from the Clipboard. (The string `"Here is some text from the Clipboard"` is already in the Clipboard.)

```
? (get-scrap :text)
"Here is some text from the Clipboard"
T
```

`put-scrap` [*Function*]

Syntax `put-scrap` *scrap-type scrap-value* `&optional` *overwrite-p*

Description The `put-scrap` function stores *scrap-value* in the scrap, as type *scrap-type*. If the value of *overwrite-p* is true (the default), then all other entries (of any type) in the scrap are cleared; if the value of *overwrite-p* is false, scrap entries of other types are not cleared.

The `put-scrap` function works by looking up the scrap handler for *scrap-type* and calling `set-internal-scrap` with the handler and scrap value.

The `put-scrap` function pushes *scrap-type* onto the `*scrap-state*` list and sets the variable `@` to *scrap-value*.

Arguments

scrap-type A scrap type. In the initial MCL environment, the three predefined scrap types are `:text`, `:fred`, and `:lisp`. The file `pict-scrap.lisp` in your Examples folder adds the `:pict` type.

scrap-value The value of the new scrap: that is, what is stored in the scrap. This should be in a format compatible with *scrap-type*.

overwrite-p A Boolean variable indicating whether scrap values of other types should be cleared. The default value is true, which clears all other types from the scrap.

Example

The following code puts the phrase “This is only a text” onto the scrap and retrieves it:

```
? (put-scrap :text "This is only a text")
"This is only a text"
? (get-scrap :text)
"This is only a text"
T
```

`*scrap-state*` [*Variable*]

Description The `*scrap-state*` variable contains a list of scrap types and indicates which types currently have a valid scrap. This variable is modified by calls to `put-scrap`.

`*scrap-handler-alist*` [*Variable*]

Description The `*scrap-handler-alist*` variable contains an association list of scrap-type keywords and scrap-handler objects. Initially, this association list has three entries (one for `:text`, one for `:fred`, and one for `:lisp`). If you define new scrap handlers, you should add entries for them to this list.

Example

This scrap-handler association list contains entries for four scrap handlers. The `:pict` scrap handler is defined in `pict-scrap.lisp` in the MCL Examples folder.

```
? *scrap-handler-alist*
((:PICT . #<PICT-SCRAP-HANDLER #x4BB4F9>) (:LISP . #<LISP-
SCRAP-HANDLER #x302CC9>) (:FRED . #<FRED-SCRAP-HANDLER
#x3029E1>) (:TEXT . #<TEXT-SCRAP-HANDLER #x3025E9>))
```

---

scrap-handler [*Class name*]

Description The class `scrap-handler` is the class of scrap handlers. Methods are provided for the scrap-handler functions `get-internal-scrap`, `set-internal-scrap`, `internalize-scrap`, and `externalize-scrap`.

---

get-internal-scrap [*Generic function*]

Syntax `get-internal-scrap` (*handler* `scrap-handler`)

Description The `get-internal-scrap` generic function returns the value of the scrap of a given type. This function is called by `get-scrap`.

Argument *handler* A scrap handler.

---

set-internal-scrap [*Generic function*]

Syntax `set-internal-scrap` (*handler* `scrap-handler`) *value*

Description The `set-internal-scrap` generic function sets the value of the scrap of a given type. This function is called by `put-scrap`.

Arguments *handler* A scrap handler.
value The new value.

---

internalize-scrap [*Generic function*]

Syntax `internalize-scrap` (*handler* `scrap-handler`)
`internalize-scrap` (*handler* `text-scrap-handler`)

Description The `internalize-scrap` generic function converts the scrap from external to internal format. This function is called when the user switches into Macintosh Common Lisp from another application or from a desk accessory. The function retrieves data from the Macintosh system heap using the appropriate system calls and then calls `set-internal-scrap` on the result.

The operation of the Macintosh system heap and the appropriate system calls are described in *Inside Macintosh.*

Argument *handler* A scrap handler.

`externalize-scrap` [*Generic function*]

Syntax `externalize-scrap` (*handler* `scrap-handler`)
`externalize-scrap` (*handler* `text-scrap-handler`)

Description The `externalize-scrap` generic function converts the scrap from internal to external format. This function is called when the user switches from Macintosh Common Lisp to another application or to a desk accessory. The function copies data to the Macintosh system heap using the appropriate system calls.

The default method for `scrap-handler` does nothing.

The operation of the Macintosh system heap and the appropriate system calls are described in *Inside Macintosh.*

Argument *handler* A scrap handler.

The Read-Eval-Print Loop

Associated with each Listener is a read-eval-print loop run by the `toplevel-loop` function. This function takes input from the Listener (and other buffers), evaluates it, prints out the result, and then gets another input.

`toplevel-loop` [*Function*]

Syntax `toplevel-loop`

Description The `toplevel-loop` function implements the read loop.

Eval-Enqueue

An event (such as choosing a command or clicking a dialog box) that begins a long process should not simply execute the process. If it does, the process runs with interrupts disabled and future events are ignored until the process returns. This is fine for quick actions but can be a problem for time-consuming actions.

The solution to this problem is for event actions to spawn separate processes or queue up forms. In the latter case, the forms are received and processed in order by the topmost listener. This keeps interrupts enabled.

There are many ways to queue up forms. The simplest is to push them onto a list and have the topmost listener's read-eval-print loop pop things from the list. This can be done automatically with the function `eval-enqueue` which is designed to work with the built-in read-eval-print loop function, `toplevel-loop`.

`eval-enqueue` [*Function*]

Syntax `eval-enqueue` *form*

Description The `eval-enqueue` function queues up *form* for evaluation in the read-eval-print loop. The `eval-enqueue` function returns immediately. This means that *form* is not executed at the event-handling level but instead is executed as if it had been entered into the Listener. (It is executed only when other forms entered into the Listener or queued up have returned.)

This function is useful for initiating programs from within event handlers. The form is executed as part of the normal read-eval-print loop rather than as part of an event handler. This means that other events can be processed during the execution of *form*.

Note that `eval-enqueue` is a function, and so its argument is evaluated. The result of this evaluation is put into the read-eval-print loop.

Argument *form* An MCL form.

Examples

Here is an example of how to use `eval-enqueue` to evaluate a form with event processing enabled. The first menu item does not disable event handling; the second menu item does. (Note that both can be aborted by typing Command-period.)

```
? (setq my-menu (make-instance 'menu :menu-title "Events"))
#<MENU "Events">
```

```
? (menu-install my-menu)
T
? (setq can-process-this-item
        (make-instance 'menu-item
                       :menu-item-title "Process events"
                       :menu-item-action
                       #'(lambda (item)
                           (declare (ignore item))
                           (eval-enqueue
                            '(dotimes (x 100)
                               (print "Choose menus"))))))
#<MENU-ITEM "Process events">
? (setq cant-process-this-item
        (make-instance 'menu-item
                       :menu-item-title
                        "Don't process events"
                       :menu-item-action
                       #'(lambda (item)
                           (declare (ignore item))
                           (dotimes (x 100)
                            (print "Choose if you can"))))))
#<MENU-ITEM "Don't process events">
? (add-menu-items my-menu
                  can-process-this-item
                  cant-process-this-item)
```

The user can also use `eval-enqueue` in dialog boxes. The action is initiated by the dialog box, but the user can still access other parts of the system (including other dialog buttons) while the action is running.

In the following example, the action of the Go button is queued so that other events can be processed while it is running. This allows the user to click the Stop button.

Note that the action of the Stop button does not call `eval-enqueue`. If it did, the queued form would never be run (because the form queued by the Go button would never return). The Stop button communicates with the action of the Go button by changing the value of a lexical variable.

```
? (let ((stop nil))
       (flet ((start ()
                (setq stop nil)
                (loop
                  (if stop (return))
                  (print "Click stop when bored"))))
         (make-instance 'window
```

```
:window-title "Stop and Go"
:view-subviews
(list
 (make-instance 'button-dialog-item
   :dialog-item-text "Go"
   :dialog-item-action
   #'(lambda (item)
       (declare (ignore item))
       (eval-enqueue `(funcall ,#'start))))
 (make-instance 'button-dialog-item
   :dialog-item-text "Stop"
   :dialog-item-action
   #'(lambda (item)
       (declare (ignore item))
       (setq stop t)))))))
```

---

get-next-queued-form *[Function]*

Syntax `get-next-queued-form`

Description The `get-next-queued-form` function returns the next form from the pending queue or returns `nil` if there are no forms pending. A second value returned is `t` if there was a pending form and `nil` if there was no pending form.

During programming sessions, queued-up forms include text entered in the Listener and evaluated from buffers as well as forms passed to `eval-enqueue`.

Chapter 11:

Apple Events

Contents

This chapter describes how Macintosh Common Lisp supports Apple events and the Apple Event Manager.

You should read this chapter if you want to understand how Macintosh Common Lisp supports required Apple events and how you can support Apple events in your application. It also describes how to communicate between Macintosh Common Lisp and another process, such as ToolServer or HyperCard.

Before reading this chapter, you should be familiar with the Macintosh Event Manager and with MCL event handling. You should also read about the Apple Event Manager in *Inside Macintosh*. When communicating with another program, you should read the other program's Apple events documentation as well; for example, if you are communicating between Macintosh Common Lisp and HyperCard, you should look at the "AppleEvent Primer" stack in the folder "Your Tour of HyperCard," distributed with version 2.1 of HyperCard.

Implementation of Apple events

The Finder uses Apple events to open applications and quit them, to open documents, and to print documents. In addition, Apple events and the Apple Event Manager may provide services to other applications and request services from them. The Apple Event Manager is available only under System 7. To determine whether the Apple Event Manager is available, call the Gestalt function described in the compatibility guidelines information in *Inside Macintosh*.

Macintosh Common Lisp provides built-in support for receiving Apple events and replying to them. It supports the four required Apple events: Open Application, Open Documents, Print Documents, and Quit Application. It also provides facilities for defining other Apple event handlers.

Creating Apple events and sending them to other applications are not directly supported in Macintosh Common Lisp. However, in your Examples folder are three files illustrating how to send Apple events in Macintosh Common Lisp. These files are

- `appleevent-toolkit.lisp`, containing useful functions for sending Apple events to other processes, including to HyperCard
- `eval-server.lisp`, which shows how to handle standard `doscript` and `eval` Apple events
- `toolserver.lisp`, an example of an Apple events interface to ToolServer

Applications and Apple Events

Macintosh Common Lisp defines a class, `application`, on which Apple event handlers are specialized. Macintosh Common Lisp defines Apple event handlers as generic functions, specialized on the `application` class. In addition, MCL provides a number of other application-based generic functions which are not directly related to Apple events.

Apple event handlers work exactly like other MCL event handlers. For example, the MCL event handler `window-null-event-handler` is specialized on `window`. To customize this behavior, you can create your own subclasses of `window` and write methods on `window-null-event-handler` for those classes. In just the same way, for Apple event handlers, you can create your own subclass of `application` and add your own specialized methods on the basic handlers. You can also write handlers for new Apple events.

In Macintosh Common Lisp, only one instance of the class `application` is used at a time. The instance represents the current application object, bound to the variable `*application*`. Apple event handler methods are called on the value of `*application*`.

Because Macintosh Common Lisp bypasses the Apple Event Manager's dispatch routine and does its own dispatching, you do not need to concern yourself with how your function is called by the Apple Event Manager. Lisp takes care of both dispatching and run-time error checking. If no error occurs, the handler should simply return in the normal way. The value returned by the handler is ignored.

If an error occurs, the handler should signal an `appleevent-error` condition, with an error number and an optional error string.

An Apple event handler method has four arguments:

application The application, always the value of `*application*`.

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

For an extended example of how to write Apple event handlers, see the file `eval-server.lisp` in the MCL Examples folder.

Application class and built-in methods

This section describes the application class, its built-in subclasses, and the methods and generic functions defined on them.

`application` [*Class name*]

Description The `application` class is the class on which Apple event handlers are specialized.

`lisp-development-system` [*Class name*]

Description The `lisp-development-system` class is a subclass of the `application` class. When MCL starts up, the value of `*application*` is an instance of `lisp-development-system`.

`*application*` [*Variable*]

Description The `*application*` variable is bound to an instance of a subclass of `application`. Its initial value is an instance of the `lisp-development-system` class.

`application-error` [*Generic function*]

Syntax `application-error (`*application* `application)` *condition error-pointer*
`application-error (`*application* `lisp-development-system)`*condition error-pointer*

Description The generic function `application-error` is called whenever a condition is signaled that has no handler. The method for `application` quits the application. The method for `lisp-development-system` enters a break-loop.

You can customize your error handling by defining a subclass of `application` and setting `*application*` to an instance of your class. User `application-error` methods should have a non-local exit, because if `application-error` returns, MCL calls it again with a condition so that it may not return. However, if it returns from that call, MCL throws to the toplevel.

Arguments

application — The application. MCL standard event handling always uses the value of `*application*`.

condition — The error condition.

error-pointer — An integer representing the address of the stack frame of the function that signaled the error. The method specialized on lisp-development-system uses this address to determine the name of the function and uses this address as an input to the stack backtrace facility.

Example

```
? (defclass my-application (application)())
#<STANDARD-CLASS MY-APPLICATION>
? (defmethod application-error ((application my-application)
                                 condition error-pointer)
    (declare (ignore error-pointer))
    (message-dialog (format nil "error: ~a" condition))
    (toplevel))
#<STANDARD-METHOD APPLICATION-ERROR (MY-APPLICATION T T)>
? (setf *application* (make-instance 'my-application))
#<MY-APPLICATION #xB09279>
```

---

application-overwrite-dialog [*Generic function*]

Syntax `application-overwrite-dialog` (*application* `application`) *filename prompt*

Description The generic function `application-overwrite-dialog` displays a dialog when there is an attempt to overwrite an existing file. The dialog asks whether to replace the file or choose a new filename.

Arguments

application — The application. MCL standard event handling always uses the value of `*application*`.

filename — A pathname or string that specifies an existing file.

prompt — The prompt message.

`find-edit-menu` [*Generic function*]

Syntax `find-edit-menu (`*application* `application)`

Description The generic function `find-edit-menu` returns the first menu in the menu bar containing the command -X.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`toplevel-function` [*Generic function*]

Syntax `toplevel-function (`*application* `application)` *init-file*
`toplevel-function (`*application* `lisp-development-system)` *init-file*

Description The generic function `toplevel-function` is called when an application created by `save-application` starts. The method for `application` calls `open-application-document` or `print-application-document` on each document specified in the Finder for opening or printing. The method for lisp-development-system loads files of type `fasl` that were selected in the Finder, loads *init-file*, calls `load-preferences-file`, and opens Fred windows for any text files that were selected in the Finder, and creates a Listener window.

Arguments *application* The application. MCL standard event handling always uses the value of *application*.

init-file A pathname or string that is the *init-file* argument to the save-application function.

`view-key-event-handler` [*Generic function*]

Syntax `view-key-event-handler (`*application* `application)` *char*

Description The generic function `view-key-event-handler` is called with *application* as the first argument when there are no active windows and the user presses a key on the keyboard. The method for application sounds a beeps.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

char The current keystroke character.

New application methods

The following methods may be defined for your `application` subclass to return the values noted. Some of the values they return are used as the default values in the Save Application dialog fields when a particular application class is selected.

`application-name` [*Generic function*]

Syntax `application-name (`*application* `application)`

Description Return the name of the application (a string). The default value is `nil`.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-file-creator` [*Generic function*]

Syntax `application-file-creator (`*application* `application)`

Description Returns a four-character string or symbol for Finder file creator type. The default value is `:|????|` (the value of the constant `default-app-creator`).

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-about-view` [*Generic function*]

Syntax `application-about-view (`*application* `application)`

Description A `view` instance containing dialog items to display in the About dialog; the mandatory MCL redistribution notice is placed below this view to make the About dialog. The default value is a static text item with the application's name.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-about-dialog` [*Generic function*]

Syntax `application-about-dialog (`*application* `application)`

Description (Optional) A `dialog` instance to be used as the About dialog.

Note: the dialog may need to contain an MCL redistribution notice to satisfy licensing agreements; it is recommended to define a view instead (see above). The default value is a dialog constructed using the value of `application-about-view.`

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-sizes` [*Generic function*]

Syntax `application-sizes (`*application* `application)`

Description Returns two values, the minimum Finder memory partition size, and the preferred Finder memory partition size, each in kilobytes. If a value is specified as `nil`, the number from the associated Save Application dialog item will be used; these are initially taken from 'size' resource of the MCL application itself.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-resource-file` [*Generic function*]

Syntax `application-resource-file (`*application* `application)`

Description Either `nil` or the name of a file whose resources will be copied in to the new application, adding to or replacing those copied from the MCL application itself (all resources are copied from MCL, except the '`CCL2`' resource). See the Inside Macintosh documentation on Finder Resources to define the proper resources for your application. The default value is `nil`.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-suspend-event-handler` [*Generic function*]

Syntax `application-suspend-event-handler (`*application* `application)`

Description This function is called with the value of `*application*` when MCL is suspended. The `application` method converts the scrap, deactivates windows, and hides windoids if `*hide-windoids-on-suspend*` is true.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-resume-event-handler` [*Generic function*]

Syntax `application-resume-event-handler (`*application* `application)`

Description This function is called with the value of `*application*` when MCL is resumed. The `application` method converts the scrap, reactivates the front window, and shows hidden windoids if `*hide-windoids-on-suspend*` is true.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

`application-eval-enqueue` [*Generic function*]

Syntax `application-eval-enqueue (`*application* `application)` *form*
`application-eval-enqueue (`*application* `lisp-development-system)` *form*

Description This function is called with the value of `*application*` by the `eval-enqueue` function. The `application` method calls `funcall` (for a function or symbol) or `eval` (for a list) on *form*. The `lisp-development-system` method adds form to the eval queue of the frontmost active listener if one exists, otherwise invokes `call-next-method`.

Arguments *application* The application. MCL standard event handling always uses the value of `*application*`.

form A symbol, function or lisp form.

Example:

```
(defclass my-application (application)())

(defmethod application-name ((app my-application))
  "AwesomeApp")

(defmethod application-file-creator ((app my-application))
  :|MyAp|)

(defmethod application-resource-file ((app my-application))
  "ccl:myapp;app-resources.rsrc")

(defmethod application-about-view ((app my-application))
  (make-dialog-item 'static-text-dialog-item
                    #@(10 10)
                    #@(300 30)
                    "AwesomeApp™
The most awesome application ever!"))

(defmethod application-sizes ((app my-application))
  (values 5000 7000))
```

Standard Apple event handlers

This section describes the generic functions defined to handle the four basic Apple events, their predefined methods, and auxiliary functions called by them.

open-application-handler [*Generic function*]

Syntax `open-application-handler(`*application* `application)` *appleevent reply refcon*

Description The generic function `open-application-handler` handles the Open Application Apple event. The default method does nothing.

Arguments *application* The application, always the value of `*application*`.

appleevent — The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply — Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon — The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

`quit-application-handler` [*Generic function*]

Syntax `quit-application-handler(`*application* `application)`*appleevent reply refcon*

Description The generic function `quit-application-handler` handles the Quit Application Apple event. The default method quits Macintosh Common Lisp.

Arguments

application — The application, always the value of `*application*`.

appleevent — The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply — Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon — The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

`open-documents-handler` [*Generic function*]

Syntax `open-documents-handler (`*application* `application)` *appleevent reply refcon*

Description The generic function `open-documents-handler` handles the Open Documents Apple event. The method for `application` calls `open-application-document` on *application* for each document.

Arguments

application — The application. MCL standard event handling always uses the value of `*application*`.

appleevent — The Apple event, which is a `macptr` of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply — Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon — The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

`open-application-document` [*Generic function*]

Syntax `open-application-document (`*application* `lisp-development-system)` *filename startup*
`open-application-document (`*application* `application)` *filename startup*

Description The generic function `open-application-document` is called by the `open-documents-handler` method. The method for `lisp-development-system` loads files of type `:fasl` and opens files of type `:text` for editing; the method for `application` does nothing. You can customize an Apple event by defining a subclass of `application` and setting `*application*` to an instance of your class.

Arguments

application — The application. MCL standard event handling always uses the value of `*application*`.

filename — The file to load and open for editing.

startup — A boolean value that indicates whether the event is occurring on startup. If the value is true, this function was called upon startup.

`print-documents-handler` [*Generic function*]

Syntax `print-documents-handler(`*application* `application)` *appleevent reply refcon*

Description The generic function `print-documents-handler` handles the Print Documents Apple event. The method for `application` calls `print-application-document` on *application* for each document.

Arguments

application The application. MCL standard event handling always uses the value of `*application*`.

appleevent The Apple event, which is a `macptr` of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

`print-application-document` [*Generic function*]

Syntax `print-application-document (`*application* `lisp-development-system)`*filename startup*

`print-application-document (`*application* `application)` *filename startup*

Description The generic function `print-application-document` is called by the `print-document-handler` method for `application`. The `print-application-document` method for `lisp-development-system` prints *filename;* the `print-application-document` method for `application` does nothing. You can customize an Apple event by defining a subclass of `application` and setting `*application*` to an instance of your class.

Arguments

application The application. MCL standard event handling always uses the value of `*application*`.

filename The file that the function prints.

startup A boolean value that indicates whether the event is occurring on startup. If the value is true, this function was called upon startup.

Defining new Apple events

This section describes functions and macros used in defining new Apple events. If you want your application to understand additional Apple events, use the features described here.

appleevent-error [*Condition type*]

Syntax `make-condition 'appleevent-error &rest` *initargs*

Description If an Apple event handler finds an error, it should signal this condition. Any MCL errors that occur while handling the Apple event are automatically caught by Macintosh Common Lisp and handled appropriately.

A condition of the `appleevent-error` condition type is created with the Common Lisp function `make-condition` (see *Common Lisp: The Language*, page 901).

Arguments *initargs* A list of keywords and values used to initialize the `appleevent-error`.

`:oserr` An error number.

`:error-string`
A string of human-readable text that identifies the error to a user.

Example

Here is an example of using `appleevent-error.`

```
(error (make-condition 'appleevent-error
                             :oserr #$AEEventNotHandled
                       :error-string "Didn't understand event"))
```

`ae-error` [*Macro*]

`ae-error-str`

Syntax

`ae-error` `&body` `(`*form*`)+`
`ae-error-str` *error-string* `&body` `(`*form*`)+`

Description

These macros simplify calls to the Apple Event Manager by signaling the `appleevent-error` condition if the error code returned by the Apple Event Manager is not `0` (`NoErr`). The value of the call should be the value of the body of the macro.

The `ae-error-str` macro lets you specify an error string; the `ae-error` macro does not.

All calls to the Apple Event Manager should be wrapped in a call to either the `ae-error` macro or the `ae-error-str` macro.

Arguments

form	One or more forms to be executed within the body of the macro.
error-string	A human-readable error string.

`with-aedescs` [*Macro*]

Syntax

`with-aedescs` (*vars*) `&body` *body*

Description

The `with-aedescs` macro creates a temporary record of type `AEDesc` for the extent of the macro. It is similar to the macro `rlet`. It wraps *body* within an `unwind-protect`, so that no matter how *body* is exited, `with-aedescs` disposes of all its temporary records in the correct way. If the data handle has anything in it, `with-aedescs` calls `#_AEDisposeDesc`. Thus any memory allocated by the Apple Event Manager is properly disposed of.

If you have a need for an `AEDesc` record with indefinite extent, you must use `make-record`. When you want to dispose of the record, you must explicitly call `#_AEDisposeDesc`, then `dispose-record`.

Arguments

vars	One or more variables.
body	One or more forms to be executed within the body of the macro.

Example

There are examples of using `with-aedescs` in `eval-server.lisp` and `tool-server.lisp` in your Examples folder.

`check-required-params` [*Function*]

Syntax `check-required-params` *error-string appleevent*

Description The `check-required-params` function uses the Apple Event Manager to check whether all required parameters of the Apple event *appleevent* have been extracted. If a parameter has been missed, the `appleevent-error` condition is signaled with `:oserr #$AEParamMissed` and `:error-string` *error-string*.

Arguments

error-string	A human-readable error string.
appleevent	An Apple event.

`appleevent-idle` [*Pascal function*]

Syntax `appleevent-idle`

Description The `appleevent-idle` Pascal function should be specified whenever the Apple Event Manager asks for a function to call while it is waiting (for example, in calls to `#_AEInteractWithUser`). It should never be called directly, only passed.

`%path-from-fsspec` [*Function*]

Syntax `%path-from-fsspec` *fsspecptr*

Description The `%path-from-fsspec` function extracts a Lisp pathname from *fsspecptr*.

Argument

fsspecptr	An MCL macptr that points to an object of Macintosh type `FSSpec`. For more information on macptrs, see "Macptrs" on page 531

Installing Apple event handlers

The following functions install and deinstall Apple event handlers.

`install-appleevent-handler` [*Function*]

Syntax `install-appleevent-handler` *class id function* `&optional` *refcon*

Description The function `install-appleevent-handler` installs an Apple event handler.

Arguments

class	A four-letter keyword denoting the class of the event, for example, `:|aevt|`.
id	A four-letter keyword denoting the ID of the event, for example, `:|odoc|`.
function	A function or a symbol with a function binding.
refcon	An optional reference identifier, which can be any MCL object; it identifies the specific installation of a handler.

`deinstall-appleevent-handler` [*Function*]

Syntax `deinstall-appleevent-handler` *class id*

Description The `deinstall-appleevent-handler` function deinstalls an Apple event handler.

Arguments

class	A four-letter keyword denoting the class of the event, for example, `:|aevt|`.
id	A four-letter keyword denoting the ID of the event, for example, `:|odoc|`.

Installing handlers for queued Apple event replies

Some Apple events received in the event queue, specifically Apple events sent with the `#$kAEQueueReply` mode, are replies to previously sent Apple events. Their handlers are installed differently from other Apple events; the handler is installed when the originating Apple event is sent, and is automatically deinstalled after one use.

The following functions handle queued Apple event replies correctly.

`install-queued-reply-handler` [*Function*]

Syntax `install-queued-reply-handler` *appleevent-or-id function* `&optional` *refcon*

Description The `install-queued-reply-handler` function installs a handler for a queued reply.

Arguments

appleevent-or-id
Either a return ID number or the originating Apple event from which a return ID number can be extracted.

function A function to be called when the reply comes back. This function should be a normal Apple event handler as described in "Defining new Apple events" on page 404.

refcon An optional reference identifier, which can be any MCL object; it identifies the specific installation of a handler.

`queued-reply-handler` [*Generic function*]

Syntax `queued-reply-handler(`*application* `application)` *appleevent reply refcon*

Description The generic function `queued-reply-handler` calls the installed reply handler for the return ID of *appleevent.* If there is no applicable reply handler, it calls `no-queued-reply-handler`.

Arguments

application The application, always the value of `*application*`.

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

`no-queued-reply-handler` [*Generic function*]

Syntax `no-queued-reply-handler(`*application* `application)`*appleevent reply refcon*

Description The default method of the generic function `no-queued-reply-handler` signals the `appleevent-error` condition with `:oserr #$AEEventNotHandled`.

Arguments

application The application, always the value of `*application*`.

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

Sending Apple events

There are no built-in functions to send Apple events within Macintosh Common Lisp. However, it is easy to write your own functions for sending Apple events.

To send an Apple event from Macintosh Common Lisp, do the following four steps.

1. Create a target.
2. Create the event.
3. Put data into the Apple event.
4. Send the Apple event.

In the file `appleevent-toolkit.lisp` in your Examples folder, you will find a selection of functions that help you do these steps. For example, you can create Apple events with the MCL function `create-appleevent`.

Since these functions are not part of standard Macintosh Common Lisp, load the file `appleevent-toolkit.lisp` to use them.

The file `eval-server.lisp`, also in the Examples folder, handles `eval`, `dosc`, and `scpt` Apple events. This file also contains code to communicate with HyperCard and MPW.

Chapter 12:

Processes

Contents

Macintosh Common Lisp supports multiple processes. This chapter discusses creating processes, obtaining information about processes, and scheduling processes. This chapter also describes two locking mechanisms for synchronizing processes.

Processes in Macintosh Common Lisp

Processes facilitate concurrent execution of computations. They share a single address space and communicate via shared Lisp objects. A scheduler controls the state of the processes. It chooses which process to run based on several conditions.

A process is either *active* or *stopped*. An active process is either running or waiting to be scheduled. A stopped process is a process that is halted. A process is scheduled only if it is waiting. Therefore, a stopped process must be made active before it is scheduled.

Associated with a process are two sets of objects known as *run reasons* and *arrest reasons*. Run reasons and arrest reasons are lists. A process is active when it has at least one run reason and has no arrest reasons. Run or arrest reasons are added to the lists with `pushnew`.

Each process you create has an initial function and a list of arguments. The function is applied to the list of arguments when the process is first run or when the process is reset.

When an MCL application starts up, it creates two processes. The first, known as Initial, is responsible for processing events. The second, known as Listener, runs the `read-eval-print` loop using a Lisp Listener. If an error occurs in the event processor, a new event processing process is created (Standin) and a break occurs in the initial process using a new listener. Errors that occur in the stand-in event process are ignored. So if you get an error during event processing, it is recommended that you figure out what is wrong and get out of the break. You exit the break by typing `command-.` or `command-/` in the listener for the break loop. Failure to exit the break can lead to unexpected behavior, for example `meta-.` not working (because, for instance, there is not enough room to create a new window, but the error that would tell you so is ignored).

If there are only two processes running, then `Command-.` and `Command-,` apply to the main process (the one that is not processing events). To break or abort in the event process, use `Option-Command`. If there are more than two processes running, a dialog appears allowing you to choose a process to abort. If all processes are idle, `Command-.` and `Option-Command-.` both abort the event processor. If one process is running and one is idle, `Command-.` aborts the busy process, and `Option-Command-.` brings up the dialog.

If you have an “init.lisp” (or “init.fasl”) file that prints anything to `*standard-output*`, it will print in a Fred window titled "Initialization Output".

The class front-listener-terminal-io is a subclass of terminal-io. It has a method for stream-current-listener that returns *top-listener* (the top listener for *current-process*) or the frontmost listener if one exists, otherwise a new listener.

The default method for stream-current-listener returns either *top-listener* if not nil or sets *top-listener* to a newly created listener and returns the new listener.

All the stream methods such as stream-tyi for terminal-io ultimately call stream-current-listener. The initial value for both *trace-output* and *standard-output* is an instance of front-listener-terminal-io.

Process priorities

All processes are assigned a priority. It is recommend that you do not assign a priority greater than 0, which is the default priority. System processes, such as the event handler, run at priority 1.

Within a priority level, the scheduler runs all processes in a round-robin fashion. Processes having a higher priority level run to the exclusion of processes having lower priorities. Regardless of priority, a process will not run longer than a time interval specified when it was created (i.e., its quantum).

The Processes Inspector window shows the priorities of all processes.

Creating processes

Two functions are available for creating processes. These functions are named process-run-function and make-process. The function process-run-function creates a process and executes it asynchronously immediately after the function is called. The function make-process creates a process that is activated at a later time.

The function `process-run-function` calls `make-process`, `process-preset`, and `process-enable`. If you create a process using the function `make-process`, you must call `process-preset` to set the initial function and initial arguments, and call `process-enable` to make the process active.

`process-run-function` [*Function*]

Syntax `process-run-function` *name-or-kwds function* `&rest` *args*

Description The `process-run-function` function creates a process, presets the process to apply *function* to *args*, and starts the process.

Arguments

name-or-kwds A string identifying the process or a list of alternating keyword names and values. The keywords are:

`:name` A string that is the name of the process. The string "Anonymous" is the default.

`:restart-after-reset`
A predicate. If the value is true, the process restarts after a reset. If the value is `nil`, the process waits for scheduling.

`:priority`
The priority of the process. The default is 0.

`:quantum`
A time interval, in ticks, during which the process can run before the scheduler runs a different process. The default is 6 (i.e., 0.1 seconds).

`:stack-size`
The initial stack size in bytes. The default is 16384.

`:background-p`
A value indicating whether the variable `*idle*` is `nil` when the process is scheduled. If `:background-p` is `nil`, the variable `*idle*` is set to `nil` each time the process is scheduled. The default is `nil`.

function A function that the process executes.

args The arguments passed to *function*.

`make-process` [*Function*]

Syntax `make-process` *name* `&key` *kwds*

Description The `make-process` function creates a process named *name*.

Arguments	*name*	The name of the new process.
	kwds	An alternating list of keywords and values. These specify initial options for the process. Valid keywords are:
	`:stack-group`	The stack group used by the process. If not specified, an appropriate stack group is created.
	`:priority`	The priority of the process. The default is 0.
	`:quantum`	A time interval, in ticks, during which the process can run before the scheduler runs a different process. The default is 6 (i.e., 0.1 seconds).
	`:run-reasons`	A list of reasons indicating that the process is active.
	`:arrest-reasons`	A list of reasons indicating that the process is inactive.
	`:stack-size`	The initial stack size in bytes. The default is 16384.
	`:background-p`	A value indicating whether the variable `*idle*` is `nil` when the process is scheduled. If `:background-p` is `nil`, the variable `*idle*` is set to `nil` each time the process is scheduled. The default is `nil`.

Process attribute functions

The following MCL functions return the attributes of a process.

`process-name` [*Function*]

Syntax `process-name` *process*

Description The `process-name` function returns the name of the process, as specified when the process was created by the `make-process` function or the `process-run-function` function.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

process-stack-group [*Function*]

Syntax `process-stack-group` *process*

Description The `process-stack-group` function returns the stack group that is executing on behalf of the process.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

process-initial-stack-group [*Function*]

Syntax `process-initial-stack-group` *process*

Description The `process-initial-stack-group` function returns the stack group created when process was created. The size of the stack group is determined by the variable `*default-process-stackseg-size*`.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

default-process-stackseg-size [*Variable*]

Description Controls the initial size of the stack group when a process is created. The default is 16K.

process-initial-form [*Function*]

Syntax `process-initial-form` *process*

Description The `process-initial-form` function returns a list whose `car` is the initial function of the process and whose `cdr` is the list of arguments for that function.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

process-wait-function [*Function*]

Syntax `process-wait-function` *process*

Description The `process-wait-function` function returns the function passed to `process-wait`.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

process-wait-argument-list [*Function*]

Syntax `process-wait-argument-list` *process*

Description The `process-wait-argument-list` function returns the arguments to the function passed to `process-wait`.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

process-priority [*Function*]

Syntax `process-priority` *process*

Description The `process-priority` function returns the priority of *process*. To change the priority, you can use setf.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

default-quantum [*Variable*]

Syntax `*default-quantum*`

Description The `*default-quantum*` variable contains the time interval during which the process runs before another process can be scheduled. To change the quantum, you can use `setf.`

`process-quantum-remaining` [*Function*]

Syntax `process-quantum-remaining` *process*

Description The `process-quantum-remaining` function returns the time remaining before rescheduling occurs. The time returned is in sixtieths of a second (i.e., ticks).

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

Run and arrest reason functions

Each process you create has a list of run reasons and a list of arrest reasons. Before a process becomes active, it must have at least one run reason and no arrest reasons.

The following functions specify run reasons and arrest reasons.

`process-run-reasons` [*Function*]

Syntax `process-run-reasons` *process*

Description The `process-run-reasons` function returns the list of run reasons for *process*.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

`process-arrest-reasons` [*Function*]

Syntax `process-arrest-reasons` *process*

Description The `process-arrest-reasons` function returns the list of arrest reasons for *process*.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

`process-enable` [*Function*]

Syntax `process-enable` *process*

Description The `process-enable` function activates the process specified by *process*. All run and arrest reasons are removed from their respective lists and a run reason of `:enable` is given to *process*.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

`process-disable` [*Function*]

Syntax `process-disable` *process*

Description The `process-disable` function stops the process specified by *process*. All run and arrest reasons are removed from their respective lists.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

`process-enable-run-reason` [*Function*]

Syntax `process-enable-run-reason` *process* &optional (*reason* `:user`)

Description The `process-enable-run-reason` function adds *reason* to a list of run reasons for a process. This can activate the process.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

reason A value to add to the run reasons list. The value of *reason* is compared using the `eq` function.

`process-disable-run-reason` [*Function*]

Syntax `process-disable-run-reason` *process* &optional (*reason* `:user`)

Description The `process-disable-run-reason` function removes *reason* from a process's run reasons. This can deactivate the process.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

reason The value to remove from the run reasons list. The value of *reason* is compared using the `eq` function.

`process-enable-arrest-reason` [*Function*]

Syntax `process-enable-arrest-reason` *process* &optional (*reason* `:user`)

Description The `process-enable-arrest-reason` function adds *reason* to a process's arrest reasons. This can deactivate the process.

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

reason A value to add to the run reasons list. The value of *reason* is compared using the `eq` function.

`process-disable-arrest-reason` [*Function*]

Syntax `process-disable-arrest-reason` *process* &optional (*reason* `:user`)

Description The `process-disable-arrest-reason` function deletes *reason* from a process's arrest reasons. This can activate the process.

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

reason The value to remove from the run reasons list. The value of *reason* is compared using the `eq` function.

`process-active-p` [*Function*]

Syntax `process-active-p` *process*

Description The `process-active-p` function returns t if *process* is active (i.e., the process can run if its wait function allows).

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

`process-whostate` [*Function*]

Syntax `process-whostate`

Description Returns the description of the process.

`process-warm-boot-action` [*Function*]

Syntax `process-warm-boot-action` *process*

Description Returns the process's warm-boot action. This value is not currently used by MCL.

`process-simple-p` [*Function*]

Syntax `process-simple-p` *process*

Description Returns `t` for a simple process `nil` for a normal process.

`process-background-p` [*Function*]]

Syntax `process-background-p` *process*

Description Returns `t` if the process is running in the background.

`process-last-run-time` [*Function*]

Syntax `process-last-run-time` *process*

Description Returns the last time the process ran, as a universal time.

`process-total-run-time` [*Function*]

Syntax `process-total-run-time` *process*

Description Returns the amount of time the process has received, in ticks (sixtieths of a second).

`process-creation-time` [*Function*]

Syntax `process-creation-time` *process*

Description Returns the time the process was created, as a universal time.

`clear-process-run-time` [*Function*]

Syntax `clear-process-run-time` *process*

Description Resets the previous total run time to zero. This is useful for testing.

Starting and stopping processes

The following MCL functions start and stop processes.

`process-preset` [*Function*]

Syntax `process-preset` *process function* &rest *args*

Description The `process-preset` function sets the initial function of *process* to *function* and the initial arguments to *args*. The process is reset so that it throws out of its present computation and starts up by applying *function* to *args*.

If this function is called for a stopped process, the process is not activated.

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

function A function that the process executes.

args The arguments passed to *function*.

`process-reset` [*Function*]

Syntax `process-reset` *process* `&optional` *unwind-option kill without-aborts*

Description The `process-reset` function causes a process to throw to its top level and apply its initial function to its initial arguments when it runs again.

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

unwind-option The possible values for this argument are:

`:unless-current` or `nil`
: Unwinds unless the stack group is the one that is currently executing. This is the default value.

`:always`
: Unwind always. This can cause the function `process-reset` to throw through its caller rather than returning.

`t` Never rewinds.

kill An argument specifying whether to kill a process after unwinding. The possible values for this argument are `:kill` and `nil`. If *kill* is `:kill`, the process is killed after unwinding. The default is `nil`.

without-aborts MCL currently ignores *without-aborts*.

`process-reset-and-enable` [*Function*]

Syntax `process-reset-and-enable` *process*

Description The `process-reset-and-enable` function resets then enables *process*.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

`process-abort` [*Function*]

Syntax `process-abort` &optional *condition*

Description Exits a process by signaling `abort.`

`process-flush` [*Function*]

Syntax `process-flush` *process*

Description The `process-flush` function causes a process to wait indefinitely. The functions `process-preset` or `process-reset` unflush the process.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

`process-kill` [*Function*]

Syntax `process-kill` *process* &optional (*without-aborts* `:ask)`

Description The `process-kill` function destroys the process. The process is reset, stopped, and removed from `*all-processes*`. **Note:** if the process is arrested, it does not kill itself until it is enabled.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

without-aborts MCL currently ignores *without-aborts*.

`process-interrupt` [*Function*]

Syntax `process-interrupt` *process function* `&rest` *args*

Description The `process-interrupt` function interrupts a process by applying *function* to *args*.

Arguments *process* A process, such as one returned by `make-process` or `process-run-function`.

function A function.

args The arguments passed to *function*.

Scheduler

The scheduler controls which process is run. The scheduler runs once a tick and looks at the quantum remaining for the current process. If the quantum has expired, it looks at every active process in a round-robin fashion. If the wait function of a process returns true, the process is scheduled to run.

A process can block by calling `process-block`. A blocked process will not run until it is unblocked by `process-unblock`.

Commonly you want to block, or wait, for an event but to give up after a certain interval has passed. `process-block-with-timeout` and `process-wait-with-timeout` allow the programmer to specify a timeout period (in sixtieths of a second) after which the function should just return.

`*current-process*` [*Variable*]

Syntax `*current-process*`

Description The `*current-process*` variable contains the object of the process that is currently executing.

`without-interrupts` [*Special Form*]

Syntax `without-interrupts &body` *body*

Description The `without-interrupts` special form inhibits scheduling during execution of *body*.

Arguments

body	Zero or more Lisp forms.

`process-wait` [*Function*]

Syntax `process-wait` *whostate function* `&rest` *args*

Description The `process-wait` function causes the current process to wait until the application of *function* to *args* returns true. Bindings when `process-wait` is called are not in effect when *function* executes because *function* is executed in the scheduler's environment.

Arguments

whostate	A string that describes the reason for waiting. This string is displayed in the Processes Inspector window when viewing process states.
function	A function that must be functionp, not a symbol.
args	The arguments applied by *function*.

`process-block` [*Function*]

Syntax `process-block` *process whostate*

Description The `process-block` function causes process *process* to block. The process remains blocked until `process-unblock` is called on it.

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

whostate A string that describes the reason for blocking. This string is displayed in the Processes Inspector window when viewing process states.

`sleep` [*Function*]

Syntax `sleep` *seconds*

Description The `sleep` function causes the current process to wait for a period of time specified by *seconds*.

Arguments

seconds A number that specifies time in seconds.

`process-wait-with-timeout` [*Function*]

Syntax `process-wait-with-timeout` *whostate time function* &rest *args*

Description The `process-wait-with-timeout` function waits for a period of time to elapse or waits until the application of *function* to *args* returns true before returning. If *time* is `nil`, this function behaves the same as `process-wait`.

Arguments

whostate A string that describes the reason for waiting. This string is displayed in the Processes Inspector window when viewing process states.

time The amount of time in intervals of sixtieths of a second (i.e., ticks).

function A function that must be functionp, not a symbol.

args The arguments used by *function*.

`process-block-with-timeout` [*Function*]

Syntax `process-block-with-timeout` *process time whostate*

Description The `process-block-with-timeout` function causes *process* process to stay blocked for a period of time to elapse or until the process is unblocked. If *time* is `nil`, this function behaves the same as `process-block`.

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

time The amount of time in intervals of sixtieths of a second (i.e., ticks).

whostate A string that describes the reason for blocking This string is displayed in the Processes Inspector window when viewing process states.

`process-unblock` [*Function*]

Syntax `process-unblock` *process*

Description The `process-unblock` function unblocks *process*.

Arguments

process A process, such as one returned by `make-process` or `process-run-function`.

`process-allow-schedule` [*Function*]

Syntax `process-allow-schedule`

Description The `process-allow-schedule` function explicitly calls the scheduler to allow other processes to run.

`*active-processes*` [*Variable*]

Syntax `*active-processes*`

Description The `*active-processes*` variable is a list of active processes.

`*all-processes*` [*Variable*]]

Syntax `*all-processes*`

Description The `*all-processes*` variable is a list of all processes that exist.

Locks

Locking mechanisms are an aid to handling process synchronization. In general, they are used for processes that share a resource. Locks prevent a process from executing while another process owns the lock. Once a process obtains the lock, the process prevents other processes that require the lock from executing.

MCL provides two methods for process synchronization. The first method consists of creating and obtaining a lock. The second method creates a queue of processes; the queued processes execute in the order that they were added to the queue.

The following MCL forms control these locking mechanisms.

`make-lock` [*Function*]

Syntax `make-lock`

Description The `make-lock` function creates and returns a lock.

`process-lock` [*Function*]

Syntax `process-lock` *lock* &optional *lock-owner interlock-function*

Description The `process-lock` function obtains a lock. This function waits until the lock is free before returning. If *lock-owner* does not hold the lock, `process-lock` waits until the lock becomes free, then grabs the lock and calls *interlock-function* (atomically). If *lock-owner* holds the lock, *interlock-function* is not called.

Arguments *lock* The lock returned from `make-lock`.

lock-owner The owner of the lock. The default is the current process. The value of *lock-owner* is compared using the `eq` function.

interlock-function A function that is executed after `process-lock` obtains the lock.

`lock-owner` [*Function*]

Syntax `lock-owner` *lock*

Description The `lock-owner` function returns the owner of the lock.

Arguments *lock* A lock returned by `make-lock`.

`with-lock-grabbed` [*Macro*]

Syntax `with-lock-grabbed` (*lock* &optional *lock-owner whostate*) &body *body*

Description The `with-lock-grabbed` macro executes *body* while *lock* is held. After executing *body*, *lock* is released.

Arguments *lock* The lock returned by `make-lock`.

lock-owner The owner of the lock. The default is the current process.

whostate A string that is displayed in the Processes Inspector window.

body Zero or more Lisp forms.

`process-unlock` [*Function*]

Syntax `process-unlock` *lock* `&optional` *lock-owner* (*error-p* t)

Description The `process-unlock` function releases a lock. If the lock was free or locked by a different process and *error-p* is true, an error is signaled. If *error-p* is true, the lock is not released when the lock is held by a different process; if *error-p* is `nil`, the lock is released even when the lock is held by a different process.

Arguments *lock* The lock returned by `make-lock`.

lock-owner	The owner of the lock. The default is the current process. The value of *lock-owner* is compared using the `eq` function.
error-p	A value indicating whether to signal an error condition.

`store-conditional` [*Function*]

Syntax `store-conditional` *lock old new*

Description This MCL 4.0 function checks to see whether the the `lock-value` of *lock* is `eq` to *old*, and, if so, it stores *new* into the cell. The test and the set are done as a single atomic operation. `store-conditional` returns `t` if the test succeeded and `nil` if the test failed. If *lock* is known at compile time to be of type `lock`, the code will be inlined, hence very fast.

This function is not available in MCL 3.1.

Arguments

`lock`	A lock
`old`	Any lisp object
`new`	Any lisp object

If the *lock* argument to a `process-lock` or `process-unlock` form is known at compile time to be of type `lock`, then it will compile into an initial call to `store-conditional` (which will be inlined) followed, only if the `store-conditional` fails, by an out-of-line call to `process-lock` or `process-unlock`.

`make-process-queue` [*Function*]

Syntax `make-process-queue` *name* &optional *size*

Description The `make-process-queue` function creates and returns a queue for processes requesting access to a shared resource. The queue controls the order in which processes are executed on a first-in-first-out basis.

Arguments

name	The identifier for the process queue.
size	The maximum number of processes that can join the queue. The default is an unlimited number.

`process-enqueue` [*Function*]

Syntax `process-enqueue` *queue* `&optional` *queue-owner whostate*

Description The `process-enqueue` function adds a process to a queue. If the queue is empty, the process becomes the owner of the queue, and the function returns. If *queue* is not empty and not full, the function adds the process to the end of *queue*.

The `process-enqueue` function waits until the process reaches the front of the queue and becomes the owner of the queue. If *queue* is full, `process-enqueue` waits until *queue* is not full before adding the process to the end of *queue*.

By default, the process added to the queue is the current process. The optional argument *queue-owner* specifies a different process to place on the queue.

Arguments

queue	The queue returned by `make-process-queue`.
queue-owner	The process. The default is the current process. If the process is already on the queue, an error is signaled. The value of *queue-owner* is compared using the `eq` function.
whostate	A string that describes the reason for waiting. This string is displayed in the Processes Inspector window when viewing process states. "Lock" is the default value.

---

process-enqueue-with-timeout [*Function*]

Syntax `process-enqueue-with-timeout` *queue timeout* `&optional` *queue-owner whostate*

Description The `process-enqueue-with-timeout` function adds a process to a queue. If the queue is empty, the process is placed on *queue*, the process becomes the owner of the queue, and the function returns. If *queue* is not empty and not full, the function adds the process to the end of *queue*.

The process waits until it reaches the front of the queue before it becomes the owner of the queue. If *queue* is full, `process-enqueue` waits until *queue* is not full before adding the process to the end of *queue*.

By default, the process added to the queue is the current process. The optional argument *queue-owner* specifies a different process to place on the queue.

If the process does not reach the front of the queue within the time interval *timeout*, the process is dequeued.

Arguments

queue	The queue returned by `make-process-queue`.
timeout	The amount of time in intervals of sixtieths of a second (i.e., ticks). Additionally, the keyword `:usurp` is a valid value for *timeout*. When *timeout* is `:usurp`, the process is placed unconditionally at the front of the queue.

queue-owner The process. The default is the current process.

whostate A string that describes the reason for waiting. This string is displayed in the Processes Inspector window when viewing process states. "Lock" is the default value.

`with-process-enqueued` [*Macro*]

Syntax `with-process-enqueued` (*queue* `&optional` *queue-owner whostate*
(*signal-dequeue-errors* `t`))
`&body` *body*

Description The `with-process-enqueued` macro executes *body* while *queue* is controlled by *queue-owner*. After executing *body*, *queue-owner* relinquishes control of *queue*.

Arguments

queue The queue returned by `make-process-queue`.

queue-owner The owner of the queue. The default is the current process.

whostate A string that is displayed in the processes Inspector window.

signal-dequeue-errors
A value that determines whether to signal an error condition. When *signal-dequeue-errors* is true, an error is signaled if *queue-owner* does not control *queue* when `process-dequeue` is called on exiting `with-process-enqueue`. The default value is `t.`

body Zero or more Lisp forms.

`process-dequeue` [*Function*]

Syntax `process-dequeue` *queue* `&optional` *queue-owner* (*error-p* `t`)

Description The `process-dequeue` function relinquishes control of a queue so that another process can control the queue. If *queue-owner* controls the queue, `process-dequeue` removes the process from the queue, which gives the next process on the queue an opportunity to run. If *queue-owner* does not control the queue and *error-p* is true, an error is signaled.

Arguments

queue The queue returned by `make-process-queue`.

queue-owner The process to remove from the queue. The default is the current process.

error-p	A value indicating whether to signal an error condition. The default value is t.

`process-queue-locker` [*Function*]

Syntax `process-queue-locker` *queue*

Description The `process-queue-locker` function returns the process that controls queue or returns `nil` if the queue is empty.

Arguments

queue	The queue returned by `make-process-queue`.

`reset-process-queue` [*Function*]

Syntax `reset-process-queue` *queue*

Description The `reset-process-queue` function removes all processes from *queue*. When this function returns, the queue is empty.

Arguments

queue	The queue returned by `make-process-queue`.

Stack groups

Stack groups are used to implement coroutines, generators, and processes. A stack group represents a computation and its internal state, including the control, value, and special binding stacks.

There is always one current stack group. When a stack group becomes the current stack group, that stack group is said to be *resumed*. The former current stack group is known as the *resumer*. The resumer can pass an object to the new current stack group, with the restriction that stack allocated objects should not be passed between stack groups.

The following MCL functions create, initialize, and resume stack groups.

`make-stack-group` [*Function*]

Syntax `make-stack-group` *name* `&optional` *stack-size*

Description The `make-stack-group` function creates and returns a new stack group.

Arguments

name — An object that identifies the stack group.

stack-size — The size of the new stack. The default *stack-size* is 16384 bytes.

`stack-group-preset` [*Function*]

Syntax `stack-group-preset` *stack-group function* `&rest` *args*

Description The `stack-group-preset` function initializes a stack group. When *stack-group* is resumed the stacks are empty and *function* is applied to *args*.

Arguments

stack-group — A stack group returned by `make-stack-group`.

function — The function applied when *stack-group* is resumed.

args — The arguments passed to *function*.

`stack-group-resume` [*Function*]

Syntax `stack-group-resume` *stack-group value*

Description The `stack-group-resume` function resumes *stack-group* and passes the object *value* to *stack-group*. The current stack group becomes *stack-group*'s resumer.

Arguments

stack-group — A stack group returned by `make-stack-group`.

value — An object that the former current stack group passes to the new current stack group.

`stack-group-return` [*Function*]

Syntax `stack-group-return` *value*

Description The `stack-group-return` function resumes the current stack group's resumer. The object *value* is passed to the current stack group's resumer.

Arguments

value — An object that the current stack group passes to its resumer.

`previous-stack-group` [*Function*]

Syntax `previous-stack-group` *stack-group*

Description The `previous-stack-group` function returns the resumer of *stack-group*.

Arguments *stack-group* A stack group returned by `make-stack-group`.

`symbol-value-in-stack-group` [*Function*]

Syntax `symbol-value-in-stack-group` *symbol stack-group*

Description The `symbol-value-in-stack-group` function returns the value of the special variable *symbol* in *stack-group*. To change the value, you can use setf.

Arguments

symbol A special variable. If *symbol* is not bound in *stack-group*, the global value is returned.

stack-group A stack group returned by `make-stack-group`.

`symbol-value-in-process` [*Function*]

Syntax `symbol-value-in-process` *symbol process*

Description The `symbol-value-in-process` function returns the value of the special variable *symbol* in *process*.. To change the value, you can use `setf`.

Arguments

symbol A special variable. If *symbol* is not bound in *stack-group*, the global value is returned.

process A process, such as one returned by `make-process` or `process-run-function`.

Miscellaneous Process Parameters

`*autoclose-inactive-listeners*` [*Variable*]

Description Controls the behavior of inactive Listeners. If set to true, listener windows automatically close when they become inactive. The default value is `nil.`

`*bind-io-control-vars-per-process*` [*Variable*]

Description Controls whether or not I/O control variables are shared or per-process. If set to true, new processes get their own bindings of the I/O control variables(see CLtL2 Table 22-7). The default value is `nil`.

Chapter 13:

Streams

Contents

This chapter discusses the MCL implementation of streams and defines MCL classes and generic functions for dealing with streams.

Implementation of streams

Macintosh Common Lisp implements all of the Common Lisp I/O (input/output) functions by means of streams.

Macintosh Common Lisp implements streams as a simple class of objects, based on the abstract class `stream`. There are only a few things a stream needs to know how to do.

Output streams need to have methods defined for character and string output.

Input streams need to have methods defined to read and "unread" characters (unreading allows "peeking ahead"). Input streams also need to know how to tell when they are at end of file.

You can define your own specialized stream types. The file `serial-streams.lisp` in your Examples folder defines a stream that does input and output through the Macintosh serial port.

MCL expressions relating to streams

The following MCL classes and generic functions deal with streams.

`stream` [*Class name*]

Description The `stream` class is the class from which all streams inherit. It is an abstract class. It should not be directly instantiated but instead should be used to create new subclasses. It has one initialization argument, `:direction`.

`input-stream` [*Class name*]

Description The `input-stream` class is the class of input streams, built on `stream`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance (`*stream* `input-stream) &rest` *initargs*

Description The primary method on `initialize-instance` for `input-stream` initializes an input stream. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.) Input streams have one additional initialization argument.

Arguments

stream	A stream.
initargs	This is the additional initialization argument for input streams.
`:direction`	The direction of the input stream. The default value of `:direction` is `:input.`

`output-stream` [*Class name*]

Description The `output-stream` class is the class of output streams, built on `stream`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance (`*stream* `output-stream) &rest` *initargs*

Description The primary method on `initialize-instance` for `output-stream` initializes an output stream. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.) Output streams have one additional initialization argument.

Arguments

stream	A stream.
initargs	This is the additional initialization argument for output streams.
`:direction`	The direction of the output stream. The default value of `:direction` is `:output.`

`stream-direction` [*Generic function*]

Syntax `stream-direction (`*stream* `stream)`

Description The `stream-direction` generic function reads the direction of *stream*. It will return `:input`, `:output`, `:io`, or `:closed.`

Argument *stream* A stream.

`stream-tyo` [*Generic function*]

Syntax `stream-tyo` (*stream* `output-stream`) *char*

Description The `stream-tyo` generic function directs *stream* to output *char* in an appropriate way; for example, if the stream is a window, `stream-tyo` displays the character in the window. This function must be defined for all output streams.

Arguments

stream	A stream.
char	A character or an ASCII value.

Example

The Common Lisp function `write-char` can be defined as

```
? (defun write-char
        (char &optional (stream *standard-output*))
        (if (eq stream t) (setq stream *terminal-io*))
        (stream-tyo stream char))
WRITE-CHAR
```

`stream-force-output` [*Generic function*]

Syntax `stream-force-output` (*stream* `output-stream`)

Description The generic function `stream-force-output` physically writes all pending output. It is used for streams that buffer their write operations. For example, it doesn't make sense to do an operating system call or a physical disk access to write a byte every time `stream-tyo` is applied to a disk file. The output is stored in a buffer and written to disk after a certain accumulation or when `stream-force-output` is called. Buffering can significantly increase the speed of streams that have a high per-operation overhead (such as disk output and graphics).

The `stream-force-output` function should be defined for buffered output streams.

Argument *stream* An output stream.

stream-tyi

[*Generic function*]

Syntax `stream-tyi` (*stream* `input-stream`)

Description The generic function `stream-tyi` reads the next character from the stream and returns it. If this function is at end-of-file, it returns `nil`. Input functions such as `read` and `read-line` work by making repeated calls to `stream-tyi`. This function must be defined for all input streams.

The `stream-tyi` function should never be applied to a Listener directly, but only via `*terminal-io*`.

Argument *stream* An input stream.

Example

The Common Lisp function `read-char` can be defined as

```
? (defun read-char (&optional (stream *standard-input*)
                    (eof-error-p t)
                    eof-value recursive-p)
       (declare (ignore recursive-p))
       (if (eq stream t)
         (setq stream *terminal-io*)
         (or (stream-tyi stream)
             (if eof-error-p
               (error "End of file on ~s" stream)
               eof-value))))
READ-CHAR
```

stream-untyi

[*Generic function*]

Syntax `stream-untyi` (*stream* `input-stream`) *char*

Description The generic function `stream-untyi` unreads *char* from the stream, effectively pushing it back onto the head of the stream. The next call to `stream-tyi` returns *char*. The `stream-untyi` function cannot be called several times in a row; it can be called only once for each call to `stream-tyi`, and *char* must be the character that was returned by the last call to `stream-tyi`. The `stream-untyi` function must be defined for all input streams.

The `stream-untyi` function is usually implemented in one of two ways: If the stream contains a pointer to a file, string, or other data record, `stream-untyi` simply decrements the pointer. If the stream does not contain a pointer to a data record, then `stream-untyi` sets a variable to the value of *char*. The `stream-tyi` function cooperates by checking the value of the variable; if it is not `nil`, it returns that value instead of getting input from the normal source.

Arguments

stream	An input stream.
char	The last character read from the stream.

Example

The Common Lisp function `unread-char` can be defined as

```
? (defun unread-char (char &optional
                           (stream *standard-input*))
         (if (eq stream t)
           (setq stream *terminal-io*)
           (stream-untyi stream char)))
UNREAD-CHAR
```

---

stream-writer [*Generic function*]

Syntax `stream-writer` (*stream* `stream`)

Description The generic function `stream-writer` returns two values, a function and a value. Applying the function to the value is equivalent to applying `stream-tyo` to the stream, but is usually much faster. Users can specialize `stream-writer`, but they need to be sure that there are no `stream-tyo` methods specialized on a subclass of the class on which the `stream-writer` method is specialized. The `maybe-default-stream-writer` macro knows how to ensure that there are no such `stream-tyo` methods.

Argument *stream* A stream.

---

stream-reader [*Generic function*]

Syntax `stream-reader` (*stream* `stream`)

Description The generic function `stream-reader` returns two values, a function and a value. Applying the function to the value is equivalent to applying `stream-tyi` to the stream, but is usually much faster. Users can specialize `stream-reader`, but they need to be sure that there are no `stream-tyi` methods specialized on a subclass of the class on which the `stream-reader` method is specialized. The `maybe-default-stream-reader` macro knows how to ensure that there are no such `stream-tyi` methods.

Argument *stream* A stream.

Example

Here is an example of the use of `stream-reader` and `stream-writer` to define a function that, given two filenames, copies the data fork of the first file to the data fork of the second file. The second file is created if it does not exist. Because `:if-exists :overwrite` has not been specified for the second file, the function signals an error if the second file already exists.

```
? (defun my-copy-file (from-file to-file &aux char)
       (with-open-file (from-stream from-file)
         (with-open-file
           (to-stream to-file :direction :output)
           (multiple-value-bind
             (reader reader-arg)
             (stream-reader from-stream)
             (multiple-value-bind
               (writer writer-arg)
               (stream-writer to-stream)
               (loop
                 (unless
                   (setq char (funcall reader reader-arg))
                   (return))
                 (funcall writer writer-arg char)))))))
MY-COPY-FILE
```

---

maybe-default-stream-writer [*Macro*]

Syntax `maybe-default-stream-writer` (*stream class*) {*form*}$^+$

Description If the `stream-tyo` method for *stream* is the same as the one for an instance of *class*, the macro `maybe-default-stream-writer` returns the value or values of the last *form*. Otherwise, it returns two values: the effective method for applying `#'stream-tyo` to *stream*, and *stream* itself.

Because maybe-default-stream-writer returns the effective method rather than #'stream-tyo, it avoids method-dispatch.

Arguments

stream	A stream.
class	A Lisp class.
form	One or more Lisp forms.

Example

See the example under maybe-default-stream-reader.

---

maybe-default-stream-reader [*Macro*]

Syntax maybe-default-stream-reader (*stream class*) {*form*}+

Description If the stream-tyi method for *stream* is the same as the one for an instance of *class*, the macro maybe-default-stream-reader returns the value or values of the last *form*. Otherwise, it returns two values: the effective method for applying #'stream-tyi to *stream*, and *stream* itself. Because maybe-default-stream-reader returns the effective method rather than #'stream-tyi, it avoids method-dispatch.

Arguments

stream	A stream.
class	A Lisp class.
form	One or more Lisp forms.

Example

Here are examples of stream-writer and stream-reader created using maybe-default- methods.

The MCL file-stream class stores a Macintosh pointer containing a parameter block and a buffer for use with the Macintosh #_Read and #_Write traps. Its stream-writer method looks something like this:

```
? (defvar *file-stream-class* (find-class 'file-stream))
*FILE-STREAM-CLASS*
? (defmethod stream-writer ((stream file-stream))
   (maybe-default-stream-writer (stream *file-stream-class*)
      (values #'%ftyo       ; low-level character output
function
              (fblock stream))))  ; parameter block pointer
```

The stream-reader method for the file-stream class is very similar to its stream-writer method:

```
? (defmethod stream-reader ((stream file-stream))
   (maybe-default-stream-reader (stream *file-stream-class*)
```

```
          (values #'%ftyi  ; low-level character output function
                   (fblock stream))))  ; parameter block pointer
STREAM-READER
```

stream-peek [*Generic function*]

Syntax `stream-peek` (*stream* `stream`)

Description The `stream-peek` generic function returns the last character read into *stream* without removing it from the queue. The next call to `stream-tyi` or `stream-peek` reads the same character.

Argument *stream* A stream.

The default `stream-peek` method is defined as

```
? (defmethod stream-peek ((stream stream))
           (let ((char (stream-tyi stream)))
             (when char (stream-untyi stream char) char)))
#<STANDARD-METHOD STREAM-PEEK (STREAM)>
```

stream-column [*Generic function*]

Syntax `stream-column` (*stream* `stream`)

Description The generic function `stream-column` returns the current column of the stream. This is used for tabbing purposes and may also be used by `stream-fresh-line`.

Argument *stream* A stream.

stream-line-length [*Generic function*]

Syntax `stream-line-length` (*stream* `stream`)

Description The generic function `stream-line-length` returns the line length of the stream.

Argument *stream* A stream.

`stream-fresh-line` [*Generic function*]

Syntax `stream-fresh-line` (*stream* `stream`)

Description The generic function `stream-fresh-line` is called by the Lisp function `fresh-line`. The usual version simply prints a new line. Output streams should provide a definition of this function if they want `fresh-line` to work properly.

The general idea behind `fresh-line` is that it prints a new line if the stream is not already at the beginning of a line.

Argument

stream	A stream.

`stream-write-string` [*Generic function*]

Syntax `stream-write-string` (*stream* `stream`) *string start end*

Description The generic function `stream-write-string` writes to *stream* the characters of *string* between *start* and *end.*

The expression

```
(stream-write-string stream string 0 (length
string))
```

will be faster than

```
(dotimes (i (length string))
  (stream-tyo stream (char string i)))
```

Arguments

stream	A stream.
string	A string.
start	The beginning of the range to copy. The location `0` is before the first character, `1` between the first and the second, and so on.
end	The end of the range to copy.

Example

```
? (stream-write-string *terminal-io* "Hi there" 0 1)
H
NIL
```

`stream-clear-input` [*Generic function*]

Syntax `stream-clear-input` (*stream* `stream`)

Description The generic function `stream-clear-input` deletes all pending input from the stream. This function is normally defined only for buffered input streams, such as the terminal stream or serial streams.

Argument *stream* A stream.

`stream-eofp` [*Generic function*]

Syntax `stream-eofp` (*stream* `stream`)

Description The generic function `stream-eofp` returns true if the stream is at its end (that is, if there is no more data to read) and false if there is more data to read from the stream.

The `stream-eofp` function must be defined for all input streams.

Argument *stream* A stream.

`stream-listen` [*Generic function*]

Syntax `stream-listen` (*stream* `stream`)

Description The generic function `stream-listen` returns true if there are more characters to read from an input stream and false if there are no more characters. The `stream-listen` method for `stream` is simply `(not (stream-eofp))`. The `stream-listen` function does not normally need to be specialized for new `stream` classes.

Argument *stream* A stream.

`stream-rubout-handler` [*Generic function*]

Syntax `stream-rubout-handler` (*stream* `stream`) *reader*

Description The generic function `stream-rubout-handler` is called by `read` and `read-line` to deal with user editing of the input. This function should call *reader* with one argument, a stream. The default method, specialized on `stream`, simply calls *reader* with an argument of *stream* and provides no deletion handling.

Arguments *stream* A stream.
reader A stream reader.

Example

The following code handles deletions (“rubouts”) for a hypothetical serial I/O stream talking to a dumb terminal:

```
? (defclass serial-io-stream (input-stream output-stream) ())
#<STANDARD-CLASS SERIAL-IO-STREAM>
? (defclass serial-io-stream-rubout-handler
   (input-stream output-stream)
   ((stream :initarg :stream :reader serial-io-stream)
    (buffer :initform (make-array 10
                                  :fill-pointer 0
                                  :adjustable t
                                  :element-type 'character)
            :reader serial-io-stream-buffer)
    (mark :initform nil :accessor serial-io-stream-mark)))
#<STANDARD-CLASS SERIAL-IO-STREAM-RUBOUT-HANDLER>
? (defmethod stream-tyi ((rubout-handler
                          serial-io-stream-rubout-handler))
    (let* ((mark (serial-io-stream-mark rubout-handler))
           (buffer (serial-io-stream-buffer rubout-handler))
           (size (fill-pointer buffer))
           (stream (serial-io-stream rubout-handler))
           (peek (stream-peek stream)))
      (if (and mark (not (eql peek #\rubout)))
        (prog1
          (aref buffer mark)
          (setf (serial-io-stream-mark rubout-handler)
                (and (< (incf mark) size) mark)))
        (let ((char (stream-tyi stream)))
          (when char
            (if (eql #\rubout char)
              (unless (eql 0 size)
                (tyo #\backspace stream)
                (tyo #\space stream)
                (tyo #\backspace stream)
                (setf (fill-pointer buffer) (decf size))
                (setf (serial-io-stream-mark buffer)
                      (and (> size 0) 0))
                (throw rubout-handler nil))
              (progn
                (vector-push-extend char buffer)
                char)))))))
#<STANDARD-METHOD STREAM-TYI (SERIAL-IO-STREAM-RUBOUT-
HANDLER)>
? (defmethod stream-rubout-handler
```

```
                ((stream serial-io-stream) reader)
    (let ((rubout-handler
              (make-instance 'serial-io-stream-rubout-handler
               :stream stream)))
      (loop
        (catch rubout-handler
          (return (funcall reader rubout-handler))))))
#<STANDARD-METHOD STREAM-RUBOUT-HANDLER (SERIAL-IO-STREAM T)>
```

`stream-close` [*Generic function*]

Syntax `stream-close` (*stream* `stream`)

Description The generic function `stream-close` tells the stream that a program is finished with it. After being closed, the stream cannot be used for input or output. Methods on `stream-close` set the `stream-direction` to `:closed` and may perform various cleanup operations, such as disposing of data structures that are no longer needed.

Some streams may be reopened after they have been closed. However, reopened streams have generally lost their previous state.

Argument *stream* A stream.

`stream-abort` [*Generic function*]

Syntax `stream-abort` (*stream* `stream`)

Description When the function `close` is called with a true `:abort` keyword, the generic function `stream-abort` is called. The `stream-abort` generic function should handle any bookkeeping for an abnormal closing of the stream.

Argument *stream* A stream.

Obsolete functions

Since Macintosh Common Lisp now uses generic functions, the following two functions are obsolete. They are included for backward compatibility.

`tyi` [*Function*]

Syntax `tyi &optional` *stream*

Description The `tyi` function reads one character from *stream* and returns it, using `stream-tyi`. The character is echoed if *stream* is interactive, except that the DEL character is not echoed. This function is included for compatibility with earlier versions of Lisp.

Argument *stream* A stream. The default value is `*standard-input*`.

`tyo` [*Function*]

Syntax `tyo` *char* `&optional` *stream*

Description The `tyo` function outputs *char* to *stream*, using `stream-tyo`. It is included for compatibility with earlier versions of Lisp.

Arguments *char* An integer or a character object.

stream A stream. The default value is `*standard-output*`.

Chapter 14:

Programming the Editor

Contents

This chapter describes the functions and concepts needed to program Fred the editor. You should read it if you are creating an application that uses text editing, or if you want to extend the editing capabilities of Fred, to improve your programming environment.

Before reading this chapter you should be familiar with the standard functionality of the editor, Fred, discussed in Chapter 1: Editing in Macintosh Common Lisp. You should also be familiar with the way that Macintosh Common Lisp handles windows, discussed in Chapter 4: Views and Windows.

If you are creating editable dialog items, you should also read Chapter 5: Dialog Items and Dialogs.

Fred Items and Containers

Text editing in MCL takes place inside instances of `fred-item` and `fred-dialog-item`. Both of these classes inherit from `fred-mixin`. `fred-mixin` provides the basic functionality for displaying and editing text with Fred.

`fred-dialog-items` are used to create editable text fields inside dialog boxes. `fred-items` are used inside `fred-windows`, the built-in text editor windows of MCL

Every Fred window contains at least one `scrolling-fred-view` which in turn contains a `fred-item`. (Although `fred-window` and `scrolling-fred-view` do not inherit from `fred-item`, they do support many Fred operations by delegating to the `fred-item` they contain.)

In addition to containing a `fred-item`, every scrolling-fred-view also contains one or two scroll bars.

The actual text being edited in a Fred item or Fred dialog-item is stored in a **buffer.** Macintosh Common Lisp does not keep a file open while it is being edited. It merely reads the file into a buffer, lets you edit it, then reopens the file when you save the buffer.

A buffer is implemented like a more efficiently editable string. (Conceptually, it is a sequence of characters, each of which has an associated font.) The internal representation of a Fred buffer allows characters to be inserted or deleted quickly.

Operations are performed within buffers at places called **buffer marks.** Roughly, a buffer mark indicates any interesting place in the buffer, such as the place where text is inserted, the beginning of a selection range, or the point at which text becomes visible in the window. Buffer marks contain a position and a pointer to their owning buffer. A Fred buffer can be accessed only through its buffer marks.

The buffer is displayed in the window containing the Fred item. A window's contents include the whole buffer, not only the part currently visible on screen (so, to take a trivial example, you can select or search the entire contents of a Fred window, not only the visible portion of the buffer).

In general, higher-level editing operations, such as setting fonts, are implemented as methods applied to instances of `fred-mixin`, the class that governs the behavior of Fred items and Fred dialog items. Fred windows and scrolling Fred views also often have delegating methods on these high-level operations. Low-level operations take a buffer mark as an argument.

Each of these concepts is discussed in more detail in the paragraphs that follow.

Fred windows and Fred views

Fred windows are the main editors of MCL. The class `fred-window` is a subclass of `window` and a superclass of `listener`.

Fred windows are output streams and may be used in any situation that calls for a stream. Characters output to a Fred window stream are inserted at the insertion point, which corresponds to a buffer mark. Stream output to Fred windows is buffered and is not displayed until `fred-update` or `force-output` is called.

A Fred window contains:

- at least one scrolling Fred view
- a minibuffer, a small independent display at the bottom of the window, used to display MCL messages. The minibuffer is also a Fred item.
- an insertion point, indicated by a blinking vertical line, where typing is usually inserted in the window. The insertion point is a buffer mark.
- a display start position, indicating the beginning of the first line of the buffer displayed in the buffer's window. The display start position is also a buffer mark.
- a filename string.

Fred windows may also have

- a selection range, which is displayed in inverse video and is defined by a buffer mark. The buffer mark that defines the selection range is distinct from the insertion point, although most Fred window functions try to keep the insertion point at one end of the selection range.

A scrolling Fred view contains:

- a Fred item, which displays the contents of a buffer.
- zero, one, or two scroll bars.

Fred dialog items

Fred dialog items, a subclass of `simple-view`, include any dialog item with editable text. Their class is `fred-dialog-item`, whose superclasses include the class `fred-mixin` and an internal class whose superclasses are `key-handler-mixin` and `dialog-item`.

A Fred dialog item has

- a view size
- a view position
- a default font
- one or more colors
- an event handler
- a key handler
- text

It also knows whether or not it is enabled.

For a full description of dialog items, see “Dialog items” on page 188.

Buffers and buffer marks

A **buffer** is a sequence of characters much like a string. However, the implementation of buffers makes the insertion and deletion of characters much more efficient than with strings. The characteristics of buffers are inherited from `fred-mixin`.

A buffer has

- a set of buffer marks
- a modification counter, which is incremented any time the buffer is modified
- a property list

There is no data type for buffers; the representation of buffers is internal to the MCL implementation. Buffers keep track of their operations by using buffer marks and are accessed only through buffer marks.

A **buffer mark** contains a position and a pointer to its buffer. A buffer mark indicates a position in a buffer where some editing process might take place. For example, the position at which new characters can be entered into a buffer is indicated by a buffer mark. The positions of buffer marks are recalculated each time the buffer is modified. For example, when you type new characters or paste a selection into a buffer, the position of the buffer mark corresponding to the current insertion point changes, and so do the positions of buffer marks located after the newly inserted text.

Buffer marks are defined by the data type `buffer-mark`. An instance of `buffer-mark` contains

- a pointer to its owning buffer
- a position in its buffer, dynamically updated as the buffer changes

The following properties of a buffer mark can be determined:

- A direction, forward or backward. The direction determines what happens when a character is inserted at the position of the mark. Forward marks move forward, placing themselves after the new character; backward marks stay behind the new character. The insertion point is initially a forward mark but can be changed to a backward mark.
- Other information, such as contents and font information.

Copying and deletion mechanism: The kill ring

For deletion and copying, Fred supports both an Emacs-style kill ring and the Macintosh Clipboard.

The **kill ring** is a list of blocks of text that have been deleted. Any command that deletes or copies text saves that text on the kill ring. There is only one kill ring, shared among all buffers; with it you can move or copy text from buffer to buffer. Cut, Copy, and Emacs commands such as Control-K (`ed-kill-line`) move text onto the kill ring. Also saved on the kill ring is text deleted incidentally, for example, text that is deleted when you type or paste over a selection. Saving all deleted text onto the kill ring provides a level of safety not supported by the usual Macintosh Undo mechanism.

The Macintosh commands Cut, Copy, and Paste move text to and from the Clipboard. The Paste command ignores the kill ring, always pasting from the Clipboard.

See “Working with the kill ring” on page 512 for additional details.

MCL expressions relating to buffer marks

The following MCL expressions relate to buffer marks.

`buffer-mark` [*Class name*]

Description The `buffer-mark` class is the class of buffer marks.

The following functions govern the operation of buffers.

`buffer-mark-p` [*Function*]

Syntax `buffer-mark-p` *thing*

Description The `buffer-mark-p` function returns `t` if and only if *thing* is a buffer mark; otherwise, it returns `nil`. It has the same effect as `(typep` *thing* `'buffer-mark)`.

Argument

thing	Any Lisp object.

`make-mark` [*Function*]

Syntax `make-mark` *buffer-mark* `&optional` *position backward-p*

Description The `make-mark` function creates and installs a new mark in the same buffer as *buffer-mark*, with the specified position and direction. If given, *position* must be a mark or an integer. The new mark is returned.

Arguments

buffer-mark	A buffer mark.
position	A position in a buffer. It can be `nil` (the default), an integer offset from the beginning of the buffer, a mark, or `t`, meaning the end of the buffer. Its default value is `(buffer-position` *buffer-mark*`)`.
backward-p	A Boolean value. If its value is `t`, a backward mark is created. The default value is `nil`.

`set-mark` [*Function*]

Syntax `set-mark` *buffer-mark position*

Description The `set-mark` function sets the position of *buffer-mark* to *position* and returns the updated mark.

Arguments

buffer-mark	A buffer mark.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer, a mark, or `t`, meaning the end of the buffer.

`move-mark` [*Function*]

Syntax `move-mark` *buffer-mark* `&optional` *distance*

Description The `move-mark` function moves *buffer-mark* an amount specified by *distance*, which should be an integer. This function is equivalent to `(set-mark` *buffer-mark* `(+ (buffer-position` *buffer-mark*`)` *distance*`))`.

Arguments

buffer-mark	A buffer mark.
distance	A positive or negative integer specifying the distance to move the mark. The default value is 1.

`mark-backward-p` [*Function*]

Syntax `mark-backward-p` *buffer-mark*

Description The `mark-backward-p` function returns `t` if *buffer-mark* is a backward mark; otherwise, it returns `nil`.

Argument

buffer-mark	A buffer mark.

`same-buffer-p` [*Function*]

Syntax `same-buffer-p` *buffer-mark1 buffer-mark2*

Description The `same-buffer-p` function returns `t` if *buffer-mark1* and *buffer-mark2* are buffer marks pointing to the same buffer.

Arguments

buffer-mark1	A buffer mark.
buffer-mark2	Another buffer mark.

`make-buffer` [*Function*]

Syntax `make-buffer &key :chunk-size :read-only :font`

Description The `make-buffer` function returns a buffer mark representing a new, empty buffer.

Arguments

`:chunk-size`	The length of each string making up a buffer, which is implemented as a linked list of strings. The default is `#x1000` (`4096`) for a `fred-window` and `128` for a `fred-dialog-item`. (The generic function `fred-chunk-size` returns the actual length.)

`:read-only`	The read / write status of the buffer, specifying whether information in the buffer can be modified. If the value of `:read-only` is true, any attempt to modify the buffer will cause an error to be signaled. The default value is `nil`.
`:font`	The display font for characters typed into the buffer. The default is the value of `*fred-default-font-spec*`.

Example

```
? (setq my-buffer (make-buffer))
#<BUFFER-MARK 0/0>
```

---

buffer-size [*Function*]

Syntax `buffer-size` *buffer-mark*

Description The `buffer-size` function returns the number of characters in *buffer-mark*.

Argument *buffer-mark* A buffer mark.

Example

Here is an example of the use of `buffer-size`. (The generic function `fred-buffer` returns the insertion point associated with a Fred window; it is documented on page 489.)

```
? (setf my-window (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4E15D9>
? (buffer-size (fred-buffer my-window))
0
```

---

buffer-modcnt [*Function*]

Syntax `buffer-modcnt` *buffer-mark*

Description The `buffer-modcnt` function returns the modification count of *buffer-mark*. The modification count is the number of times the buffer has been modified since it was created. By comparing the value returned by `buffer-modcnt` at different times, you can tell whether the buffer has been modified in the meantime.

Argument *buffer-mark* A buffer mark.

Example

This code shows how you might use `buffer-modcnt` to determine whether a buffer has been modified.

```
(let ((start-count (buffer-modcnt buffer-mark)))
   (maybe-do-something buffer-mark)
   (unless (eql (buffer-modcnt buffer-mark) start-count)
      (princ "Did something!")))
```

---

buffer-plist [*Function*]

Syntax `buffer-plist` *buffer-mark*

Description The `buffer-plist` function returns the property list of the buffer containing *buffer-mark*. The system itself keeps certain information on buffer property lists, so you should not use `setf` with `buffer-plist`.

At present, Macintosh Common Lisp uses the buffer's property list for nothing except storing the value of `fred-package`.

Argument *buffer-mark* A buffer mark.

Example

```
? (buffer-plist my-buffer)
NIL
```

---

buffer-getprop [*Function*]

Syntax `buffer-getprop` *buffer-mark key* `&optional` *default*

Description The `buffer-getprop` function looks up the *key* property on the property list (`buffer-plist` *buffer-mark*). It returns the value associated with *key*, if found; otherwise, it returns *default*.

Arguments

buffer-mark	A buffer mark.
key	A property on the property list associated with *buffer-mark*.
default	The default value to be returned.

Example

See the example under `buffer-putprop.`

`buffer-putprop` [*Function*]

Syntax `buffer-putprop` *buffer-mark key value*

Description The `buffer-putprop` function gives *key* the value *value* on the property list (`buffer-plist` *buffer-mark*). The value *value* is returned.

Arguments

buffer-mark	A buffer mark.
key	The key to set in the property list.
value	The new value to associate with *key*.

Example

```
? (buffer-putprop my-buffer :font '("Times" 12))
("Times" 12)
? (buffer-getprop my-buffer :font)
("Times" 12)
? (buffer-plist my-buffer)
(:FONT ("Times" 12))
```

`buffer-position` [*Function*]

Syntax `buffer-position` *buffer-mark* `&optional` *position*

Description The `buffer-position` function returns the position (number of characters from the start of *buffer-mark*) of *position* in *buffer-mark*. If *position* is `nil` (the default) or not supplied, the value of `(buffer-position` *buffer-mark*`)` is returned. If *position* is an integer, `buffer-mark` checks that *position* is in the range of legal buffer positions, then returns *position*. If *position* is a mark in the same buffer as *buffer-mark*, its position is returned. Otherwise, an error is signaled.

Arguments

buffer-mark	A buffer mark.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer, a mark, or `t`, meaning the end of the buffer. Its default value is `(buffer-position` *buffer-mark*`)`.

`buffer-line` [*Function*]

Syntax `buffer-line` *buffer-mark* `&optional` *position*

Description The `buffer-line` function returns the line number of *buffer-mark* that contains *position*.

Arguments	*buffer-mark*	A buffer mark.
	position	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark*. Its default value is `(buffer-position` *buffer-mark*`)`.

`buffer-line-start` [*Function*]

Syntax `buffer-line-start` *buffer-mark* `&optional` *start count*

Description The `buffer-line-start` function returns two values. If there are enough lines in the buffer, `buffer-line-start` returns as the first value the position of the start of the *count*th line from the line containing `start` and, as the second value, `nil`. If there aren't enough lines, `buffer-line-start` returns as the first value the end of the range searched (the start of the buffer if *count* is negative, the end of the buffer if *count* is positive) and, as the second value, an integer specifying the number of lines of shortfall.

Arguments	*buffer-mark*	A buffer mark.
	start	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark*.
	count	The number of lines from *start* to search. A *count* of 0 means the start of the line containing *start*, a *count* of –1 means the start of the previous line, a *count* of 1 means the start of the next line, and so on. The default value is 0.

`buffer-line-end` [*Function*]

Syntax `buffer-line-end` *buffer-mark* `&optional` *end count*

Description The `buffer-line-end` function returns two values. If there are enough lines in the buffer, `buffer-line-end` returns, as the first value, the position of the start of the *count*th line from the line containing *end* and, as the second value, `nil`. If there aren't enough lines, `buffer-line-end` returns the end of the range searched (the start of the buffer if *count* is negative, the end of the buffer if *count* is positive) and a second value specifying the number of lines of shortfall.

Arguments	*buffer-mark*	A buffer mark.
	end	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark*.

count	The number of lines from *end* to search. A *count* of 0 means the end of the line containing *end*, a *count* of –1 means the end of the previous line, a *count* of 1 means the end of the next line, and so on. The default value is 0.

`buffer-column` [*Function*]

Syntax `buffer-column` *buffer-mark* `&optional` *position*

Description The `buffer-column` function returns the number of characters between *position* and the start of the line that contains *position*.

Arguments

buffer-mark	A buffer mark.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark*. Its default value is (`buffer-position` *buffer-mark*`)`.

`lines-in-buffer` [*Function*]

Syntax `lines-in-buffer` *buffer-mark*

Description The `lines-in-buffer` function returns the number of lines in the buffer.

This function works by counting the number of newline characters in the buffer; therefore it takes longer to run as the buffer grows in size.

Argument

buffer-mark	A buffer mark.

`buffer-char` [*Function*]

Syntax `buffer-char` *buffer-mark* `&optional` *position*

Description The `buffer-char` function returns the character at the specified position in *buffer-mark*.

Arguments

buffer-mark	A buffer mark.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark*. Its default value is (`buffer-position` *buffer-mark*`)`.

buffer-char-replace [*Function*]

Syntax `buffer-char-replace` *buffer-mark char* `&optional` *position*

Description The `buffer-char-replace` function replaces the character at the specified position in *buffer-mark* with *char*. It returns the old character.

Arguments

buffer-mark	A buffer mark.
char	A character to insert in the buffer.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark*. Its default value is (`buffer-position` *buffer-mark*).

buffer-insert [*Function*]

Syntax `buffer-insert` *buffer-mark string* `&optional` *position*

Description The `buffer-insert` function inserts *string* into *buffer-mark* at *position*.

Arguments

buffer-mark	A buffer mark.
string	Anything acceptable to the `string` function: that is, a string, symbol, or character.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark*. Its default value is (`buffer-position` *buffer-mark*).

buffer-substring [*Function*]

Syntax `buffer-substring` *buffer-mark one-end* `&optional` *other-end*

Description The `buffer-substring` function returns a simple string of the characters in *buffer-mark* in the range described by the arguments.

The order of the *-end* arguments doesn't matter; they are interpreted in whatever order produces a meaningful result.

Arguments

buffer-mark	A buffer mark.
one-end	One end of the string to be returned.
other-end	The other end of the string. The default value is the position of *buffer-mark*.

`buffer-insert-substring` [*Function*]

Syntax `buffer-insert-substring` *buffer-mark string* `&optional` *start end position*

Description The `buffer-insert-substring` function inserts the substring of *string* specified by *start* and *end* into *buffer-mark* at *position*.

Arguments

buffer-mark	A buffer mark.
string	Anything acceptable to the `string` function: that is, a string, symbol, or character.
start	The starting position. The default value is 0.
end	The ending position. The default value is `(length` *string*`)`.
position	An integer position, or a mark in the same buffer as *buffer-mark*. The default value is `(buffer-position` *buffer-mark*`)`.

`buffer-insert-with-style` [*Function*]

Syntax `buffer-insert-with-style` *buffer-mark string style* `&optional` *start*

Description The `buffer-insert-with-style` function inserts *string* in *buffer-mark*. If *style* is given, the function sets the style of *buffer-mark* to *style*, beginning at *start*.

Arguments

buffer-mark	A buffer mark.
string	Anything acceptable to the `string` function: that is, a string, symbol, or character.
style	A font style.
start	The starting position. The default value is the position of *buffer-mark*.

`buffer-current-sexp` [*Function*]

Syntax `buffer-current-sexp` *buffer-mark* `&optional` *position*

Description The `buffer-current-sexp` function returns two values. The first is the s-expression in *buffer-mark* at *position*. Because this function actually reads the characters from the buffer, you may evaluate what it returns. It returns `nil` if there is no s-expression at *position*.

The second value returned is `t` if an s-expression was found at *position*, or `nil` if no s-expression was found at *position*.

Arguments

buffer-mark	A buffer mark.
position	An integer position, or a mark in the same buffer as *buffer-mark*. The default value is `(buffer-position` *buffer-mark*`)`.

The definition of the current s-expression is determined according to the following rules:

- If *position* precedes an open parenthesis, the current s-expression is the text between that open parenthesis and its matching close parenthesis.

  position`(......`*current s-expression*`.......)`

- If *position* follows a close parenthesis, the current s-expression is the text between that close parenthesis and its matching open parenthesis.

  `(......`*current s-expression*`.......)`*position*

- If *position* precedes a quotation mark (that is, double quotes), the current
s-expression is the text between that quotation mark and its matching quotation mark.

  position `"......`*current s-expression*`......."`

- If *position* follows a quotation mark, the current s-expression is the text between that quotation mark and its matching quotation mark.

  `"......`*current s-expression*`......."` *position*

- If *position* is immediately before, immediately after, or in the middle of a symbol, the current s-expression is the symbol.
- Otherwise there is no current s-expression.

---

buffer-current-sexp-start [*Function*]

Syntax `buffer-current-sexp-start` *buffer-mark* `&optional` *position*

Description The `buffer-current-sexp-start` function returns the position of the current s-expression, or `nil` if there is no current s-expression. The current s-expression is determined according to the rules described in the definition of `buffer-current-sexp`.

Arguments

buffer-mark	A buffer mark.
position	An integer position, or a mark in the same buffer as *buffer-mark*. The default value is `(buffer-position` *buffer-mark*`)`.

`buffer-current-sexp-bounds` [*Function*]

Syntax `buffer-current-sexp-bounds` *buffer-mark* `&optional` *position*

Description The `buffer-current-sexp-bounds` function returns the starting and ending positions of the current s-expression; if there is no current s-expression, it returns `nil`. The current s-expression is determined according to the rules given for the function `buffer-current-sexp` earlier in this section.

Arguments

buffer-mark A buffer mark.

position An integer position, or a mark in the same buffer as *buffer-mark*. The default value is `(buffer-position` *buffer-mark*`)`.

`buffer-delete` [*Function*]

Syntax `buffer-delete` *buffer-mark start* `&optional` *end*

Description The `buffer-delete` function deletes the characters in *buffer-mark* in the range described by the *start* and *end* arguments.

Arguments

buffer-mark A buffer mark.

start The start of the range to delete.

end The end of the range to delete. The default value is (`buffer-position` *buffer-mark*).

`buffer-downcase-region` [*Function*]

`buffer-upcase-region` [*Function*]

`buffer-capitalize-region` [*Function*]

Syntax

`buffer-downcase-region` *buffer-mark start* `&optional` *end*
`buffer-upcase-region` *buffer-mark start* `&optional` *end*
`buffer-capitalize-region` *buffer-mark start* `&optional` *end*

Description These three functions convert the words between *start* and *end* to lowercase, uppercase, and initial capitals, respectively.

Arguments

buffer-mark A buffer mark.

start The start of the region to convert, expressed as an integer or buffer mark.

end	The end of the region to convert, expressed as an integer or buffer mark. The default value is (`buffer-position` *buffer-mark*).

`buffer-char-pos` [*Function*]

`buffer-not-char-pos` [*Function*]

Syntax

`buffer-char-pos` *buffer-mark char-or-string* `&key :start :end :from-end`

`buffer-not-char-pos` *buffer-mark char-or-string* `&key :start :end :from-end`

Description

The `buffer-char-pos` function returns the position of the first occurrence of a character that is an element of *char-or-string* in the buffer between `:start` and `:end`. An error is signaled if `:start` is not less than `:end`. The *char-or-string* argument may be a string or a character. The search is case insensitive; that is, the comparison is done using `char-equal`.

The `buffer-not-char-pos` function performs the same function, except that it returns the position of the first character in the buffer that is not an element of *char-or-string*.

Arguments

buffer-mark	A buffer mark.
char-or-string	A character or string of characters. The string is treated as a set of characters.
`:start`	The first position in the buffer to search. The default value is the cursor position.
`:end`	The last position in the buffer to search. The default value is 0 if `:from-end` is true or the buffer size if `from-end` is false.
`:from-end`	A value determining in which direction the search proceeds. If `:from-end` is true, the search proceeds from `:end` to `:start`. The default value is `nil`.

`buffer-string-pos` [*Function*]

Syntax

`buffer-string-pos` *buffer-mark string* `&key :start :end :from-end`

Description The `buffer-string-pos` function returns the position of the first occurrence of *string* in the buffer between `:start` and `:end`. An error is signaled if `:start` is not less than `:end`. If `:from-end` is non-`nil`, the search proceeds backward. If *string* is not found, `nil` is returned; otherwise, the position of the first character of the string is returned. The search is case insensitive; that is, the comparison is done using `char-equal`.

Arguments

buffer-mark	A buffer mark.
string	A string.
`:start`	The first position in the buffer to search. The default value is the cursor position.
`:end`	The last position in the buffer to search. The default is 0 if `from-end` is true or the buffer size if `from-end` is false.
`:from-end`	A value determining in which direction the search proceeds. If `:from-end` is true, the search proceeds from `:end` to `:start`. The default is `nil`.

`buffer-substring-p` [*Function*]

Syntax `buffer-substring-p` *buffer-mark char-or-string* `&optional` *position*

Description The `buffer-substring-p` function returns `t` if *char-or-string* appears at the specified position in *buffer-mark*. The comparison is case insensitive. The *char-or-string* argument may be a string or a character.

Arguments

buffer-mark	A buffer mark.
char-or-string	A character or string of characters.
position	A position in the buffer. The default value is the position of *buffer-mark*.

`buffer-word-bounds` [*Function*]

Syntax `buffer-word-bounds` *buffer-mark* `&optional` *position*

Description The `buffer-word-bounds` function returns two values, the start and end of the word at *position*. If *position* is not in a word, both values are equal to `(buffer-position` *buffer-mark position*`)`.

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default value is the position of *buffer-mark*.

`buffer-fwd-sexp` [*Function*]

Syntax `buffer-fwd-sexp` *buffer-mark* `&optional` *position end character ignore-#-comments*

Description The `buffer-fwd-sexp` function returns the position of the end of the s-expression starting at *position*. If the s-expression is not closed, the function returns `nil`.

Arguments

buffer-mark A buffer mark.

position A position in the buffer. The default value is the position of *buffer-mark*.

end The last position in the buffer to search. The default value is (`buffer-size` *buffer-mark*).

character A character. The value defaults to `(buffer-char` *buffer-mark position*`)`.

ignore-#-comments
A value that determines whether to ignore initial sharp-sign (number-sign) comments. If the value of this is true, initial sharp-sign comments are skipped; if `nil`, the end of an initial sharp-sign comment is returned.

`buffer-bwd-sexp` [*Function*]

Syntax `buffer-bwd-sexp` *buffer-mark* `&optional` *position over-sharps*

Description The `buffer-bwd-sexp` function returns the position of the start of the s-expression ending at *position*, the value of which defaults to `(buffer-position` *buffer-mark*`)`. If the s-expression is not closed, the function returns `(max 0 (1- position))`.

Arguments

buffer-mark A buffer mark.

position A position in the buffer. The default value is the position of *buffer-mark*.

over-sharps An argument specifying whether to consider a preceding reader macro, such as `#@`, as part of the s-expression. If the value of this is true, reader macros are included; if `nil`, they are not.

`buffer-skip-fwd-wsp&comments` [*Function*]

Syntax `buffer-skip-fwd-wsp&comments` *buffer-mark start end*

Description The `buffer-skip-fwd-wsp&comments` function returns the first position in *buffer-mark* after *start* that is not white space or within a comment.

If *start* is not less than *end*, `nil` is returned.

Arguments

buffer-mark	A buffer mark.
start	The first position in the buffer to search.
end	The last position in the buffer to search.

---

buffer-insert-file [*Function*]

Syntax `buffer-insert-file` *buffer-mark pathname* `&optional` *position*

Description The `buffer-insert-file` function inserts the file specified by *pathname* into *buffer-mark* at *position*.

This function preserves and restores font information; for additional information, see "Using multiple fonts" on page 472.

Arguments

buffer-mark	A buffer mark.
pathname	The pathname of the file to insert.
position	A position in the buffer. The default is `buffer-position` *buffer-mark*`)`.

---

buffer-write-file [*Function*]

Syntax `buffer-write-file` *buffer-mark pathname* `&key :if-exists`

Description The `buffer-write-file` function outputs the contents of the buffer to the file specified by *pathname*.

This function preserves and restores font information; for additional information, see "Using multiple fonts" on page 472.

Arguments

buffer-mark	A buffer mark.
pathname	The pathname of the file to insert.
`:if-exists`	A keyword that specifies what to do if the file already exists. If the value of `:if-exists` is `:error`, an error is signaled. If it is `:supersede`, the file is deleted and a new file is written. If it is `:overwrite` and the file is unlocked, the data fork of the existing file is replaced with the contents of the buffer and the resource fork is modified by the font information for the buffer. If the file is locked, an error is signaled.

Using multiple fonts

Fred supports a multiple font capability, as described in Chapter 1: Editing in Macintosh Common Lisp In addition, you can use functions to manipulate fonts in buffers.

Font-spec information is stored with each buffer. Each character in the buffer is associated with a font spec. (Note that the font information is actually stored as a series of ranges in the buffer; a separate font spec is not stored for each character.) In addition, two global font specifications exist for each buffer; one is the font to use in an empty buffer and the other is the font to use for the next insertion.

Global font specifications

Various functions set the empty buffer font. An insertion into an empty buffer when the font is unspecified is in the empty buffer font. The empty buffer font defaults to `*fred-default-font-spec*`.

Various functions set the next insertion font. However, if it is not set, the font for an insertion into a non-empty buffer depends on whether a character precedes the insertion point and is on the same line as the insertion point. In this case, the insertion font is the same font as the preceding character. When the insertion begins on a new line or at the beginning of the buffer, the insertion font is the same font as the following character. This behavior is called the **font neighbor rule**.

The font neighbor rule determines the font to use for a buffer insertion operation (e.g., `buffer-insert` and `buffer-insert-substring`). However, if a function has previously set the next insertion font to a non-`nil` value, that value overrides the font neighbor rule.

In addition to the functions described here, `buffer-write-file` and `buffer-insert-file` preserve and restore font information in a `'FRED'` resource.

◆ *Note*: if you use another text editor to edit a file, font information may become misaligned with the text.

Style vectors

Programming with multiple fonts introduces a new data structure, a style vector. A style vector can be applied to a series of characters in a buffer. For instance, you can use a style vector to specify that the first 10 characters after a specified position should be displayed in 12-point New York bold, the next 20 characters in 9-point Monaco, and the following 10 characters in 12-point Chicago outline. Style vectors do not automatically take note of their position: when you apply one, you have to specify a position using a buffer mark.

Functions for manipulating fonts and font styles

Use the following functions to manipulate fonts, font-codes, and font styles in buffers.

`buffer-char-font-spec` [*Function*]

Syntax `buffer-char-font-spec` *buffer-mark* `&optional` *position*

Description The `buffer-char-font-spec` function returns the font spec of the character at *position* in *buffer-mark*.

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default is `(buffer-position` *buffer-mark*`)`.

`buffer-current-font-spec` [*Function*]

Syntax `buffer-current-font-spec` *buffer-mark*

Description The `buffer-current-font-spec` function returns the current font spec of *buffer-mark*. Any text added to the buffer is in this font.

Argument

buffer-mark	A buffer mark.

`buffer-set-font-spec` [*Function*]

Syntax `buffer-set-font-spec` *buffer-mark font-spec* `&optional` *start end*

Description The `buffer-set-font-spec` function sets the font spec of *buffer-mark*. If *start* is not given, `buffer-set-font-spec` sets the insertion font. Font specifications always merge with the current font.

Arguments

buffer-mark	A buffer mark.
font-spec	A font spec.
start	The start of the range in the buffer. This can be a mark or a number. The default is `(buffer-position` *buffer-mark*`)`.
end	The end of the range in the buffer. The default is the end of the buffer.

`buffer-replace-font-spec` [*Function*]

Syntax `buffer-replace-font-spec` *buffer-mark old-spec new-spec*

Description The `buffer-replace-font-spec` function replaces the font specified by *old-spec* with the one specified by *new-spec* in the entire buffer and returns the font's index in the buffer's font list. If the font specified by *old-spec* is not in the buffer's font list, the function does nothing and returns `nil`.

Arguments

buffer-mark	A buffer mark.
old-spec	A font specification.
new-spec	A font specification.

Example

This function could be written as follows:

```
? (defun buffer-replace-font-spec (buf old-spec new-spec)
   (multiple-value-bind (old-ff old-ms) (font-codes old-spec)
     (multiple-value-bind (new-ff new-ms)
                          (font-codes new-spec old-ff old-ms)
       (buffer-replace-font-codes buf
          old-ff old-ms new-ff new-ms))))
```

`buffer-font-codes` [*Function*]

Syntax `buffer-font-codes` *buffer*

Description The function `buffer-font-codes` returns a font/face code and a mode/size code. If the font codes for the next insertion are set, the font/face code and the mode/size code for the next insertion are returned; otherwise, the font neighbor rule determines the font/face code and the mode/size code returned.

Arguments

buffer	A Fred buffer.

Example

```
? (setf my-window (make-instance 'fred-window))
#<FRED-WINDOW "New" #x6FBCE1>
? (buffer-font-codes (fred-buffer my-window))
262144
65545
```

---

buffer-set-font-codes [*Function*]

Syntax `buffer-set-font-codes` *buffer ff ms* `&optional` *start end*

Description The `buffer-set-font-codes` function sets the font for *buffer*. If *start* is not `nil`, this function changes the font in the buffer between *start* and *end*. If *start* is `nil` and the buffer is empty, this function sets the font for the empty buffer. If *start* or *end* is `nil`, unspecified, or both have the same value, and the buffer is not empty, this function sets the font for the next insertion.

If either *ff* or *ms* is `nil`, this function clears the next insertion font and the font neighbor rule determines the font for the next insertion.

Arguments

buffer	A Fred buffer.
ff	A font/face code. A font/face code is a 32-bit integer that combines the font's name and its face (e.g., plain, bold, italic). For more information see "Functions related to font codes" on page 80.
ms	A mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (e.g., inclusive-or, exclusive-or, complemented) and the font size.
start	The initial position changed in the buffer. This is a buffer mark, an integer, or `nil`. The default is `nil`.
end	The final position changed in the buffer. This is a buffer mark, an integer, or `nil`. The default is `nil`.

Example

```
? (defvar my-window)
MY-WINDOW
? (setf my-window (make-instance 'fred-window))
```

```
#<FRED-WINDOW "New" #x6FAC81>
? (multiple-value-bind (ff ms) (font-codes '("courier" 12
:plain))
(buffer-set-font-codes (fred-buffer my-window) ff ms))
NIL
```

---

buffer-replace-font-codes [*Function*]

Syntax `buffer-replace-font-codes` *buffer-mark old-ff old-ms new-ff new-ms*

Description The `buffer-replace-font-codes` function replaces the font specified by *old-ff* and *old-ms* with the one specified by *new-ff* and *new-ms* in the owning buffer of *buffer-mark* and returns the font's index in the buffer's font list. If the font specified by *old-ff* and *old-ms* does not exist in the buffer, the function does nothing and returns `nil`.

Arguments

buffer-mark	A buffer mark.
old-ff	The old font / face code.
new-ff	The new font / face code.
old-ms	The old mode / size code.
new-ms	The new mode / size code.

---

buffer-remove-unused-fonts [*Function*]

Syntax `buffer-remove-unused-fonts` *buffer-mark*

Description The `buffer-remove-unused-fonts` function removes unused fonts from the buffer associated with *buffer-mark*.

Argument

buffer-mark	A buffer mark.

---

buffer-get-style [*Function*]

Syntax `buffer-get-style` *buffer-mark* `&optional` *start end*

Description The `buffer-get-style` function returns a style vector corresponding to the fonts, sizes, and styles used in the specified range in the buffer.

Arguments

buffer-mark	A buffer mark.

start	The start of the range in the buffer. This can be a mark or a number. The default is `(buffer-position` *buffer-mark*`)`.
end	The end of the range in the buffer. The default is the end of the buffer.

buffer-set-style [*Function*]

Syntax `buffer-set-style` *buffer-mark style-vector start-position*

Description The `buffer-set-style` function sets the styles used in *buffer-mark*, beginning at *start-position*, according to *style-vector*.

Arguments

buffer-mark	A buffer mark.
style-vector	A style vector.
start-position	The beginning position in the buffer at which to insert the styles. The default is `(buffer-position` *buffer-mark*`)`.

buffer-next-font-change [*Function*]

Syntax `buffer-next-font-change` *buffer-mark* `&optional` *position*

Description The `buffer-next-font-change` function scans the buffer of *buffer-mark* for the first font change following *position* and returns the position of the change. If there are no changes following *position*, `nil` is returned.

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default is `(buffer-position` *buffer-mark*`)`.

buffer-previous-font-change [*Function*]

Syntax `buffer-previous-font-change` *buffer-mark* `&optional` *position*

Description The `buffer-previous-font-change` function scans the buffer of *buffer-mark* for the first font change before *position* and returns the position of the change. If there are no changes before *position*, `nil` is returned.

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default is `(buffer-position` *buffer-mark*`)`.

Fred classes

`fred-mixin` and its subclasses `fred-item` and `fred-dialog-item` provide the basic Fred display and editing behavior. `fred-window` and `scrolling-fred-view` are classes of views which contain `fred-items`.

`fred-mixin` [*Class name*]

Description The `fred-mixin` class defines the basic Fred display and editing behavior. It is a superclass of both `fred-item` and `fred-dialog-item`; it has no instances of its own.

This class does not have a method for `initialize-instance`. It adds the following initialization arguments used by its subclasses:

`:comtab` The command table to use with the buffer. The default is the value of `*comtab*`.

`:copy-styles-p`
An argument that determines whether to copy styles when copying. The default value is `nil`.

`:history-length`
The number of Fred commands that are remembered. Only commands that actually change text are remembered. The default value for Fred windows and Fred dialog items is `*fred-history-length*`; for the Listener, it is `*listener-history-length*`.

`fred-item` [*Class name*]

Description A `fred-item` is a subclass of `fred-mixin` and `key-handler-mixin`. The buffer area in a `scrolling-fred-view` is a `fred-item`.

In addition to the `fred-mixin` and `key-handler-mixin` *initargs*, `fred-item` has the following initial argument:

`:part-color-list`
A list of color specs. The initial value form is `nil`. The accessor is `part-color-list`.

`window-fred-item` [*Class name*]

Description

A `window-fred-item` is a subclass of `fred-item`. The `fred-update` method for this class updates the window title and the window's change mark. Each `scrolling-fred-view` in a Fred window uses this class, with the exception of the mini-buffer.

`fred-dialog-item` [*Class name*]

Description

The `fred-dialog-item` class is the class of Fred dialog items. This class is based on `fred-mixin` and `basic-editable-text-dialog-item` (internal to `:CCL`).

Like any other dialog item, a `fred-dialog-item` can be the subview of any view.

`initialize-instance` [*Generic function*]

Syntax

`initialize-instance` (*item* `fred-dialog-item`) `&rest` *initargs*

Description

The `initialize-instance` primary method for `fred-dialog-item` initializes a Fred dialog item so that it can be used. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item — A Fred dialog item.

initargs — A list of arguments used to initialize the Fred dialog item. The following initialization arguments are available:

`:view-container`
: The item's container. This value is set by `set-view-container`. Its initial value is `nil`.

`:view-font`
: The font in which text in the item appears. The default is `("Chicago" 12 :PLAIN)`.

`:view-position`
: The position in the dialog box where the item will be placed, in the local view coordinates. If this argument is not specified, the first available position large enough to hold the item is used. If no space is large enough, the dialog item is placed in the upper-left corner of the dialog. The default value is `nil`.

`:view-size`
: The size of the Fred dialog item. If not specified, this value is calculated so that the item fits in the view. If the specified value is too small, the item is clipped when it is drawn. The default value is `nil`.

`:view-nick-name`
: The nickname of the Fred dialog item. This feature is used in conjunction with `view-named`. The default value is `nil`.

`:allow-returns`
: An argument specifying whether to allow returns to be typed into the item. If `:allow-returns` is `nil` (the default), pressing the Return key while this item is the window's current key handler will invoke the window's default button, if it has one. A Return character can be inserted by pressing Shift-Return. This value is checked by the accessor `allow-returns-p` and changed by `set-allow-returns`.

`:allow-tabs`
: An argument specifying whether to allow tabs in the buffer. If `:allow-tabs` is `nil` (the default), pressing the Tab key while this item is the window's current key handler will select the next key handler. For example, pressing Tab in an editable text field will move the cursor to the next editable text field. A Tab character can be inserted by pressing Shift-Tab. This value is checked by the accessor `allow-tabs-p` and changed by `set-allow-tabs`.

`:copy-styles-p`
: An argument specifying whether to copy styles when copying. The default value is `nil`.

`:dialog-item-text`
: The default text to insert in the buffer. The default value is `""`, the empty string.

`:dialog-item-enabled-p`
: An argument specifying whether the dialog item is enabled; the default value is true.

`:part-color-list`
: A list of colors to which the parts of the Fred dialog item should be set. The default value is `nil`. The three possible keywords are `:frame`, the outline of the Fred dialog item; `:text`, its text; and `:body`, its body.

`:draw-outline`
: An argument specifying whether a boxed outline appears around the editable text. The default value is true.

`:buffer-chunk-size`
: The chunk size of the buffer. A buffer is conceptually a linked list of strings; the chunk size is the length of each of these strings. The default value for Fred dialog items is 128.

`:text-edit-sel-p`
: An argument specifying whether text can be selected. The default value is true.

`:comtab`
: The command table to use with the buffer. The default is the value of `*comtab*`.

`:line-right-p`
: A value that indicates the direction that a line of text is printed in the buffer. The trap `#_GETSysJust` determines the default value for `:line-right-p`. If the value is `nil`, the text direction is left to right. If the value is true, the text direction is right to left. The accessor for `:line-right-p` is `fred-line-right-p`.

`:word-wrap-p`
: A value that indicates whether a line wraps on word boundaries. The default is `nil`. The accessor for `:word-wrap-p` is `fred-word-wrap-p`.

`:justification`
: A value that indicates text alignment. The value is either `:left`, `:right`, or `:center`. If `:line-right-p` is `nil`, the default is `:left`; if `:line-right-p` is true, the default is `:right`. The accessor for `:justification` is `fred-justification`.

`fred-window` [*Class name*]

Description The `fred-window` class is the class of Fred windows, based on `fred-mixin` and `window`.

`initialize-instance` [*Generic function*]

Syntax `initialize-instance` (*window* `fred-window`) `&rest` *initargs*

Description The `initialize-instance` primary method for `fred-window` initializes a Fred window so that it can be used. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments *fred-window* A Fred window.

initargs A list of arguments used to initialize the Fred window. The following initialization arguments are available:

`:view-container`
: The view's container. This value is set by `set-view-container`. Its initial value is `nil`.

`:view-font`
: The font specification used by the view. The default is `("Geneva" 0 :PLAIN)`.

`:view-scroll-position`
: The initial scroll position of the view. This position corresponds to the origin in a Macintosh GrafPort. The default value is `#@(0 0)`.

`:view-position`
: A point, keyword, or list giving the initial position of the window. The default position is `#@(6 44)`.

`:view-size`
: A point giving the initial size of the window. The default is `#@(502 150)`.

`:view-nick-name`
: The nickname of the view. The default value is `nil`.

`:filename`
: The name of the file to appear in the window. The default value is `nil`. If the file does not exist, an error is signaled.

`:wrap-p`
: An argument specifying whether text wraps in the window. The default value is `nil`.

`:view-subviews`
: A list of subviews. Fred windows do not normally contain subviews.

`:window-title`
: A string specifying the title of the window. The title of a Fred window is computed from the pathname of the file displayed in it, if there is one; a window that is not displaying a file's contents has the title `New`.

`:window-show`
: An argument determining whether a window is shown when it is created. If this argument is true (the default), a window is shown when it is created. If `nil`, the window is created invisibly.

`:window-layer`
: An integer describing the layer in which the new window will be created. By default this is 0 (the front window). For details, see `set-window-layer`, defined on page 172."

`:color-p`
: An argument specifying whether the window is a color window. If `nil` (the default), the window is based on the Macintosh window type. If non-`nil`, the window is a color window.

`:window-type`
: A keyword describing the type of window to be created. The default is `:document-with-zoom`. This argument should be one of the following keywords:
`:document`
`:document-with-grow`
`:document-with-zoom`
`:double-edge-box`
`:single-edge-box`
`:shadow-edge-box`
`:tool`

`:copy-styles-p`
: An argument specifying whether to copy styles when copying. The default value is true.

`:procid`
: A number indicating the procID (window definition ID) of the window to be created. This is an alternative to specifying `:window-type` for programmers who want to use WDEFs with nonstandard procIDs.

`:comtab`
: The command table to use for editing in the buffer.

`:history-length`
: The number of commands retained in the edit history. The default value is the value of `*fred-history-length*`, which is initially 20.

`:help-spec`
: A value describing the Balloon Help for the window. This may be a string or one of a number of more complicated specifications, which are documented in the file `help-manager.lisp` in your Library folder. The default value is `nil`.

`:window-do-first-click`
: A Boolean value determining whether the click that selects a window is also passed to `window-click-event-handler`. Its default value is `nil`.

`:close-box-p`
: A Boolean value determining whether the window will have a close box. Close boxes aren't available on all windows.

`:wptr`
: For use by advanced programmers. An argument determining whether a new window is created or whether a previously existing window record is used. If the argument is not specified, `initialize-instance` calls `#_NewWindow` or `#_NewCWindow`. If the argument is specified, it should be a pointer to a window record on the Macintosh heap.

`:line-right-p`

A value that indicates the direction that a line of text is printed in the buffer. The trap `#_GETSysJust` determines the default value for `:line-right-p`. If the value is `nil`, the text direction is left to right. If the value is true, the text direction is right to left. The accessor for `:line-right-p` is `fred-line-right-p`.

`:word-wrap-p`

A value that indicates whether a line wraps on word boundaries. The default is `nil`. The accessor for `:word-wrap-p` is `fred-word-wrap-p`.

`:justification`

A value that indicates text alignment. The value is either `:left`, `:right`, or `:center`. If `:line-right-p` is `nil`, the default is `:left`; if `:line-right-p` is true, the default is `:right`. The accessor for `:justification` is `fred-justification`.

---

scrolling-fred-view [*Class name*]

Description A `scrolling-fred-view` is a view containing 0, 1, or 2 scroll bars and a `fred-item`. Instances of this class are components of Fred windows and of several dialogs displayed by the Tools menu.

A `scrolling-fred-view` is a subclass of `fred-mixin` . In addition to the `fred-mixin` and `key-handler-mixin` *initargs*, the set of initial arguments and values that initialize `scrolling-fred-view` are:

Initargs:

`:part-color-list`

A list of color specs. The initial value form is `nil`. The accessor is `part-color-list`.

`:grow-box-p` A boolean value that indicates whether the view leaves space below the vertical scroll bar. The default is `nil`, indicating that space is not left for a grow-box. The accessor is `grow-box-p`.

This argument is useful for aesthetic purposes (e.g., when the view size is the same as the window size).

`:h-scroll-fraction`

A value that indicates the size of the horizontal scroller as a fraction of the view width. The value is either `nil` or an integer. The accessor is `h-scroll-fraction`. An integer n indicates a view width fraction of $1/n$.

`:draw-scroller-outline`
: A boolean value that indicates whether an outline is visible around the scroller. The default is true. The accessor is `draw-scroller-outline`.

`:h-scroll-class`
: The class for the horizontal scroller. The default value is `'fred-h-scroll-bar`.

`:v-scroll-class`
: The class for the vertical scroller. The default value is `'fred-v-scroll-bar`.

`:track-thumb-p`
: An argument specifying the scrolling behavior of the view. If `:track-thumb-p` is true, the scroll box and view contents move as the user drags the scroll box because `scroll-bar-changed` is called. If `:track-thumb-p` is `nil`, an outline of the scroll box moves during scrolling and the scroll box and view contents move when the user releases the mouse button. The default value is the value of `*fred-track-thumb-p*`.

`:h-scrollp`
: A boolean value that specifies whether the `scrolling-fred-view` has a horizontal scroll bar. The default is true.

`:v-scrollp`
: A boolean value that specifies whether the `scrolling-fred-view` has a vertical scroll bar. The default is true.

`:bar-dragger`
: A value that determines whether the horizontal or vertical boundary between two panes can be dragged. The default is `nil`. A value of `:vertical` specifies that the horizontal boundary between two panes can be dragged; a value of `:horizontal` specifies that the vertical boundary between two panes can be dragged.

`:h-pane-splitter`
: A value that specifies whether there is a horizontal pane splitter. A horizontal pane splitter resides on a horizontal scroll bar and divides the width of a pane. This initial argument is ignored if the argument `:h-scrollp` is `nil`.

  If `:h-pane-splitter` is `:left`, the horizontal pane splitter is located at the left of the view; if `:h-pane-splitter` is `:right` or any other true value except `:left`, the pane splitter is located at the right of the view. The default value is `nil`.

`:v-pane-splitter`

A value that specifies whether there is a vertical pane splitter. A vertical pane splitter resides on a vertical scroll bar and divides the height of a pane. This initial argument is ignored if `:v-scrollp` is `nil`.

If `:v-pane-splitter` is `:top`, the vertical pane splitter is located at the top of the view; if `:v-pane-splitter` is `:bottom` or any other true value except `:top`, the pane splitter is located at the bottom of the view. The default value is `nil`.

`:fred-item-class`

The class for the view's `fred-item`. The default is `'fred-item`.

Example

The following example adds a `scrolling-fred-view` to a window.

```
? (setf my-window (make-instance 'window
      :window-title "Window 1" :view-size #@(400 300)))
#<FRED-WINDOW "Window 1" #xBC0D09>
? (setf view-1 (make-instance 'scrolling-fred-view
:view-size #@(400 300) :h-pane-splitter :left :bar-dragger
:vertical))
#<SCROLLING-FRED-VIEW #xBEBF51>
? (add-subviews my-window view-1)
```

Fred functions

High-level functions of the editor are defined on `fred-item` and `fred-dialog-item` through the class `fred-mixin`, which defines the general behavior of Fred. In addition, Fred window and scrolling-fred-view support many of the high-level operations through delegation to their active fred-item.

In addition to the functions outlined in this section, many Fred functions are associated with keystrokes. The names and actions of these functions are given in Chapter 1: Editing in Macintosh Common Lisp They take one argument, either `fred-dialog-item`, `fred-item` or `fred-window`.

`fred` [*Function*]

Syntax `fred &optional` *pathname new-window*

Description The `fred` function is a simpler way to create a Fred window. If *pathname* is given, `fred` attempts to open the file with that pathname.

Arguments

pathname A pathname, string, or stream associated with a file. If the file specified by *pathname* does not exist, Macintosh Common Lisp signals an error. If *pathname* is not given, Fred creates an empty Fred window.

new-window A Boolean value. If this value is `t`, Macintosh Common Lisp opens a new window, even if another window is already open to the specified file. Otherwise, Macintosh Common Lisp asks whether to open a new window or select the old one.

`view-mini-buffer` [*Generic function*]

Syntax `view-mini-buffer` (*view* `fred-mixin`)

Description The `view-mini-buffer` generic function returns the minibuffer associated with *view*.

Argument *view* A Fred window or Fred dialog item.

Example

```
? (view-mini-buffer (make-instance 'fred-dialog-item))
#<MINI-BUFFER #x3AAF49>
```

`window-key-handler` [*Generic function*]

Syntax `window-key-handler (`*view* `fred-window)`

Description The `window-key-handler` generic function returns the current key handler of *view*, unless the current key handler is the mini-buffer, in which case this function returns a key handler that is not the mini-buffer.

Arguments *view* A fred-window.

`fred-item` [*Generic function*]

Syntax `fred-item (`*view* `scrolling-fred-view)`

Description The `fred-item` generic function returns the `fred-item` in *view*.

Arguments *view* A `scrolling-fred-view`.

`h-scroller` [*Generic function*]

Syntax `h-scroller (`*view* `scrolling-fred-view)`

Description The `h-scroller` generic function returns the horizontal scroller in a `scrolling-fred-view`.

Arguments *view* A `scrolling-fred-view`.

`v-scroller` [*Generic function*]

Syntax `v-scroller (`*view* `scrolling-fred-view)`

Description The `v-scroller` generic function returns the vertical scroller in a `scrolling-fred-view`.

Arguments *view* A `scrolling-fred-view`.

`add-scroller` [*Generic function*]

Syntax `add-scroller (`*view* `scrolling-fred-view)` *direction* `&key` *pane-splitter*

Description The function `add-scroller` adds a scroller to a `scrolling-fred-view`, if one does not exist in the specified direction.

Arguments

view	A `scrolling-fred-view`.
direction	The keyword `:vertical` or `:horizontal`.
pane-splitter	The `:pane-splitter` initial argument that specifies the pane splitter position. If *pane-splitter* is `nil`, there is no pane splitter. The default value is `nil`. If the scroll bar is `:vertical`, a value of `:top` positions the pane splitter above the scroll bar and any other non-`nil` value positions the pane splitter below the scroll bar.

If the scroll bar is `:horizontal`, a value of `:left` positions the pane splitter to the left of the scroll bar and any other non-`nil` value positions the pane splitter to the right of the scroll bar.

`remove-scroller` [*Generic function*]

Syntax `remove-scroller (`*view* `scrolling-fred-view)` *direction*

Description The function `remove-scroller` removes a scroller from a `scrolling-fred-view`, if one exists in the specified direction.

Arguments

view	A `scrolling-fred-view`.
direction	The keyword `:vertical` or `:horizontal`.

`fred-buffer` [*Generic function*]

Syntax fred-buffer (*view* fred-mixin)

Description The `fred-buffer` generic function returns the buffer mark associated with the insertion point of *view*. The `window-null-event-handler` or `key-handler-idle` generic function displays the blinking vertical bar wherever this mark is located (unless the mark is off the screen, in which case no vertical bar is displayed).

Argument *view* A Fred window or Fred dialog item.

`fred-line-right-p` [*Generic function*]

Syntax `fred-line-right-p (`*view* `fred-mixin)`
`fred-line-right-p (`*view* `fred-window)`

Description The `fred-line-right-p` generic function returns a value that indicates the printing direction of a line of text. If the value is `nil`, the text direction is left to right. If the value is true, the text direction is right to left.

Arguments *view* A `fred-window` or `fred-dialog-item`.

`fred-word-wrap-p` [*Generic function*]

Syntax `fred-word-wrap-p (`*view* `fred-mixin)`
`fred-word-wrap-p (`*view* `fred-window)`

Description The `fred-word-wrap-p` generic function returns a value that indicates whether a line wraps on word boundaries. If the value is `nil`, lines do not wrap on word boundaries. If the value is true, a line wraps on word boundaries.

Arguments *view* A `fred-window` or `fred-dialog-item`.

`fred-justification` [*Generic function*]

Syntax `fred-justification (`*view* `fred-mixin)`
`fred-justification (`*view* `fred-window)`

Description The `fred-justification` generic function returns a value that indicates text alignment. The value is either `:left`, `:center`, or `:right` indicating left alignment, center alignment, or right alignment, respectively.

Arguments *view* A `fred-window` or `fred-dialog-item`.

`grow-box-p` [*Generic function*]

Syntax `grow-box-p (`*view* `scrolling-fred-view)`

Description The `grow-box-p` generic function returns a boolean value that indicates whether the view leaves space below the vertical scroll bar. If the value returned is `nil`, there is no additional space below the vertical scroll bar; if the value is true, there is additional space for a grow-box beneath the vertical scroll bar.

Arguments *view* A `scrolling-fred-view`.

`h-scroll-fraction` [*Generic function*]

Syntax `h-scroll-fraction (`*view* `scrolling-fred-view)`

Description The `h-scroll-fraction` generic function returns a value that indicates the size of the horizontal scroller as a fraction of the view width. An integer *n* indicates a view width fraction of $1/n$.

Arguments *view* A `scrolling-fred-view`.

---

fred-autoscroll-h-p view [*generic function*]

fred-autoscroll-v-p view [*generic function*]

Syntax
`fred-autoscroll-h-p (`*view* `fred-mixin)`
`fred-autoscroll-v-p (`*view* `fred-mixin)`

These generic functions are called indirectly by `(view-click-event-handler fred-mixin)`. They control whether the view will be automatically scrolled when the mouse cursor goes outside of it. The default methods return true. You can specialize this generic function on subclasses of `fred-mixin` in order to change this behavior.

Arguments *view* A `fred-mixin`.

---

draw-scroller-outline [*Generic function*]

Syntax `draw-scroller-outline (`*view* `scrolling-fred-view)`

Description The `draw-scroller-outline` generic function returns a boolean value that indicates whether an outline is visible around the scroller. If the value returned is `nil`, there is not an outline around the scroller; if the value is true, there is an outline around the scroller.

Arguments *view* A `scrolling-fred-view`.

---

fred-chunk-size [*Generic function*]

Syntax `fred-chunk-size (`*view* `fred-mixin)`

Description The `fred-chunk-size` generic function returns the chunk size of *view,* that is, the size of each of the strings through which buffers are implemented.

Argument *view* A Fred window or Fred dialog item.

Example

```
? (fred-chunk-size (fred))
```

4096

fred-display-start-mark

[*Generic function*]

Syntax `fred-display-start-mark` (*view* `fred-mixin`)

Description The `fred-display-start-mark` generic function returns the buffer mark of the first character visible in the window. By moving this mark, you affect which part of the buffer `fred-update` displays.

Note that after every Fred keyboard command, Fred attempts to make the cursor visible on the screen, repositioning the `fred-display-start-mark` if necessary. To disable this behavior for the duration of one Fred command, set the variable `*show-cursor-p*` to `nil`.

Argument

view	A Fred window or Fred dialog item.

set-fred-display-start-mark

[*Generic function*]

Syntax `set-fred-display-start-mark` (*view* `fred-mixin`) *position* `&optional` *no-drawing*

Description The `set-fred-display-start-mark` generic function sets the buffer mark of the first character drawn in the window to *position*. If the value of *no-drawing* is `nil` (the default), the view is redrawn immediately. Otherwise the view is invalidated and will be redrawn the next time an `event-dispatch` occurs.

Arguments

view	A Fred window or Fred dialog item.
position	A position in the window (a mark or a number).
no-drawing	A Boolean value. The default value is `nil`.

show-cursor-p

[*Variable*]

Description The `*show-cursor-p*` variable is bound to `t` by `run-fred-command`. It determines whether the insertion point will be made visible by `run-fred-command` after the command has been executed.

If the value of this variable is true, then the insertion point is made visible.

If the value of this variable is `nil`, *then* it is not made visible.

For further details, see “Fred dispatch sequence” on page 519.

`window-show-cursor` [*Generic function*]

Syntax `window-show-cursor` (*window* `fred-mixin`) `&optional` *position scrolling*

Description The `window-show-cursor` generic function performs an update on *view*, scrolling if necessary in order to make *position* visible.

Arguments

view	A Fred window or Fred dialog item.
position	A position to make visible in the window. The default is the insertion point.
scrolling	A Boolean value. If the value of *scrolling* is `nil`, no scrolling is performed.

`fred-blink-position` [*Generic function*]

Syntax `fred-blink-position` (*view* `fred-mixin`)

Description The `fred-blink-position` generic function returns the position of the parenthesis or quotation mark that matches a parenthesis or quotation mark at the insertion point. If there is no matching character, `fred-blink-position` returns `nil`.

Argument

view	A window.

`fred-update` [*Generic function*]

Syntax `fred-update` (*view* `fred-mixin`)

Description The `fred-update` generic function updates the display of *view*, using the current values of the insertion point, default-position mark, selection region, and the contents of the buffer.

Note that by default, after every Fred keyboard command, Fred attempts to make the insertion point visible on the screen, repositioning the `fred-display-start-mark` if necessary. To disable this behavior for the duration of one Fred command, set the variable `*show-cursor-p*` to `nil`. Several built-in Fred commands (those that do not change the insertion point) disable this behavior.

The `fred-update` function is called automatically in several situations: in response to window update events from the Macintosh Window Manager, which occur due to resizing or uncovering a portion of the window; after the execution of every Fred keyboard command; in response to clicks in the window; and in response to `force-output` calls to the dialog item (views inherit from output streams).

If you modify the window or its buffer in any other context (for example, by using a menu command or a function meant to be called directly by user code), you must call `fred-update` explicitly. Otherwise, the changes you make will not be visible on the screen.

Argument *view* A Fred window or Fred dialog item.

---

ed-insert-char [*Generic function*]

Syntax `ed-insert-char` (*view* `fred-mixin`) *character*

Description The `ed-insert-char` generic function inserts *character* into the text of the Fred window or dialog item at the insertion point. The display is not automatically updated.

Arguments

view	A Fred window or Fred dialog item.
character	Any character.

---

ed-insert-with-style [*Generic function*]

Syntax `ed-insert-with-style` (*view* `fred-mixin`) *string style*
`&optional` *position*

Description The `ed-insert-with-style` generic function inserts *string* at *position* in *view*. If the value of `*paste-with-styles*` is true, it applies *style* over *string*. If the value of `*paste-with-styles*` is `nil`, it does not.;

When you write functions that cause text to be pasted, use `ed-insert-with-style` to ensure consistency with the way Macintosh Common Lisp handles styles.

Arguments

view	A Fred window or Fred dialog item.
string	The string to insert in the buffer. This argument may also be `nil`, in which case nothing is inserted.
style	The style vector to map over the string after it is inserted in the buffer. This argument may also be `nil`, in which case the string is inserted in the current insertion font.
position	A position in the window (a mark or a number). The default is the window's insertion point.

`fred-copy-styles-p` [*Generic function*]

Syntax `fred-copy-styles-p` (*view* `fred-mixin`)

Description The `fred-copy-styles-p` generic function indicates whether or not `copy` from *view* copies styles. The default value is `nil`, meaning that it does not.

Argument *view* A Fred window or Fred dialog item.

`*fred-special-indent-alist*` [*Variable*]

Description The `*fred-special-indent-alist*` variable contains an association list of symbols that Fred should indent specially. The `car` of each pair is the symbol, and the `cdr` is the number of distinguished arguments when the symbol is used as a function, macro, or special form.

`ed-current-symbol` [*Function*]

Syntax `ed-current-symbol` *window* `&optional` *aux-find-symbol start end*

Description The `ed-current-symbol` function returns three values giving information about the text currently selected. If no text is selected, `ed-current-symbol` returns information on the text surrounding the cursor.

The first value is a symbol if the cursor is in an interned symbol, or if the selected text contains a symbol.

The second value is `t` if an interned symbol is found, or `nil` if an interned symbol is not found.

The third value is the character immediately before the selected text or symbol, or `nil` if the selection or symbol is at the start of the buffer.

This function attempts to locate the symbol in the package of *window*, or in the current package if the window doesn't have a package. It never has the effect of interning a new symbol.

Arguments *window* A window.

aux-find-symbol

A function that determines what symbol to return from the currently selected text. (It can be used, for example, to determine that the symbol is in the desired package.) It should return either the name of a function or `nil`.

start	The beginning of the range to look at.
end	The end of the range to look at.

`ed-current-sexp` [*Generic function*]

Syntax `ed-current-sexp` (*view* `fred-mixin`) `&optional` *position* *dont-skip*

Description The `ed-current-sexp` generic function returns two values. If there is a current s-expression in *view* (that is, if there is an s-expression next to the insertion point, or an s-expression at *position,* or a selection), the function returns the s-expression and `t`. If there is no current s-expression, the function returns the two values `nil` and `nil`.

Arguments

view	A Fred window or Fred dialog item.
position	A position in the window (a mark or a number). The default is the window's cursor position.
dont-skip	An argument specifying whether to skip forward across reader macros. If true, the function skips across reader macros in deciding the current s-expression. If `nil`, it does not.

`fred-point-position` [*Generic function*]

Syntax `fred-point-position` (*view* `fred-mixin`) *h* `&optional` *v*

Description The `fred-point-position` generic function returns the buffer position of the character nearest to the point specified by *h* and *v* in the local coordinates of the window containing the Fred dialog item. (See Chapter 2: Points and Fonts for a description of the point format.) This function assumes that the buffer has not been modified since the last call to `fred-update`.

Arguments

view	A Fred window or Fred dialog item.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to be an entire point in encoded form and is returned unchanged.

`fred-hpos` [*Generic function*]

Syntax `fred-hpos` (*view* `fred-mixin`) `&optional` *position*

Description The `fred-hpos` generic function returns the horizontal position of the line containing *position*, in local window coordinates. The position is computed as the length (in pixels) of the line containing *position*, minus the amount of horizontal scrolling currently in effect in the window.

Arguments

view	A Fred window or Fred dialog item.
position	An integer position, or a mark in the window's buffer. The default value is the window's insertion point.

`fred-vpos` [*Generic function*]

Syntax `fred-vpos` (*view* `fred-mixin`) `&optional` *position*

Description The `fred-vpos` generic function returns the vertical position of the line containing *position*, in local window coordinates. If *position* is not visible in the window, `-1` is returned.

Arguments

view	A Fred window or Fred dialog item.
position	An integer position, or a mark in the window's buffer. The default value is the window's insertion point.

`fred-line-vpos` [*Generic function*]

Syntax `fred-line-vpos` (*view* `fred-mixin`) *line-number*

Description The `fred-line-vpos` generic function returns the vertical position of *line-number* in local window coordinates.

Arguments

view	A Fred window or Fred dialog item.
line-number	A line number.

`fred-hscroll` [*Generic function*]

Syntax `fred-hscroll` (*view* `fred-mixin`)

Description The `fred-hscroll` generic function returns the value of the desired amount of horizontal scroll in pixels in *view*.

Argument *view* A Fred window or Fred dialog item.

`set-fred-hscroll` [*Generic function*]

Syntax `set-fred-hscroll` (*view* `fred-mixin`) *hscroll*

Description The `set-fred-hscroll` generic function sets the value of the horizontal scroll in *view* to *hscroll*.

Arguments

view	A Fred window or Fred dialog item.
hscroll	The desired amount of horizontal scroll in pixels.

`selection-range` [*Generic function*]

Syntax `selection-range` (*view* `fred-mixin`)

Description The `selection-range` generic function returns two values specifying the beginning and end of the currently selected text. If no text is selected, the function returns the insertion point as both values. You can use `eql` to test whether text is currently selected. If the two values are `eql`, no text is selected.

Text selected in the active window is highlighted.

Argument *view* A Fred window or Fred dialog item.

`set-selection-range` [*Generic function*]

Syntax `set-selection-range` (*view* `fred-mixin`) `&optional` *position* *cursorpos*

Description The `set-selection-range` generic function sets the text currently selected to the buffer range between *position* and *cursorpos* and updates the window display. If *position* is equal to *cursorpos*, the selection range is made empty.

See also `select-all` on page 507.

Arguments

view	A Fred window or Fred dialog item.
position	An integer position, or a mark in the window's buffer.
cursorpos	A position in the window buffer, by default the insertion point.

`collapse-selection` [*Generic function*]

Syntax `collapse-selection` (*view* `fred-mixin`) *forward-p*

Description The `collapse-selection` generic function does nothing and returns `nil` if no text is selected. Otherwise, it deselects the selected text and returns `t`.

Arguments

view	A Fred window or Fred dialog item.
forward-p	An argument specifying the direction in which the cursor moves. If *forward-p* is `t`, the cursor moves forward to the end of the selected text; if it is `nil`, the cursor moves backward to the beginning of the selected text.

`get-back-color` [*Generic function*]

Syntax `get-back-color` (*window* `fred-window`)

Description The `get-back-color` generic function returns the background color of *window*, encoded as an integer.

Argument *window* A Fred window.

`get-fore-color` [*Generic function*]

Syntax `get-fore-color` (*window* `fred-window`)

Description The `get-fore-color` generic function returns the foreground color of *window*, encoded as an integer.

Argument *window* A Fred window.

`view-font` [*Generic function*]

Syntax `view-font` (*view* `fred-mixin`)

Description The `view-font` generic function returns three values: the font spec of the current insertion font, the font spec of the character at the insertion point (or the first character in the selection), and a Boolean value. The Boolean value is `t` if the entire selection and the insertion font use the same font spec; otherwise, the Boolean value is `nil`.

Argument *view* A Fred window or Fred dialog item.

---

set-view-font [*Generic function*]

Syntax `set-view-font` (*view* `fred-mixin`) *font-spec*

Description If text is selected, the `set-view-font` generic function merges the font specs of the characters in the selection with the given *font-spec*. If there are multiple font specs in the selection, they are all merged individually. If no text is selected, the current insertion font is merged with the given *font-spec*, and the result is used as the new insertion font.

Fred does not automatically set the insertion font when you move the insertion point. If the insertion font is `("Monaco" 9)`, typed characters appear in 9-point Monaco, even if you place the insertion point between characters displayed in Times Bold. When there is no selection, the insertion font must always be changed explicitly by a call to `set-view-font`. This behavior may change in future releases of Macintosh Common Lisp.

Arguments

view	A Fred window or Fred dialog item.
font-spec	A font spec.

---

view-font-codes [*Generic function*]

Syntax `view-font-codes` (*view* `fred-mixin`)
`view-font-codes` (*view* `fred-window`)

Description The function `view-font-codes` returns a font/face code and a mode/size code for the buffer associated with *view*. If the buffer is empty, the function returns the empty buffer font. If the next insertion font was specified, this function returns the next insertion font; otherwise, if there is a selection, the font neighbor rule determines the font/face code and the mode/size code returned, assuming the insertion point is at the first character of the selection; if there is not a selection, this function returns the font determined by the neighbor rule.

Arguments *view* An instance of `fred-mixin` or an instance of `fred-window`.

Example

```
? (view-font-codes (window-key-handler my-window))
1441792
65548
```

`set-view-font-codes` [*Generic function*]

Syntax

`set-view-font-codes (`*view* `fred-mixin) ff ms`
`set-view-font-codes (`*view* `fred-window) ff ms`

Description

The `set-view-font-codes` function sets the font for the buffer associated with *view*. If the buffer is empty, this function sets the empty buffer font. If the buffer is not empty, this function sets the next insertion font. This function is similar to the function `buffer-set-font-codes`. The difference is the specification of *view* rather than *buffer*.

If either *ff* or *ms* is nil, this function clears the next insertion font and the font neighbor rule determines the font for the next insertion.

Arguments

view	An instance of `fred-mixin` or an instance of `fred-window`.
ff	A font/face code. A font/face code is a 32-bit integer that combines the name of the font and its face (e.g., plain, bold, italic). For more information see "Functions related to font codes" on page 80.
ms	A mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (e.g., inclusive-or, exclusive-or, complemented) and the font size.

`ed-set-view-font` [*Generic function*]

Syntax

`ed-set-view-font (`*view* `fred-mixin)` *font-spec*

Description

The function `ed-set-view-font` modifies a font by applying a mask. If *view* has a selection, this function merges the current font of each character in the selection with *font-spec*. If *view* has no selection and the buffer is empty, this function merges the empty buffer font with *font-spec*.

If the buffer is not empty and `*last-command*` is a font-setting command, this function merges *font-spec* with the font specified by the previous command; otherwise, this function merges *font-spec* with the next insertion font. In either case, this function sets `fred-last-command` to `'(set-font new-ff new-ms)`.

Arguments

view	An instance of `fred-mixin`.
font-spec	A font specification.

Example

```
? (ed-set-view-font (window-key-handler my-window) '(:bold))
(:BOLD)
```

ed-view-font-codes

[*Generic function*]

Syntax `ed-view-font-codes` (*view* `fred-mixin`)

Description The function `ed-view-font-codes` returns the font codes of the next insertion font if `*last-command*` or `fred-last-command` specify a font. Otherwise, this function behaves the same as the function `view-font-codes`.

Arguments *view* An instance of `fred-mixin`.

window-set-not-modified

[*Generic function*]

Syntax `window-set-not-modified` (*view* `fred-mixin`)

Description The `window-set-not-modified` generic function is called by the window system when a Fred window is saved to a file. It sets the state of the window to be not modified and calls `fred-update`.

Argument *view* A Fred window or Fred dialog item.

window-filename

[*Generic function*]

Syntax `window-filename` (*window* `fred-window`)

Description The `window-filename` generic function returns the pathname of the file associated with the Fred window. If no pathname is associated with the window, `window-filename` returns `nil`.

Files become associated with Fred windows when `set-window-filename` is called or if the window was created with an initialization argument for `:filename`.

Argument *window* A Fred window.

set-window-filename

[*Generic function*]

Syntax `set-window-filename` (*window* `fred-window`) *new-name*

Description The `set-window-filename` generic function sets the filename associated with the Fred window. When the window contents are saved, they are saved to the new filename. If a file corresponding to the filename already exists, it is overwritten without warning when the window is next saved.

Arguments

window	A Fred window.
new-name	A pathname or string giving the file to associate with the window.

`fred-package` [*Generic function*]

Syntax `fred-package` (*view* `fred-mixin`)

Description The `fred-package` generic function returns the package associated with the window containing the item or `nil`. If `nil`, the window's package is always the current value of `*package*`.

Argument

view	A Fred window or Fred dialog item.

`set-fred-package` [*Generic function*]

Syntax `set-fred-package` (*view* `fred-mixin`) *package*

Description The `set-fred-package` generic function sets the package associated with the window containing the Fred dialog item. The *package* argument may be a package, or it may be a string or symbol naming a package.

Arguments

view	A Fred window or Fred dialog item.
package	A package indicator (that is, a string or a symbol naming a package, or a package object).

`fred-margin` [*Generic function*]

Syntax `fred-margin` (*view* `fred-mixin`)

Description The `fred-margin` generic function returns the distance in pixels between the left edge of *view* and the left edge of the first character on each line.

Argument

view	A Fred window or Fred dialog item.

`set-fred-margin` [*Generic function*]

Syntax `set-fred-margin` (*view* `fred-mixin`) *new-margin*

Description The `set-fred-margin` generic function sets the distance in pixels between the left edge of *view* and the left edge of the first character on each line to *new-margin*.

Arguments

view	A Fred window or Fred dialog item.
new-margin	A fixnum specifying the width of the margin in pixels. If *new-margin* is not a fixnum, an error is signaled.

`fred-tabcount` [*Generic function*]

Syntax `fred-tabcount` (*window* `fred-window`)

Description The `fred-tabcount` generic function returns the number of spaces per tab in *window*.

Argument *window* A Fred window.

`fred-wrap-p` [*Generic function*]

Syntax `fred-wrap-p` (*window* `fred-mixin`)

Description The `fred-wrap-p` generic function returns a Boolean value, `t` if lines wrap in *window*, `nil` if they do not.

Argument *window* A Fred window.

Example

The following code defines a Fred command that toggles whether text is wrapped in a window. The function `ed-refresh-screen` used in this example is defined in the file `assorted-fred-commands.lisp` in your MCL Examples folder.

```
(def-fred-command (:control :meta #\w)
  (lambda (w)
    (setf (fred-wrap-p w) (not (fred-wrap-p w)))
    (ed-refresh-screen w)))
```

`window-save` [*Generic function*]

Syntax `window-save` (*window* `fred-mixin`)

Description The `window-save` generic function saves the window to its disk file. If the window has no filename, `window-save-as` is called.

Argument *window* A Fred window.

`window-save-as` [*Generic function*]

Syntax `window-save-as` (*window* `fred-mixin`)

Description The `window-save-as` generic function calls the standard `SfPutFile` dialog box, allowing the user to choose a directory and input a filename, and saves the contents of the window to the filename.

Note that if the user clicks Cancel in this dialog box, Macintosh Common Lisp throws to `:cancel`. User code may wish to perform a `catch-cancel` to prevent a return to the top level. (The macro `catch-cancel` is documented on page 240)

Argument *window* A Fred window.

`window-revert` [*Generic function*]

Syntax `window-revert` (*window* `fred-mixin`) `&optional` *dont-prompt*

Description The `window-revert` generic function causes the window to revert to the last version saved.

Arguments

window A Fred window.

dont-prompt A Boolean value. If the value is `nil` (the default), the user is asked to confirm the reversion before it is performed. If the value of this parameter is true, the reversion is performed without asking the user.

`window-hardcopy` [*Generic function*]

Syntax `window-hardcopy` (*window* `fred-window`) `&optional` *show-dialog*

Description The `window-hardcopy` generic function sends the contents of the window or dialog item to the current printer. Before printing takes place, the user is prompted to specify various printer options.

To insert a hard page break in printouts, press Control-Q Control-L (the quoted form feed character). The character appears as a square box in MCL windows.

Arguments

window	A Fred window or Fred dialog item.
show-dialog	A Boolean value. If this value is true (the default), MCL presents a print job dialog to the user. Otherwise, MCL uses the values entered last time, or the default values if no print job dialog has been shown yet.

ed-beep [*Function*]

Syntax `ed-beep &rest` *the-rest*

Description The `ed-beep` function sounds a beep. It returns `nil`.

Argument

the-rest	A rest argument; ignored.

Functions implementing standard editing processes

The functions that are found in the standard Macintosh editing menu are implemented as MCL generic functions. In addition, Macintosh Common Lisp provides a more extensive Undo facility.

cut [*Generic function*]

copy [*Generic function*]

paste [*Generic function*]

clear [*Generic function*]

undo [*Generic function*]

undo-more [*Generic function*]

select-all [*Generic function*]

Syntax

`cut` (*view* `fred-mixin`)
`copy` (*view* `fred-mixin`)
`paste` (*view* `fred-mixin`)
`clear` (*view* `fred-mixin`)
`undo` (*view* `fred-mixin`)
`undo-more` (*view* `fred-mixin`)
`select-all` (*view* `fred-mixin`)

Description

These generic functions are each specialized on the `fred-mixin` class (as well as on `window`; see Chapter 4: Views and Windows).

The `cut` generic function deletes the currently selected text from the buffer and stores it in the Clipboard and the kill ring.

The `copy` generic function adds the currently selected text to the Clipboard and pushes it onto the kill ring. The selection is not removed from the window or dialog item.

The `paste` generic function replaces the currently selected text with the text in the Clipboard. If no text is selected, the text in the Clipboard is inserted at the insertion point.

The `clear` generic function deletes the currently selected text from the buffer without storing it in the Clipboard or the kill ring.

The `undo` generic function undoes the most recent edit if it can be undone.

The `undo-more` generic function undoes edits earlier in the edit history if they can be undone.

The `select-all` generic function makes the entire contents of the buffer the currently selected text.

Argument

view A Fred window or Fred dialog item.

window-can-do-operation [*Generic function*]

Syntax `window-can-do-operation` (*view* `fred-mixin`) *operation* `&optional` *menu-item*

Description The `window-can-do-operation` generic function is called to determine whether an Edit menu item should be enabled. It returns a Boolean value indicating whether *view* can perform *operation*. (This is a more general replacement for the older MCL function `window-can-undo-p`, which could check only for Undo.)

Arguments

view	A Fred window or Fred dialog item.
operation	A symbol indicating one of the standard editing operations: `cut`, `clear`, `copy`, `paste`, `select-all`, `undo`, or `undo-more`.
menu-item	The corresponding Edit menu item.

Example

The following code indicates that `*top-listener*` contains a selection that can be cut.

```
? (window-can-do-operation *top-listener* 'cut)
T
```

Multiple-level Undo

To support Undo, Fred buffers keep a history list of all changes that have been made since the buffer was created (or up to a user-definable limit imposed by `*fred-history-length*`). When an Undo command is issued, the most recent command on the history list is undone. Repeatedly issuing Undo commands undoes earlier and earlier changes, back to the initial state of the buffer or to the limit imposed by `*fred-history-length*`.

A single Fred command may involve several insertions and deletions. In the functions that relate to Undo, the argument *append-p* indicates that an insertion or a deletion is part of the same operation as the previous insertion or deletion.

Successive adjacent deletions or insertions, as well as multiple replacements via the Search dialog, are considered a single command.

This Undo history is maintained on a buffer-by-buffer basis. The number of commands saved for each Fred buffer is under user control.

Functions relating to Undo

The following functions support Undo in Macintosh Common Lisp. (See also “Undo commands” on page 60.)

`ed-delete-with-undo` [*Generic function*]

Syntax `ed-delete-with-undo` (*view* `fred-mixin`) *start end* `&optional` *save-p reverse-p append-p*

Description The `ed-delete-with-undo` generic function deletes the range in *view*’s buffer specified by *start* and *end* and saves the deletion on the kill ring and in the history list of the buffer of *view*. It returns a `cons` with the string of the deleted range in the `car` and the style vector of the deleted range in the `cdr`.

The `ed-delete-with-undo` generic function works with the `*last-command*` variable (defined on page 522) to concatenate successive deletions. If this function is called repeatedly, it concatenates the deleted text into a single item on the kill ring rather than creating several items on the kill ring. Deleting with `ed-delete-with-undo` sets the last command to `:kill`.

All Fred commands that delete text call `ed-delete-with-undo`.

Arguments

view	A Fred window or Fred dialog item.
start	The start of the range to delete. This argument may be an integer or a buffer mark.
end	The end of the range to delete. This argument may be an integer or a buffer mark.
save-p	An argument specifying what to do with deleted text. If the value of this argument is true (the default), the deleted text is added to the kill ring. If it is `nil`, the deleted text is added to the kill ring only if it is nontrivial. Nontrivial strings are those that contain both word-forming characters and word-delimiting characters. The justification is that text that is explicitly deleted should always be saved (therefore the value of *save-p* is true by default). Text killed incidentally, for instance, when the user types over selected text, is saved only when it is complicated.
reverse-p	An argument specifying the direction of the deletion. The value of this argument is true if the text is killed by a backward deletion.
append-p	An argument specifying whether this deletion is part of the same operation as the previous insertion, deletion, or replacement. If true, it is.

`ed-insert-with-undo` [*Function*]

Syntax `ed-insert-with-undo` *view string* `&optional` *position append-p*

Description The `ed-insert-with-undo` function inserts *string* in *view* at *position* and saves *string* in the history list of the buffer of *view*.
If *string* is to be appended to a previous Undo command, *append-p* is true.

Arguments

view	A Fred window or Fred dialog item.
string	Either a string, or a `cons` of a string and a style.
position	A position in the view. The default is the insertion point.
append-p	An argument specifying whether this insertion is part of the same operation as the previous insertion, deletion, or replacement. If true, it is.

`ed-replace-with-undo` [*Function*]

Syntax `ed-replace-with-undo` *view start end string* `&optional` *append-p*

Description The `ed-replace-with-undo` function replaces the range of characters from *start* to *end* in *view* with *string* and saves the replaced range in the history list of the buffer of *view*.

Arguments

view	A Fred window or Fred dialog item.
start	The start of the range to replace. This argument may be an integer or a buffer mark.
end	The end of the range to replace. This argument may be an integer or a buffer mark.
string	Either a string, or a `cons` of a string and a style.
append-p	An argument specifying whether this replacement is part of the same operation as the previous insertion, deletion, or replacement. If true, it is.

`set-fred-undo-string` [*Function*]

Syntax `set-fred-undo-string` *fred-window string* `&optional` *undo-redo*

Description The `set-fred-undo-string` function sets the suffix of the Undo menu title to *string* in *fred-window*. For example, if *string* is `"Typing"`, the Undo menu item title becomes Undo Typing or Redo Typing.

Arguments

fred-window	A Fred window.
string	A string.
undo-redo	A value, either `:UNDO` or `:REDO`. If it is `:REDO`, then the name of the Undo menu item is Redo *string* (for example, Redo Typing). If it is `:UNDO` (the default), the name of the Undo menu item is Undo *string*.

---

setup-undo [*Generic function*]

Syntax `setup-undo` (*view* `fred-mixin`) *function* `&optional` *string*

Description The `setup-undo` generic function allows Fred commands to support the Undo menu item. Any Fred action that can be undone in a way that is not supported by `ed-insert-with-undo`, `ed-replace-with-undo`, or `ed-delete-with-undo` should call `setup-undo`. The *function* argument should be a function to call when Undo is chosen. If given, *string* should be a short string to be used as the title of the menu item.

Arguments

view	A Fred window or Fred dialog item.
function	The function to call when the Undo menu item is chosen.
string	A string serving as the title of the Undo menu item. The default is “Undo”.

Examples

Here is an example of how to enable Undo and Redo. The function `insert-hello` inserts the string `"hello"`. It supports Undo and Redo.

```
? (defun insert-hello (window)
        (let* ((buf (fred-buffer window))
               (start-pos (buffer-position buf)))
          (buffer-insert buf "hello" start-pos)
          (fred-update window)
          (setup-undo window
                      #'(lambda
                          ()
                         (buffer-delete buf start-pos)
                         (fred-update window)
                         (setup-undo window
                          #'(lambda ()
```

```
                                   (insert-hello window)
                                   (fred-update window))
                                "Redo Hello"))
            "Undo Hello")))
INSERT-HELLO
```

This example shows how to do the same thing more simply.

```
? (defun alternative-insert-hello (window)
         (ed-insert-with-undo window "hello")
         (set-fred-undo-string window "Hello")
         (fred-update window))
ALTERNATIVE-INSERT-HELLO
```

---

setup-undo-with-args [*Generic function*]

Syntax `setup-undo-with-args` (*view* `fred-mixin`) *function arg* `&optional` *string*

Description The `setup-undo-with-args` generic function works like `setup-undo` except that the *function* must take two arguments; that is, it will be called via `(funcall` *function view arg*`)`. The argument is frequently a position.

Arguments

view	A Fred window or Fred dialog item.
function	The function to call when the Undo menu item is chosen.
arg	An argument to pass the function.
string	A string serving as a modifier to the title of the Undo menu item. For example, if the string is `"Typing"`, the Undo menu item title is Undo Typing or Redo Typing.

Working with the kill ring

The kill ring is a circular list containing text that has been deleted from Fred windows or dialog items. Each item in the kill ring is a `cons` cell. The `car` of the `cons` contains a string. The `cdr` of the `cons` contains either a style vector or `nil.` (Style vectors are described in “Using multiple fonts” on page 472.)

Functions for working with the kill ring

The following functions work with the kill ring.

`ed-kill-selection` [*Generic function*]

Syntax `ed-kill-selection` (*view* `fred-mixin`)

Description The `ed-kill-selection` generic function deletes the currently selected text by calling `ed-delete-with-undo`. The deleted text is saved only if it is nontrivial (see `ed-delete-with-undo` on page 509 for a definition of triviality).

Commands that insert text generally call this function before performing the insertion.

Argument *view* A Fred window or Fred dialog item.

`add-to-killed-strings` [*Function*]

Syntax `add-to-killed-strings` *string-style-cons*

Description The `add-to-killed-strings` function rotates the kill ring and pushes *string-style-cons* to the front of the kill ring.

Argument *string-style-cons*
A `cons` whose `car` is a string and whose `cdr` is a style vector or `nil`.

`rotate-killed-strings` [*Function*]

Syntax `rotate-killed-strings &optional` *count*

Description The `rotate-killed-strings` function rotates the kill ring. The third item becomes the second, the second item becomes the first, and the first becomes the last. (Remember, the kill ring is a circular list.)

Any empty items are automatically skipped.

Argument *count* An integer to be added to the normal number by which the kill ring is rotated. For example, if *count* is 0, the kill ring is rotated by 1; if it is 4, the kill ring is rotated by 5. The default value of *count* is 0.

Using the minibuffer

The minibuffer provides a convenient method for showing information to users of Fred windows. The information can be the result of a command, a progress indicator, a request for further information, or some combination of all these. All Fred windows display the window's package in the lower-left corner of the window. This is not considered part of the minibuffer.

Each instance of `fred-window` has its own minibuffer, an instance of the class `mini-buffer` that is accessed with `view-mini-buffer`. Minibuffers are output streams with some additional features.

The variable `*clear-mini-buffer*` specifies whether to clear the minibuffer after each Fred command.

Some of the following generic functions are associated with methods for Fred windows and some with methods for minibuffers.

Because minibuffers are streams, you can use them as the first argument to `format`. Sending a newline will clear the minibuffer.

Functions for working with the minibuffer

The following functions define minibuffers.

`mini-buffer` [*Class name*]

Description The `mini-buffer` class is the class of minibuffers, built on `output-stream`. A minibuffer displays information about its containing Fred window.

`view-mini-buffer` [*Generic function*]

Syntax `view-mini-buffer` (*view* `fred-mixin`)

Description The `view-mini-buffer` generic function returns the minibuffer of the window or dialog item.

Argument *view* A Fred window or Fred dialog item.

`set-mini-buffer` [*Generic function*]

Syntax `set-mini-buffer` (*view* `fred-mixin`) *string* `&rest` *format-args*

Description The `set-mini-buffer` generic function clears the text of the minibuffer, applies `#'format` to the minibuffer, *string*, and *format-args*, and then performs a minibuffer update to display the minibuffer.

The minibuffer shows only the last line printed. Sending a newline clears the minibuffer.

Arguments

view	A Fred window or Fred dialog item.
string	A format control string, suitable for passing as the second argument to `format`.
format-args	A set of *format* arguments, suitable for passing to `format` along with *format-string*.

`mini-buffer-update` [*Generic function*]

Syntax `mini-buffer-update` (*view* `fred-mixin`)

Description The `mini-buffer-update` generic function draws the contents of the minibuffer. This function is normally called whenever the window is updated. You need to call it explicitly whenever you print to a minibuffer and wish to show its contents.

Argument

view	A Fred window or Fred dialog item.

`stream-column` [*Generic function*]

Syntax `stream-column` (*stream* `mini-buffer`)

Description The `stream-column` generic function returns the length of the text displayed in the minibuffer.

Argument

stream	A minibuffer stream.

`mini-buffer-string` [*Generic function*]

Syntax `mini-buffer-string` (*minibuffer* `mini-buffer`)

Description The `mini-buffer-string` generic function returns the contents of the minibuffer as a string. This string is a vector with a fill pointer. The `stream-tyo` method on minibuffers works by pushing characters onto this string. Sending a newline to the minibuffer sets the fill pointer to 0. The `stream-column` function returns the value of the fill pointer.

Argument *minibuffer* A minibuffer.

Defining Fred commands

Besides the standard Fred commands, described in Chapter 1: Editing in Macintosh Common Lisp you can program your own commands. You may wish to create your own tables of commands (discussed in "Fred command tables" on page 517). The following macro defines a Fred command in the currently active command table.

def-fred-command [*Macro*]

Syntax `def-fred-command` *keystroke function* `&optional` *doc-string*

Description The `def-fred-command` macro is equivalent to `(comtab-set-key *comtab* 'keystroke 'function doc-string)`. (See the next section, "Fred Command Tables," for more information.)

Arguments

keystroke	A keystroke code or keystroke name.
function	A function to be called when *keystroke* is typed. Because this argument is not evaluated, it should usually be a symbol naming a function.
doc-string	A documentation string describing the action of *keystroke*.

Example

```
? (def-fred-command (:meta #\h) insert-hello)
#<A COMTAB>
```

Fred command tables

The following system is used to translate keystroke events into Fred actions.

Keystroke codes and keystroke names

This section describes the functions that allow the user to associate Lisp functions
with keystrokes.

In normal operation, keystrokes are handled by the active window. Fred treats every keystroke typed in a Fred window as a command. Associated with every possible keystroke is a Lisp function that implements the command. Some commands are simple; for example, pressing A is a command to insert A in the buffer. Some are more complex; for example, pressing Control-Meta-Shift-F is a command to select one s-expression forward. Fred makes no distinction between these two kinds of commands. Indeed, you can easily redefine A to perform a complicated series of actions.

When you press a key in a Fred window, Fred first translates it into a *keystroke code*. The keystroke code contains a character and four flags called *meta, control, function,* and *shift*. The keystroke is encoded as a small integer, with the character code in bits 0 through 7, the meta flag in bit 8, the control flag in bit 9, the function flag in bit 10, and the shift flag in bit 11. See Table 14-1.

■ **Table 14-1** Modifier bits in the keystroke code

Bit	ValueKeyword		Keyboard
8	#x100	`:meta`	Option
9	#x200	`:control`	Control or Command
10	#x400	`:function`	One of the 15 function keys
11	#x800	`:shift`	Shift

In Fred programming, keystrokes are usually described in terms of *keystroke names* rather than keystroke codes. A keystroke name is either a character or a list containing a character and zero or more modifier keywords. The modifier keywords are `:control`, `:shift`, `:meta`, and `:function`. Examples of legal keystroke names are `(:control #\a)`, `(:meta #\f)`, `(:function #\1)`, `#\a`, and `(:control :meta #\x)`.

The `:function` modifier is used to support the function keys on the Apple Extended Keyboard. The function keys are named using the characters `#\1` through `#\9` and `#\a` through `#\f`. For example, the F1 key has the keystroke name `(:function #\1)`, the F10 key has the name `(:function #\a)`, and the F15 key has the name `(:function #\f)`.

The functions `event-keystroke` and `keystroke-code` return the keystroke code.

The function `keystroke-code` can return any combination of character codes and modifier bits, but `event-keystroke` returns only a subset:

- The function bit is set only with character codes of 49–57 or 65–70 (the digits 1–9 and the letters A–F).
- The shift bit is set only with character codes representing non graphic characters or alphabetic characters combined with the Option or Control key.

You should note that the keystroke code for a letter with the shift bit set is not equal to the keystroke code for a shifted letter. That is:

```
? (keystroke-code '(#\A))
65
? (keystroke-code '(:shift #\A))
2113
```

Command tables

The binding between keystrokes and the functions they invoke is stored in a data structure called a *comtab* (short for *command table*). The global command table is stored in the variable `*comtab*`. Each window with `fred-mixin` may contain a local command table in a slot named `comtab`, the default value of which is `*comtab*`. In addition, each window may also contain a shadowing command table, which is initially `nil`.

Fred dispatch sequence

The `view-key-event-handler` method for `fred-mixin` performs the following sequence of events to process a keystroke.

When Fred receives a keyboard event, it binds the variable `*current-character*` to the character typed. It uses the function `event-keystroke` to translate the event to a keystroke code and binds `*current-keystroke*` to the keystroke code.

It then checks the variable `*fred-keystroke-hook*`, which can be a function, a command table, or `nil`. If it is a function, the function is run and is responsible for keystroke processing. If it is a command table, the keystroke is looked up in the command table. If it is `nil`, the keystroke is looked up in the `shadowing-comtab` or `comtab` of the Fred window or Fred dialog item. The keystroke look-up is performed by the generic function `keystroke-function`. If `*fred-keystroke-hook*` is `nil`, the keystroke is processed by the function `run-fred-command`.

When the function associated with the keystroke returns, Fred updates the display of the window on the screen, making sure the cursor is visible. (The function may set the variable `*show-cursor-p*` to `nil` to inhibit this.)

MCL expressions associated with keystrokes

The following functions control and report on the behavior of keystrokes.

`event-keystroke` [*Function*]

Syntax `event-keystroke` *message modifier*

Description The `event-keystroke` function takes the *message* and *modifier* fields of a Macintosh event record and returns a keystroke code. It sets the control bit if the Control key was pressed and the meta bit if the Option key was pressed. It sets the shift bit if the Shift key was pressed and the character either is not graphic or is alphabetic and the Control or Option key was also pressed. The Caps Lock key is ignored. The character portion of the keystroke code is set to the ASCII code in the *message* field, except when Option is pressed. If Option is pressed, the character portion is set to the character used to generate the keystroke (for instance, the code for Option-S is `#\s` rather than #\ß).

Fred calls this function when it receives a key-down event.

Arguments

message	The message field of an event record.
modifier	The modifier field of an event record.

`keystroke-name` [*Function*]

Syntax `keystroke-name` *keystroke-code*

Description The `keystroke-name` function returns the name of a keystroke code.

A keystroke name is either a character or a list, for example, `(:control :meta` *character*`)`, `(:function` *character*`)`, or `(:shift :meta` *character*`)`.

Argument *keystroke-code* Any valid keystroke code.

`keystroke-code` [*Function*]

Syntax `keystroke-code` *keystroke-name*

Description The `keystroke-code` function translates a keystroke name to a keystroke code.

Argument *keystroke-name*

A keystroke name. A keystroke name is either a character or a list, for example, `(:control :meta` *character*`)`, `(:function` *character*`)`, or `(:shift :meta` *character*`)`. The *keystroke-name* argument may also be a keystroke code (an integer), in which case it is simply returned.

Example

```
? (keystroke-code '(:shift :meta #\F))
2406
? (keystroke-name 2406)
(:SHIFT :META #\F)
```

---

keystroke-function [*Generic function*]

Syntax `keystroke-function` (*view* `fred-mixin`) *keystroke* `&optional` *comtab*

Description The `keystroke-function` generic function performs the full Fred command look-up for *keystroke*. It always returns a function or a command table, never `nil` or another keystroke.

It first looks up the keystroke in *comtab*, or if *comtab* is unspecified or `nil`, in the shadowing command table of *view*, if there is one; otherwise it looks in the command table of *view*. If the definition is another keystroke, that keystroke is looked up. Circularity will be detected.

Arguments

view	A Fred window or Fred dialog item.
keystroke	A keystroke name or keystroke code.
comtab	A command table.

---

fred-keystroke-hook [*Variable*]

Description The `*fred-keystroke-hook*` variable provides a hook into the Fred command dispatch process.

If this variable is a function, the function is called with one argument, the Fred window or dialog item, to do the keystroke processing. If it is a command table, the keystroke is looked up in the command table. If it is `nil`, the keystroke is looked up in the shadowing command table or command table of the Fred window or Fred dialog item.

The keystroke look-up is performed by the generic function `keystroke-function`.

`*last-command*` [*Variable*]

Description The `*last-command*` variable is bound by `run-fred-command` to the value saved by the last command. Thus if a Fred command does not set the last command with `set-fred-last-command`, the value of `*last-command*` is `nil` when the next command runs.

This information is useful when one command needs to know what the previous one was. For example, repeatedly calling `ed-yank-pop` (Meta-Y) inserts successive strings from the kill ring into a window. For this to work, each call to `ed-yank-pop` needs to know whether the last Fred command was also `ed-yank-pop`.

The user should never set `*last-command*` explicitly; use `set-fred-last-command` instead.

`fred-last-command` [*Generic function*]

Syntax `fred-last-command` (*view* `fred-mixin`)

Description The `fred-last-command` generic function returns the most recent Fred command.

Argument *view* A Fred window or Fred dialog item.

`set-fred-last-command` [*Generic function*]

Syntax `set-fred-last-command` (*view* `fred-mixin`) *new-last-command*

Description The generic function `set-fred-last-command` sets the last command of *view* to *new-last-command*.

Always use `set-fred-last-command` to set the value of `*last-command*`; do not set it directly.

Arguments *view* A Fred window or Fred dialog item.

new-last-command
A command to which to set `*last-command*`.

`*current-character*` [*Variable*]

Description The `*current-character*` variable is bound to the character of the current keystroke during the execution of Fred commands. This variable is used by functions such as `ed-self-insert`.

`*current-keystroke*` [*Variable*]

Description The `*current-keystroke*` variable is bound to the current keystroke during the execution of Fred commands.

`ed-self-insert` [*Generic function*]

Syntax `ed-self-insert` (*view* `fred-mixin`)

Description The `ed-self-insert` generic function inserts `*current-character*` into the window. You should call this function only from within Fred commands (at which time `*current-character*` is sure to be bound). The function `ed-self-insert` checks for a numeric prefix, such as Control-U, that tells it how many times to execute itself.

Argument *view* A Fred window or Fred dialog item.

MCL expressions relating to command tables

The following MCL expressions are used to create and govern command tables.

`*comtab*` [*Variable*]

Description The `*comtab*` variable is the global command table. You can modify this command table or set the variable to a new command table.

`*listener-comtab*` [*Variable*]

Description The `*listener-comtab*` variable contains the initial form of `comtab` for the class `listener`. Whenever a new Listener is created, its command table is initially the value of this variable.

By modifying `*listener-comtab*`, you can change the behavior of the Listener without affecting other Fred windows.

Note that setting this variable to a new command table (as opposed to modifying the command table it is set to) affects only those Listeners created after the change.

`*control-x-comtab*` [*Variable*]

Description The `*control-x-comtab*` variable contains the command table used by the Control-X keystroke.

`make-comtab` [*Function*]

Syntax `make-comtab &optional` *default*

Description The `make-comtab` function returns a command table, with *default* as the means to process all keystrokes.

Argument *default* A default value. If *default* is a command table, when a keystroke is looked up in the new command table, Macintosh Common Lisp looks in *default*. If *default* is `nil`, it defaults to the value of `*comtab*`. If it is anything else, for example, a function, *default* is expected to process a keystroke.

`copy-comtab` [*Function*]

Syntax `copy-comtab &optional` *source-comtab*

Description The `copy-comtab` function returns a new command table that is initially functionally equivalent to *source-comtab*.

Argument *source-comtab* A command table or `nil`. If *source-comtab* is specified as `nil`, it returns a copy of the command table that was in use when Macintosh Common Lisp was launched. This is useful if you have somehow corrupted the current command table.

---

comtabp [*Function*]

Syntax `comtabp` *thing*

Description The `comtabp` function returns true if *thing* is a command table; otherwise, it returns `nil`.

Argument *thing* Any Lisp object.

---

comtab-set-key [*Function*]

Syntax `comtab-set-key` *comtab keystroke function* `&optional` *doc-string*

Description The `comtab-set-key` function sets the definition of *keystroke* to *function* within *comtab*.

Arguments

comtab A command table.

keystroke A keystroke.

function A function to be called when *keystroke* is pressed. The function may be any of the following:

A symbol, a compiled function, or a lambda expression, indicating a function to call when *keystroke* is entered.

A command table, indicating that the keystroke is a prefix character, such as Control-X, that reads another character and looks it up in its own command table.

Another keystroke (name or code) to indicate that *keystroke* should do whatever the other keystroke would do.

The value `nil`, which causes the command table's default function to process the keystroke.

doc-string A documentation string describing the action of *keystroke*.

Example

For example, the following form binds the F12 key to a command that prints the date in the top window:

```
? (comtab-set-key *comtab* '(:function #\c)  ;F12 key
                 #'(lambda
                     (d)
                    (multiple-value-bind
                     (second minutes hour date month year)
                     (get-decoded-time)
                     (declare (ignore second minutes hour))
                     (format d "~a/~a/~a"
                      month
                      date
                      (- year 1900))))
   "print the date in the top window")
#<COMTAB #x2F9E99>
```

comtab-get-key [*Function*]

Syntax `comtab-get-key` *comtab keystroke*

Description The `comtab-get-key` function looks up the definition of *keystroke* in *comtab*. This function is the reverse of `comtab-set-key`.

Arguments

comtab	A command table. The value of *comtab* may be a symbol, a compiled function, a command table, another keystroke, or `nil`.
keystroke	A keystroke code or keystroke name.

Example

```
? (comtab-get-key *comtab* '(:meta #\h))
INSERT-HELLO
```

comtab-key-documentation [*Function*]

Syntax `comtab-key-documentation` *comtab keystroke*

Description The `comtab-key-documentation` function returns the documentation string associated with *keystroke*.

Arguments

comtab	A command table. The value of *comtab* may be a symbol, a compiled function, a command table, another keystroke, or `nil`.
keystroke	A keystroke code or keystroke name.

Example

```
? (comtab-key-documentation *comtab* '(:function #\c))
"print the date in the top window"
```

---

comtab-find-keys [*Function*]

Syntax `comtab-find-keys` *comtab function*

Description The `comtab-find-keys` function returns a list of all keystrokes bound to *function* in *comtab*.

Arguments

comtab	A command table. The value of *comtab* may be a symbol, a compiled function, a command table, another keystroke, or `nil`.
function	A function. The function symbol must be quoted (not `#"`).

Example

```
? (comtab-find-keys *comtab* 'ed-open-line)
(623)
```

Chapter 15:

Low-Level OS Interface

Contents

This chapter discusses basic information necessary for accessing the Macintosh Toolbox and OS, and other code written in other languages.

You should read this chapter if you are accessing Macintosh data structures at a low level. If you are using higher-level accessing, such as through records and traps, you should read this chapter for background information.

When making any operating system call, you should be familiar with its description in *Inside Macintosh*. The discussions in this chapter assume some familiarity with *Inside Macintosh*.

Many MCL programmers will not need to use the facilities described in this and the following chapter. The object-oriented interface to the Macintosh OS is generally much safer and easier to use, when it is sufficient.

Interfacing to the Macintosh

Macintosh Common Lisp provides two levels of interface to the Macintosh OS.

- At the higher level, you can access the Macintosh Toolbox and Macintosh Operating System through predefined MCL classes, such as `view` and `window`, and the methods that apply to them. These are easy and safe to use, insulating you from both syntactic and semantic errors associated with Macintosh data structures.
- At the lower level, you can directly call the majority of Macintosh OS entry points and use Macintosh record structures and constants.

This chapter describes general considerations when interfacing between MCL and code written in other languages, such as the code implementing the Macintosh OS. Details of accessing OS entry points are discussed in Chapter 16: OS Entry Points and Records.

In general, special care should be taken when calling outside of the Lisp world. Because most other languages provide much less error checking than MCL, calling code written in other languages has the possibility of crashing your Macintosh.

Sharing Data between MCL and the OS

Macintosh Common Lisp manipulates two distinct sets of data: Macintosh data, such as windows, patterns, and rectangles, and Lisp data, such as lists, symbols, and objects. Lisp data and Macintosh data are stored in different places and in different formats. Macintosh data is stored on the application heap, and Lisp data is stored on the Lisp heap. These two heaps operate independently. Each piece of data belongs on either one heap or the other.

Some Lisp data contains pointers to Macintosh data. For example, window objects (Lisp data) contain pointers to Macintosh window records (on the Macintosh heap). With isolated exceptions, Macintosh data should not contain pointers to Lisp objects.

In general, Macintosh data is needed only for communication with the Macintosh OS. Before MCL can pass data to the OS, the data must be coerced to a form that the OS can use. This data cannot be stored on the Lisp heap but instead must be stored on the application heap or on the stack.

Macptrs

A macptr (an object of type `macptr`) represents a 32-bit address.

Macptrs are generated in the following ways:

- By a call to a trap, such as `#_newHandle`, `#_newPtr`, or `#_newWindow` that allocates a new pointer or handle.
- By a call to `%get-ptr`, where you are referencing some memory location relative to some Macintosh pointer.
- By a call to `make-record.`
- By a call to `%int-to-ptr`.
- By the macros `%stack-block` or `rlet`.

They are required in the following circumstances:

- As the first argument to `%get-` and `%put-` functions.
- As the value of any parameter to an OS entry point passed in an address register.
- As the value of any parameter to an OS entry point that requires a pointer, record, handle, or array.

You cannot pass any other Lisp object to a function requiring a macptr. In particular, `nil` cannot be passed as a pointer to a Macintosh data structure. Instead, you pass the macptr which is the result of calling `%null-ptr`.

Two macptrs to the same address are `eql` but not necessarily `eq`. That is, the two pointers themselves may not be the same; the address they reference is the same. If both `x` and `y` point to `(%int-to-ptr 0)`, then

```
? (eq x y)
...undetermined...
? (eql x y)
T
```

The address to which a macptr points is not changed by garbage collection.

In general, performing an operation such as `%int-to-ptr` or `%get-ptr` results in the allocation of a new Lisp macptr object. However, the MCL compiler avoids allocating macptrs whenever possible.

Here is an example in which a macptr is not allocated.

```
? (defun peek-long (addr)
"Returns the contents of the longword at ADDR."
(%get-long (%int-to-ptr addr)))
PEEK-LONG
```

Since the result of `(%int-to-ptr addr)` is used directly by `%get-long`, the compiler does not need to allocate a macptr.

By taking advantage of the following in your code, you can reduce the incidental allocation of macptrs.:

- Addresses that are consumed directly by primitive operations do not allocate a macptr. Of the MCL low-level functions for reading and writing to memory locations, most avoid allocating macptrs. (See "Accessing memory" on page 534 and "Strings, pointers, and handles" on page 545.)
- Macptrs can be explicitly stack-allocated. When appropriate, you may use `dynamic-extent` declarations to indicate that the compiler may safely stack-allocate macptrs used in local contexts.
- Macptrs can be destructively modified with `%setf-macptr`.

`%setf-macptr` [*Function*]

Syntax `%setf-macptr` *macptr pointer*

Description The `%setf-macptr` function destructively modifies *macptr* so that it references the address referenced by *pointer*. The compiler open-codes this function.

Arguments

macptr	A macptr.
pointer	A macptr.

Memory management

Macintosh Common Lisp works in cooperation with the Macintosh Memory Manager. Thus you can use the traps `#_NewPtr` and `#_NewHandle` to allocate blocks of memory on the application heap. The Macintosh and Lisp heaps are dynamically resized to satisfy memory requests. This resizing sometimes triggers a garbage collection.

△ **Important** You are responsible for releasing memory allocated on the Macintosh heap. (You can do so with `#_DisposPtr` and `#_DisposHandle`.) The contents of this memory are not subject to garbage collection even if all pointers to the memory are lost. △

The `#_NewPtr` and `#_NewHandle` traps are automatically called by `make-record`, described in Chapter 16: OS Entry Points and Records. The `#_DisposPtr` and `#_DisposHandle` traps are automatically called by `dispose-record.`

Stack blocks

When you need a small amount of memory for temporary storage, it is often more convenient and more efficient to bypass the Macintosh Memory Manager. The `%stack–block` macro allows programs to allocate blocks of memory on the stack. Be very careful using `%stack–block`. When you exit the `%stack-block` form, all the memory allocated is reclaimed and so any remaining pointers to the stack block become invalid. The `%stack-block` form should be used only for well-defined temporary storage, for example, to set up rectangles or I/O parameter blocks to be passed to OS entry points, or to store *var* arguments temporarily.

%stack-block [*Macro*]

Syntax `%stack-block ({(`*symbol size*`)}`$^+$`) {`*form*`}`*

Description For each *symbol*/*size* pair, the `%stack-block` macro allocates a block of storage *size* bytes long and binds *symbol* to a macptr to the block. The *forms* are executed in the resulting environment.

The `%stack-block` macro is usually not used directly to stack-allocate Macintosh records. Instead, use the `rlet` macro described in Chapter 16: OS Entry Points and Records.”

The action of `%stack-block` is semantically equivalent to doing a `#_NewPtr`/ `#_DisposPtr` pair for each variable but is much more efficient. The bindings and the storage created by `%stack–block` have dynamic extent; they become invalid when the form is exited from.

If a storage block of indefinite extent is needed, `make-record` should be used instead; see Chapter 16: OS Entry Points and Records.”

If there is not enough room on the stack to allocate the requested memory, an error is signaled.

(The obsolete `%vstack-block` macro is now semantically equivalent to `%stack-block` and is provided for backward compatibility only.)

Arguments *symbol* Any symbol. This symbol is bound to the stack block for the duration of the call.

size	The maximum total size for an individual stack block form is 32K bytes. Every *size* must evaluate to a positive fixnum.
form	Zero or more forms, which are evaluated as the body. Declarations may appear at the head of the body.

Example

In this example, an 8-byte block is allocated on the stack. The memory is filled with the coordinates of a rectangle. A pointer to the block—and two additional words—are then passed to the OS entrypoint `#_FrameRoundRect.` When the window is redrawn, a rectangle with rounded corners appears at the given coordinates:

```
? (setq my-window (make-instance 'window))
#<WINDOW "Untitled" #x46E929>
? (defmethod view-draw-contents ((w (eql my-window)))
    (let ((oval-width 12)
          (oval-height 8))
      (%stack-block ((my-rect 8))
        (%put-word my-rect 12 0)
        (%put-word my-rect 40 2)
        (%put-word my-rect 32 4)
        (%put-word my-rect 80 6)
        (#_FrameRoundRect my-rect
                          oval-width oval-height))))
#<STANDARD-METHOD VIEW-DRAW-CONTENTS ((EQL #<WINDOW
"Untitled"
    #x46E929>))>
? (view-focus-and-draw-contents my-window)
NIL
```

Accessing memory

Once memory for a structure has been allocated, programs need methods for directly reading from and writing to the memory. Macintosh Common Lisp provides the following low-level functions for reading and writing to memory locations. While these functions give you direct access to memory locations, they do not give you structured access. For most purposes, the macros `pref`, `href`, and their corresponding setf macros will be more useful. These macros are described in Chapter 16: OS Entry Points and Records.

Each of the following functions takes *offset*, a fixnum, as an optional argument. No type-checking is performed on *offset*. It is sign-extended to 32 bits. Most calls to these functions are compiled inline for efficiency.

◆ *Note:* No error checking is performed on any of the following functions. Since their purpose is to let you read and write to the memory in unforeseen ways, they aren't designed to prevent serious programming errors, and it is possible to read and write to memory locations in ways that seriously affect your computer. For example, on a 68000 (but not a 68020) microprocessor, accessing a word at an odd memory address results in a fatal error. Writing to a nonexistent memory address results in a bus access error, and so on.

`%get-signed-byte` [*Function*]

Syntax `%get-signed-byte` *macptr* `&optional` *offset*

Description The `%get-signed-byte` function gets the byte (8 bits) at *macptr* + *offset* and returns it as a signed Lisp integer in the range –128 through 127. The compiler open-codes this function.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%get-unsigned-byte` [*Function*]

`%get-byte` [*Function*]

Syntax `%get-unsigned-byte` *macptr* `&optional` *offset*
`%get-byte` *macptr* `&optional` *offset*

Description These equivalent functions get the byte (8 bits) at *macptr* + *offset* and return it as an unsigned Lisp integer in the range 0 through 255. The compiler open-codes these functions.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%hget-byte` [*Function*]

Syntax `%hget-byte` *handle* `&optional` *offset*

Description The `%hget-byte` function accesses a handle, gets the byte (8 bits) at *handle* + *offset*, and returns it as an unsigned Lisp integer in the range 0 through 255. The compiler open-codes this function.

The expression `(%hget-byte` *handle offset*`)` is equivalent to `(%get-byte (%get-ptr` *handle*`)` *offset*`)`.

Arguments

handle A handle. This argument must be a macptr.

offset Window.

%hget-signed-byte [*Function*]

Syntax `%hget-signed-byte` *handle* `&optional` *offset*

Description The `%hget-byte` function accesses a handle, gets the byte (8 bits) at *handle* + *offset*, and returns it as a signed Lisp integer in the range –128 through 127. The compiler open-codes this function.

Arguments

handle A handle. This argument must be a macptr.

offset Window.

%get-signed-word [*Function*]

Syntax `%get-signed-word` *macptr* `&optional` *offset*

Description The `%get-signed-word` function gets the word (16 bits) at *macptr* + *offset*, sign-extends it, and returns it as a signed Lisp integer in the range –32,768 through 32,767. The compiler open-codes this function.

Arguments

macptr A macptr.

offset A fixnum used to offset the address specified by *macptr*.

%get-unsigned-word [*Function*]

%get-word [*Function*]

Syntax `%get-unsigned-word` *macptr* `&optional` *offset*
`%get-word` *macptr* `&optional` *offset*

Description These equivalent functions get the word (16 bits) at *macptr* + *offset* and return it as an unsigned Lisp integer in the range 0 through 65,535. The compiler open-codes these functions.

Arguments

macptr A macptr.

offset A fixnum used to offset the address specified by *macptr*.

`%hget-word` [*Function*]

Syntax `%hget-word` *handle* `&optional` *offset*

Description The `%hget-word` function accesses a handle, gets the word (16 bits) at *handle* + *offset*, and returns it as an unsigned Lisp integer in the range 0 through 65,535. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

`%hget-signed-word` [*Function*]

Syntax `%hget-signed-word` *handle* `&optional` *offset*

Description The `%hget-signed-word` function accesses a handle, gets the word (16 bits) at *handle* + *offset*, sign-extends it, and returns it as a signed Lisp integer in the range –32,768 through 32,767. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

`%get-long` [*Function*]
`%get-signed-long` [*Function*]

Syntax `%get-long` *macptr* `&optional` *offset*
`%get-signed-long` *macptr* `&optional` *offset*

Description These equivalent functions get the macptr at *macptr* + *offset* and return a signed Lisp integer in the range – 2,147,483,648 through 2,147,483,647. The compiler open-codes these functions.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%hget-long` [*Function*]
`%hget-signed-long` [*Function*]

Syntax `%hget-long` *handle* `&optional` *offset*
`%hget-signed-long` *handle* `&optional` *offset*

Description These functions access a handle, get the macptr at *handle* + *offset*, and return a signed Lisp integer in the range –2,147,483,648 through 2,147,483,647. The compiler open-codes these functions.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

`%get-unsigned-long` [*Function*]

Syntax `%get-unsigned-long` *macptr* `&optional` *offset*

Description The `%get-unsigned-long` function gets the macptr at *macptr* + *offset* and returns an unsigned Lisp integer in the range 0 through 4,294,967,295. The compiler open-codes this function.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%hget-unsigned-long` [*Function*]

Syntax `%hget-unsigned-long` *macptr* `&optional` *offset*

Description The `%hget-unsigned-long` function gets the macptr at *macptr* + *offset* and returns an unsigned Lisp integer in the range 0 through 4,294,967,295. The compiler open-codes this function.

Arguments

macptr	A handle.
offset	Window.

`%get-ptr` [*Function*]

Syntax `%get-ptr` *macptr* `&optional` *offset*

Description The `%get-ptr` function returns the macptr at *macptr* + *offset*. The compiler open-codes this function. The resulting macptr is heap-consed unless it is used by another open-coded low-level primitive or used as the initial binding of a variable that is declared to have dynamic extent.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%hget-ptr` [*Function*]

Syntax `%hget-ptr` *handle* `&optional` *offset*

Description The `%hget-ptr` function accesses a handle and returns the macptr at *handle* + *offset*. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

`%get-string` [*Function*]

Syntax `%get-string` *macptr* `&optional` *offset*

Description The `%get-string` function gets the Pascal string at *macptr* + *offset* and returns it as a Lisp string. This function is not open-coded by the compiler. If *macptr* points to a handle on the Macintosh heap, the handle is dereferenced to access the string.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%get-cstring` [*Function*]

Syntax `%get-cstring` *macptr* `&optional` *offset end*

Description The `%get-cstring` function gets *string* at *macptr* + *offset* and returns it as a Lisp string. This function is not open-coded by the compiler. If *macptr* points to a handle on the Macintosh heap, the handle is dereferenced to access the string.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*. The default is 0.
end	The end of the string to be gotten. The default is *offset*.

`%get-ostype` [*Function*]

Syntax `%get-ostype` *macptr* `&optional` *offset*

Description The `%get-ostype` function gets the 4 bytes at *macptr* + *offset* and returns them as an os-type, a keyword of four characters. (See *Inside Macintosh* for details on os-types.) It returns `nil`.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%get-double-float` [*Function*]

`%get-single-float` [*Function*]

Syntax

`%get-double-float` *macptr* `&optional` *offset target*
`%get-single-float` *macptr* `&optional` *offset target*

Description

These functions gets the macptr at *macptr* + *offset* and return it as a floating point number. If *target* is specified, it should be a double-float. It will be descructively modified to hold the result. (In the case of `%get-single-float`, the number will be converted from single-float to double-float format.)

If you use the *target* argument, take precautions that you do not modify in place a floating point number that is used elsewhere in your program. Take particular care not to modify the system wide unique `0.0` instance. It is safest to create a fresh target locally, for example by the call `(ccl::%copy-float 0.0)`.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.
target	A floating point number.

`%hget-double-float` [*Function*]

`%hget-single-float` [*Function*]

Syntax

`%hget-double-float` *handle* `&optional` *offset target*
`%hget-single-float` *handle* `&optional` *offset target*

Description

These functions access a handle, get the macptr at *handle* + *offset* and return it as a floating point number. If *target* is specified, it should be a double-float. It will be descructively modified to hold the result. (In the case of `%hget-single-float`, the number will be converted from single-float to double-float format.)

If you use the *target* argument, take precautions that you do not modify in place a floating point number that is used elsewhere in your program. Take particular care not to modify the system wide unique `0.0` instance. It is safest to create a fresh target locally, for example by the call `(ccl::%copy-float 0.0)`.

Arguments

handle	A macptr which is a handle.
offset	A fixnum used to offset the address specified by *macptr*.

target	A floating point number.

`%put-byte` [*Function*]

Syntax `%put-byte` *macptr data* `&optional` *offset*

Description The `%put-byte` function stores the low 8 bits of *data* at *macptr* + *offset*. It returns `nil`. The compiler open-codes this function.

Arguments

macptr	A macptr.
data	A fixnum. The low 8 bits are used. This argument is not type-checked.
offset	A fixnum used to offset the address specified by *macptr*.

`%hput-byte` [*Function*]

Syntax `%hput-byte` *handle data* `&optional` *offset*

Description The `%hput-byte` function accesses a handle and stores the low 8 bits of *data* at *handle* + *offset*. It returns `nil`. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
data	A fixnum. The low 8 bits are used.
offset	A fixnum used to offset the address specified by *macptr*.

`%put-word` [*Function*]

Syntax `%put-word` *macptr data* `&optional` *offset*

Description The `%put-word` function stores the low 16 bits of *data* at *macptr* + *offset*. It returns `nil`. The compiler open-codes this function.

Arguments

macptr	A macptr.
data	A fixnum. The low 16 bits are used. This argument is not type-checked.
offset	A fixnum used to offset the address specified by *macptr*.

`%hput-word` [*Function*]

Syntax `%hput-word` *handle data* `&optional` *offset*

Description The `%hput-word` function accesses a handle and stores the low 16 bits of *data* at *handle* + *offset*. It returns `nil`. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
data	A fixnum. The low 16 bits are used.
offset	A fixnum used to offset the address specified by *handle*.

`%put-long` [*Function*]

`%put-full-long` [*Function*]

Syntax `%put-long` *macptr data* `&optional` *offset*
`%put-full-long` *macptr data* `&optional` *offset*

Description These functions store the integer value of *data* at *macptr* + *offset*. (The function `%put-full-long` is included for backward compatibility; both functions allow full 32-bit accuracy.) The compiler open-codes these functions.

Arguments

macptr	A macptr.
data	Any Lisp object that can be coerced to a 32-bit immediate quantity.
offset	A fixnum used to offset the address specified by *macptr*.

`%hput-long` [*Function*]

Syntax `%hput-long` *handle data* `&optional` *offset*

Description The `%hput-long` function accesses a handle and stores the integer value of *data* at *handle* + *offset*. The compiler open-codes this function.

Arguments

handle	A dereferenced handle. This argument must be a macptr.
data	Any Lisp object that can be coerced to a 32-bit immediate quantity.
offset	A fixnum used to offset the address specified by *handle*.

`%put-ptr` [*Function*]

Syntax `%put-ptr` *macptr data* `&optional` *offset*

Description The `%put-ptr` function stores *data* as a macptr at *macptr* + *offset* and returns `nil`. The compiler open-codes this function.

Arguments

macptr	A macptr.

data	Another macptr.
offset	A fixnum used to offset the address specified by *macptr*.

`%hput-ptr` [*Function*]

Syntax `%hput-ptr` *handle data* `&optional` *offset*

Description The `%hput-ptr` function accesses a handle, stores *data* as a macptr at *handle* + *offset*, and returns `nil`. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
data	Another macptr.
offset	A fixnum used to offset the address specified by *handle*.

`%put-string` [*Function*]

Syntax `%put-string` *macptr string* `&optional` *offset maxsize*

Description The `%put-string` function stores *string* as a Pascal string starting at *macptr* + *offset*. The compiler does not open-code this function.

Arguments

macptr	A macptr.
string	A string.
offset	A fixnum used to offset the address specified by *macptr*. The default is 0.
maxsize	The maximum size of the string. The default is 255.

`%put-cstring` [*Function*]

Syntax `%put-cstring` *macptr string* `&optional` *offset maxsize*

Description The `%put-cstring` function stores *string* starting at *macptr* + *offset*. The compiler open-codes this function.

Arguments

macptr	A macptr.
string	A string.
offset	A fixnum used to offset the address specified by *macptr*.
maxsize	The maximum allowable size of the string. If the length of *string* is larger than *maxsize*, an error is signaled.

`%put-ostype` [*Function*]

Syntax `%put-ostype` *macptr string* `&optional` *offset*

Description The `%put-ostype` function stores *string* as 4 bytes at *macptr* + *offset*. The argument *string* may be anything that can be coerced to a 32-bit value—in other words, `%put-ostype` is another name for `%put-long.` The compiler open-codes this function.

Arguments

macptr	A macptr.
string	A string with a length of four characters, used to specify the os-type, or a symbol with a four-character `symbol-name`.
offset	A fixnum used to offset the address specified by *macptr*.

`%put-double-float` [*Function*]

`%put-single-float` [*Function*]

Syntax `%put-double-float` *macptr float* `&optional` *offset*
`%put-single-float` *macptr float* `&optional` *offset*

Description These functions store *float* at *macptr* + *offset*. (In the case of `%put-single-float`, *float* will be converted from double-float to single-float format.)

Arguments

macptr	A macptr.
float	A floating point number.
offset	A fixnum used to offset the address specified by *macptr*.

`%hput-double-float` [*Function*]

`%hput-single-float` [*Function*]

Syntax `%hput-double-float` *handle float* `&optional` *offset*
`%hput-single-float` *handle float* `&optional` *offset*

Description These functions store *float* at *handle* + *offset*. (In the case of `%hput-single-float`, the number will be converted from double-float to single-float format.)

Arguments

handle	A macptr which is a handle.
float	A floating point number.
offset	A fixnum used to offset the address specified by *macptr*.

Miscellaneous routines

The following routines provide additional tools and information for communicating between Macintosh Common Lisp and code written in other languages, such as the Macintosh OS. These include a set of functions for performing various types of data conversion and handle dereferencing, information on Pascal VAR arguments and Boolean values, and the definitions of Lisp functions that can be called by ROM routines.

Strings, pointers, and handles

The following utilities are provided for using strings, pointers, and handles.

`with-pstrs` [*Macro*]

`with-cstrs` [*Macro*]

`with-returned-pstrs` [*Macro*]

Syntax

`with-pstrs ({(`*symbol string* `&optional` *start end*`)}`$^+$`)` {*form*}$^+$
`with-cstrs ({(`*symbol string* `&optional` *start end*`)}`$^+$`)` {*form*}$^+$
`with-returned-pstrs ({(`*symbol string* `&optional` *start end*`)}`$^+$`)` {*form*}$^+$

Description

The `with-pstrs` macro allocates memory for each *string*, stores *string* in this memory in the Pascal string format, and binds the corresponding *symbol* to a pointer to the memory.

The `with-pstrs` macro saves the trouble of allocating memory and filling it with individual characters every time a Macintosh-accessible string is needed.

The `with-cstrs` macro allocates memory for each *string*, stores *string* in this memory, and binds the corresponding *symbol* to a pointer to the memory.

The `with-returned-pstrs` macro allocates 256 bytes for each string (rather than trying to optimize the amount of memory allocated). This guarantees that the returned string does not overwrite other segments of memory.

Traps that use the strings as VAR arguments for returning values should be called through the macro `with-returned-pstrs`.

Arguments

symbol A symbol that is bound to the Pascal string for the duration of the macro.

string	A string with a maximum length of 255 characters.
start	The position at which to begin reading characters from the string.
end	The position at which to stop reading characters from the string.
form	Zero or more forms that make up the body of the macro.

`macptrp` [*Function*]

Syntax `macptrp` *thing*

Description The `macptrp` function returns `t` if and only if *thing* is a macptr; otherwise, it returns `nil`.

Argument *thing* Any Lisp data object.

`pointerp` [*Function*]

Syntax `pointerp` *macptr* `&optional` *errorp*

Description The `pointerp` function returns `t` if and only if the address referenced by *macptr* is a valid address; otherwise, it returns `nil`. If *macptr* is not a macptr, an error is signaled.

Arguments

macptr	A macptr.
errorp	An argument specifying how to treat errors. If *errorp* is true, `pointerp` signals an error if *macptr* is not a macptr. Otherwise it returns `nil`.

`zone-pointerp` [*Function*]

Syntax `zone-pointerp` *thing*

Description The `zone-pointerp` function returns `t` if *thing* is a pointer to a nonrelocatable system or application-heap-zone memory block; otherwise, it returns `nil`.

The test is performed heuristically by determining if *thing* points to a Macintosh heap zone, and if so, determining if the longword before the address pointed at by *thing* is equal to the heap zone. A zone pointer is different from a generic pointer because the former points to a memory block that was allocated using `#_NewPtr`, and because the various Memory Manager pointer traps, such as `#_DisposPtr`, may be used on it.

For more information on the structure of zone pointers, see the information on the Memory Manager in *Inside Macintosh.*

Argument *thing* Any Lisp data object.

`handlep` [*Function*]

Syntax `handlep` *thing*

Description The `handlep` function returns `t` if *thing* is a Macintosh handle; otherwise, it returns `nil`.

This is determined heuristically by checking to see whether *thing* points to the system or application heap zone, and if so, indirecting through *thing* to determine if the longword prior to the address plus the zone pointer are equal to *thing*.

For more information on the structure of handles, see the information on the Memory Manager in *Inside Macintosh.*

Argument *thing* Any Lisp data object.

`with-dereferenced-handles` [*Macro*]

Syntax `with-dereferenced-handles ({(`*variable handle*`)}`$^{+}$`)` `{`*form*`}`$^{+}$

Description The `with-dereferenced-handles` macro executes *forms* with each *variable* bound to the locked, dereferenced *handle*.

Only previously unlocked handles are locked. Upon exit, only handles that were unlocked on entry are unlocked. (This prevents a bug that occurs when programming the Macintosh computer with other languages.) Unlocking of handles is protected against with `unwind-protect`, so the handles are guaranteed to be left in the same state before and after the call to `with-dereferenced-handles`, even if termination is abnormal.

Arguments

variable	A Lisp variable.
handle	A Macintosh handle.
form	A set of forms that are evaluated sequentially in the environment in which the handles are dereferenced and bound to variables.

`with-pointers` [*Macro*]

Syntax `with-pointers ({(`*variable pointer-or-handle*`)}`$^+$`) {`*form*`}`$^+$

Description The `with-pointers` macro binds *variable* to the pointer for each *pointer-or-handle* that is a pointer and binds *variable* to the locked, dereferenced handle for each *pointer-or-handle* that is a handle.

When binding handles, `with-pointers` acts just as `with-dereferenced-handles` does.

The `with-pointers` macro is useful if you are unsure whether *pointer-or-handle* will be a pointer or a handle. It signals an error if *pointer-or-handle* is neither a pointer nor a handle.

Arguments

variable A Lisp variable.

pointer-or-handle
A Macintosh pointer or handle.

form One or more forms that are evaluated in the resulting environment.

`%inc-ptr` [*Function*]

Syntax `%inc-ptr` *point* `&optional` *number*

Description The `%inc-ptr` function increments (or decrements) *pointer* by adding *number* to it and returns a new pointer. The compiler open-codes this function.

Arguments

pointer A macptr.

number An integer. The default is 1.

`%incf-ptr` [*Macro*]

Syntax `%incf-ptr` *pointer number*

Description The `%incf-ptr` function destructively modifies *pointer* by adding *number* to it and returns the modified *pointer*. The compiler open-codes this function.

Arguments

pointer A mactptr.

number An integer.

`%ptr-to-int` [*Function*]

Syntax `%ptr-to-int` *pointer*

Description The `%ptr-to-int` function returns an integer coerced from *pointer*, that is, the numerical address *pointer* points to. The compiler open-codes this function. This function may return a bignum.

Argument *pointer* A macptr.

`%int-to-ptr` [*Function*]

Syntax `%int-to-ptr` *integer*

Description The `%int-to-ptr` function returns a macptr coerced from *integer*, that is, a pointer to the numerical address *integer*. The compiler open-codes this function.

Argument *integer* An integer.

Pascal VAR arguments

Pascal VAR arguments are passed by reference rather than by value; that is, you pass the function a pointer to a piece of data rather than the data itself. The called function may then affect the data and in this way communicate information to the caller. Implementing VAR arguments in Macintosh Common Lisp is very easy. Just allocate memory of the appropriate size for the piece of data (either on the stack or on the Macintosh heap, depending on how long you want to use that piece of data) and pass the macptr as the VAR argument.

The Pascal null pointer

The following two MCL expressions are used to work with the Pascal null pointer.

`%null-ptr` [*Macro*]

Syntax `%null-ptr`

Description The result of `(%null-ptr)` is equivalent to the Pascal null pointer and to `(%int-to-ptr 0)`.

`%null-ptr-p` [*Function*]

Syntax `%null-ptr-p` *pointer*

Description The `%null-ptr-p` function returns `t` if *pointer* is a Pascal null pointer.

Argument *pointer* A pointer.

Callbacks to Lisp from the OS and other code

The following macros define Lisp functions that can be passed to the OS or to other C or Pascal code. This lets the other code make callbacks to Lisp.

`defpascal` [*Macro*]

`defccallable` [*Macro*]

Syntax

`defpascal` *name* `({`*type parameter* `}*` `[` *return-type*`])` `[`*doc-string* `]` `{`*form*`}*`

`defpascal` *name* `(:reg` *parameter*`)` `{`*form*`}*`

`defccallable` *name* `({`*type parameter* `}*` `[`*return-type*`])` `[`*doc-string*`]` `{`*form*`}*`

Description The syntax of these macros is similar to that of `defun`, except that the lambda list contains alternating types and parameter names and ends with a type specifier for the value returned by the procedure. The `&optional`, `&keyword`, and `&rest` keywords are not permitted, because they are not supported by the Pascal- or C-calling sequences.

The value cell of *name* is set to a pointer to the procedure. This pointer may be passed to OS entry points or to C or Pascal code that expects a pointer to a function.

Arguments

name A symbol to name the function.

type The type of the corresponding parameter. Arguments passed to the function are taken from the stack and coerced according to this type-specifier keyword. The following values are legal:

`:word` The argument is a 16-bit fixnum.

`:long`	The argument is a signed integer.
`:ptr`	The argument is a macptr.
parameter	The name of the parameter.
return-type	The type of the value returned by the function. The following values are legal:
`:word`	The returned value is a 16-bit fixnum.
`:long`	The returned value is interpreted as a 32-bit signed integer and returned as a signed MCL integer.
`:ptr`	The returned value is a macptr.
`:void`	The procedure does not return a value. (Omitting *return-type* is equivalent to a *return-type* of `:void.`)
doc-string	A documentation string.
form	The body of the function; zero or more forms that are evaluated as an implicit `progn` procedure.

Example

The following example is a simplified version of the `scroll-bar-proc` procedure from the file `scroll-bar-dialog-items.lisp` in the Library folder. The trap `#_TrackControl` is documented in *Inside Macintosh.*

```
? (defpascal my-track-proc (:ptr my-control
                                :word partCode
                                :void)
    (#_SetCtlValue my-control
     (+ (#_GetCtlValue my-control)
        (case partCode
          (20 -1)                   ;scroll back one line
          (21 1)                    ;scroll forward one line
          (22 (- *page-height*))    ;scroll back one page
          (23 *page-height*)))))    ;scroll forward one page
```

It could be used as follows:

```
(if
  (#_TrackControl the-control mouse-point my-track-proc)
    (view-draw-contents w))
```

In the second calling sequence for `defpascal`, only a single parameter is given. When the function is called, the values of the CPU registers are copied into a record. The values of the record can be set and accessed with `rref` and `rset` (for details, see Chapter 16: OS Entry Points and Records). The value returned by the function is the pointer to the record.

An example of the use of `defccallable` is given in Chapter 17: Foreign Function Interface

Defpascal and Interrupts

In general, `defpascal` callbacks occur without interrupts. In MCL 4.0 there is an option to enable interrupts during the callback. Usually, you will want the default `without-interrupts` behavior for callbacks from OS entry points, but if you write your own C, Pascal, etc. code that calls back to MCL, you may wish the callbacks to enable interrupts so that other MCL processes and event processing can get time during the callback. The new feature is specified with a `:without-interrupts` argument keyword, the argument for which is evaluated at load time. For example, the following callbacks will execute without interrupts:

```
(defpascal uninterruptable (:word x :word)
  x)
(defpascal uninterruptable (:word x
                            :without-interrupts t
                            :word)
  x)
```

The following callback will execute with interrupts enabled:

```
(defpascal interruptable (:word x
                          :without-interrupts nil
                          :word)
  x)
```

Chapter 16:

OS Entry Points and Records

Contents

This chapter discusses how to make calls to Macintosh OS entry points, and how to work with Macintosh data structured as Pascal records.

You should read this chapter if you are using records or OS entry points. OS entry points discussed in this chapter are those documented in *Inside Macintosh*.

You should be familiar with Chapter 15: Low-Level OS Interface before reading this chapter.

Entry Points and Records

OS Entry Points call procedures in the Macintosh OS, as defined and discussed in *Inside Macintosh.* These entry points often require the allocation of **records,** areas of Macintosh memory usually stored as Pascal records. For example, the procedure to draw and fill an oval in a window requires calling an entry point—calling a Macintosh OS procedure that knows how to draw and fill an oval. That entry point in turn requires an area of memory, a record, to store the rectangle that defines the filled oval.

MCL functions and programs work efficiently with entry points and records. You can easily access and alter data structures created at run time (such as windows and event records) as well as Macintosh resources.

◆ Note: On 68K-based Macintoshes, most OS entry points are implemented as trap instructions. On PowerPC-based Macintoshes, they are implemented as shared library entry points. For the most part, this documentation uses the terms “trap” and “entry point” interchangeably.

◆ Note: Code that calls external functions needs to be compiled if it is to run in an application with the compiler excised. Attempting to intrepret such functions will invoke the compiler, and error if the compiler is not present.

References to entry points and records

Every Macintosh OS entry point, constant, record, and record field type is now described in an interface file located in the Interfaces folder within the Library folder. When the value of `*autoload-traps*` is true, information is automatically read from the relevant interface file when the MCL reader encounters a reference to one of these values. The definition provided by the interface file describes the arguments and return values of entry points and can do relevant error checking.

Names of record formats and entry points in MCL correspond to those in MPW and *Inside Macintosh*. In MCL, however, calls to the operating system are indicated by the reader macros number sign–underscore (`#_`) or number sign–dollar sign (`#$`). (See “Loading and Calling Entry Points” on page 556.)

You can define your own record formats with `defrecord` and your own trap calls with `deftrap`. These macros are also described in this chapter.

Loading and Calling Entry Points

The "Interfaces" folder located in the "Library" folder contains source files giving definitions of over 2000 Macintosh entry points and records described in *Inside Macintosh*.

If you reference a known Macintosh entry point or record and the value of `*autoload-traps*` is true (the default), the trap is automatically loaded. If you want to load an entire interface file, use the function `require-interface`:

```
? (require-interface 'quickdraw)
"QUICKDRAW"
```

If you use an unusual selection of traps, you can create your own interface files, containing only those traps you use.

◆ *Note:* The autoloading mechanism uses the index files in the folder `ccl:library;interfaces;index;` to find trap and constant definitions. If you modify any of the interface files, you must execute the form `(reindex-interfaces)` to update the index files.

The format of interface files is slightly different in MCL 3.1 and MCL 4.0. Therefore, each one has its own "Interfaces" folder.

Calling entry points

#_symbol [*Reader macro*]

Syntax *#_symbol*

Description If the value of `*autoload-traps*` is true, the #_ reader macro tries to load the trap definition of *symbol* from the appropriate interface file and interns *_symbol* in the `traps` package. For example, `#_NewPtr` loads `_NewPtr` and interns the symbol `_NewPtr` in the `traps` package.

A call to the entry point is then compiled, according to the definition loaded from the interface file. The arguments to the entry point are defined in inside Macintosh.

Error checking is supported by the `:errchk` keyword. If the first argument to a entry point call is the keyword `:errchk`, then the entry point call will include a call to `ResError` or `MemError` when appropriate, and will otherwise check for a non-zero return value from the system call. Before using this option, check the entry point definition to make sure it supports this error reporting mechanism.

#$symbol [*Reader macro*]

Syntax #$*symbol*

Description If the value of `*autoload-traps*` is true, the #$ reader macro tries to load a constant definition for *symbol* from the proper interface file. It also interns $*symbol* in the `traps` package. For example, `#$WMgrPort` loads the constant `$WMgrPort` and interns the symbol `$WMgrPort` in the `traps` package.

Since #_ and #$ are reader macros, they are evaluated only at read time. Therefore, if you include autoloaded symbols in a macroexpansion, you must use `require-trap` or `require-trap-constant`.

`require-trap` [*Macro*]

Syntax `require-trap` *trap-name* `&rest` *rest*

Description The `require-trap` macro autoloads a trap whether called at read time or at macroexpand time.

Arguments

trap-name	A trap name.
rest	A list of other arguments.

Example

```
? (defmacro draw-string (string start end)
    (let ((pstr (gensym)))
      `(with-pstr (,pstr ,string ,start ,end)
         (require-trap #_DrawString ,pstr))))
DRAW-STRING
```

`require-trap-constant` [*Macro*]

Syntax `require-trap-constant` *trap-name*

Description The `require-trap-constant` macro autoloads a constant from the interface file, whether called at read time or at macroexpand time.

Argument *trap-name* The name of a trap constant.

`reindex-interfaces` [*Function*]

Syntax `reindex-interfaces`

Description The `reindex-interfaces` function updates the interface index files.

Traps in MCL 3.1

On the 68K Macintosh and in MCL 3.1, the Toolbox and operating system reside in ROM. However, to allow flexibility for future development, application code must be kept free of specific ROM addresses, so all references to Toolbox and operating system routines must be made indirectly through a **trap dispatch table.** To issue a call in assembly language to the Toolbox or operating system, you use a **trap macro** defined in a set of macro files.

When you assemble your program, the macro generates a **trap word.** Instruction words beginning with `$A` do not correspond to valid machine-language instructions. Instead they augment the MC68000 microprocessor's native instruction set with additional operations specific to the Macintosh computer.

An attempt to execute any instruction word beginning with `$A` causes a trap to the trap dispatcher, which determines what operation the trap word stands for, looks up the address of the corresponding routine in the trap dispatch table, and jumps to the routine.

Shared Library Entry Points in MCL 4.0

The vast majority of system calls that were "traps" on the 68K (unimplemented 68K instructions whose 16-bit opcode was #*x*A*xxx*) are entry points in some shared library on Power Macs. Although MCL 4.0 has a limited ability to compile a system call into a call to a trap via the Macintosh OS trap emulator, this is almost always undesireable; it sometimes unvolves unnecessary emulator context switch overhead, and not all traps can be emulated.

The code that's invoked by the #_ reader macro in MCL 4.0 looks in a handful of known system libraries for the symbol that follows the #_ reader macro. If it finds such a symbol, it turns the surrounding form into a foreign function call to the shared library entry point associated with that symbol; if the symbol isn't found, it falls back to the strategy of treating the call as an emulated 68K trap call, and generates a compiler warning.

The names of entry points in shared libraries are case-sensitive. However, MCL hides this characteristic from the programmer by encoding the case in the `deftrap` form, and automatically looking it up when the system code is compiled.

A number of system calls were renamed when the MacOS was moved from the 68K to the PowerPC. MCL automatically maps these renamings for you.

There are some traps that appear not to have been moved to the PowerPC at all, and can only be invoked through the emulator.

There were a small number of high-level "not-in-ROM" system calls that were not supported by MCL 3.1. Instead, they were compiled into calls to the corresponding low-level traps. On the PowerPC, both the high-level and low-level calls are shared library entry points. For backward compatibility with MCL 3.1, MCL continues to treat the names of the high-level entries as synonyms for the low-level system calls.

Locating Entry Points in Shared Libraries

The set of shared libraries that the system call expansion code looks in is referred to as the **shared library search path**. Initially, the shared library search path contains entries for “InterfaceLib” (where the vast majority of OS/ToolBox calls reside), “MathLib” (transcendental arithmetic), and “ThreadsLib” (the Thread Manager, used to implement stack groups and processes). If the entry point is not found in one of these installed libraries, then a number of other libraries are automatically installed (one at a time) and searched for the entry point. These additional libraries are “AppleScriptLib”, “ObjectSupportLib”, “QuickTimeLib”, “DragLib”, “TelephoneLib”, “Translation”, "ColorPickerLib", “SpeechLib”, “AOCELib”, “QuickDrawGXLib”, “ColorSync”, “PowerMgrLib”, and “XTNDInterface”.

You can extend the set of libraries searched initially by calling `(add-to-shared-library-search-path` *libname*`)` where *libname* is a case-sensitive string which names the shared library in question. You must call this function to inform MCL of any additional libraries to search for entry points. You can also call this function to pre-install a library that you know your code will be using, to avoid having other libraries unnecessarily installed as part of the search process.

It is possible for more than one library to contain an entry point with a given name. When looking for an entry point in a shared library, MCL simply uses the first library it finds which contains the entry point. This is not a problem for system calls, which are designed by a single company and do not contain duplicates. However, it could be a problem when using shared libraries provided by third parties. For this reason, `deftrap` has been extended to allow an entry point to be associated with a particular library. This specification also improves compilation speed by obviating the need to search for the entry point in a series of libraries.

`add-to-shared-library-search-path` [*Function*]

Syntax `add-to-shared-library-search-path` *name*

Description Adds the library specified by the case-sensitive string *name* to the library search path.

Arguments *name* A string, in which case is significant.

`remove-from-shared-library-search-path` [*Function*]

Syntax `remove-from-shared-library-search-path` *name*

Description Removes the library specified by the case-sensitive string *name* from the library search path.

Arguments *name* A string, in which case is significant.

Locating Shared Libraries

Shared library names do not refer to file names. Shared libraries are sometimes files, but they may also be stored in an application's data fork, in the system file, or even in the ROM. The MacOS provides mechanisms for locating a shared library given its name.

MCL uses these mechanisms, specifically the `GetSharedLibrary` function, passing the name of the shared library desired. This uses the system's shared library search path: first it looks in the application's data fork, then in the files in the application's directory, then in the System Extensions folder, then in the shared library and ROM registries. Since this is the search path used by all the other applications on your system, it should almost always find the correct library. One potential problem is that there's no way to pass any version information to `GetSharedLibrary`, nor is there any way to get version information or the pathname of the library it finds. Hence, if there is more than one version of a shared library in the search path, you'll find the first one.

It is possible to locate a library in a specific location by using `GetDiskFragment` and parsing the `cfrg` resource for library names and versions. You could then use that to specify which file to use for a particular library name. However, this technique violates the abstraction recomended by the OS, and so it should not be necessary.

Compile Time / Run Time Entry Location

When you compile a call to a shared library entry point, the compiled call encodes the entry point name and the name of the library containing the entry point. As long as MCL is running, the address of the entry in the library is also remembered.

When compiled code is saved and restarted (using `save-application`, or when loading compiled code from compiled files into a fresh Lisp), the location of the library and the address of the entry point in the library are forgotten. The first time the code is executed, the library and address of the entry are again looked up using `GetSharedLibrary`. This is the basic mechanism of dynamic linking using DLLs. For this mechanism to work properly, the versions of the shared library available at compile-time and run-time must be compatible.

Defining Traps

The macro `deftrap` defines the calling sequence of a trap. Arguments can be explicitly typed so as to override the definition; for example, an argument may be passed as two words instead of as a single longword.

In MCL 3.1, after performing compile-time type checking on the arguments, high-level traps expand into `stack-trap`, `register-trap`, or `%gen-trap` (discussed in "Macros for calling traps" on page 589).

deftrap [*Macro*]

Syntax `deftrap` {*trap-name*} `([({`*arg*`} {`*mactype*`})]*) ({`*return-place*`}{`*mactype*`}) | nil) {`*implementation-form*`}+)`

Description The `deftrap` macro makes *trap-name* into a symbol for a trap and defines the behavior of the trap.

Arguments

trap-name The name of the trap.
In MCL 3.1, this must be a symbol. In MCL 4.0, it can have a number of forms, as described below.

arg Any symbol whose keyword is not a record field type.
Type checking on the arguments is performed at compile time. Run-time type checking on the Lisp data types of the arguments is also performed if an optimize declaration of `(safety 3)` is in effect. Run-time type checking is always performed on pointer and long arguments.

mactype A record field type. (See "Defining record types" on page 568.)

return-place An argument specifying where the results of the trap call are returned. The value of *return-place* is `:stack`, a register, or `nil`, which indicates that no value is returned.

implementation-form
A Lisp form, one subform of which must `({`*trap-kind*`}{`*trap-number*`}{`*call-arg*`}*).`

trap-kind One of `:trap`, `:stack-trap`, `:register-trap`, or `:no-trap`. When *trap-kind* is `:stack-trap`, *call-arg* can be left off, in which case the *args* of the `deftrap` are used instead. If you specify `:trap`, `deftrap` will generate the correct kind of trap.

trap-number A fixnum, or, in the case of `:no-trap`, a form to be executed in lieu of a trap call.

call-arg If present, either a symbol or `{`*register-key*`} {`*value*`}`. If there is no *call-arg*, `([({`*arg*`} {`*mactype*`})]*)` is used. If *call-arg* is a symbol, it must be a symbol from `([({`*arg*`} {`*mactype*`})]*)`. Otherwise, any number of register keys and values may appear.

In MCL 4.0, the *trap-name* can be any of the following:

symbol-or-string

In both these cases `(string` *symbol-or-string*`)` with the leading underbar removed is the name that will be used to access the trap from Lisp using #_ syntax, as well as the name which will be looked up in the shared libraries.

`(`*macro-name entry-point-name*`)`

Here *macro-name* (a string or symbol) will be used to access the trap from Lisp using #_ syntax and *library-name* is a string naming the shared library entry point.

`(`*macro-name* `(`*shared-library-name*`))`
`(`*macro-name* `(`*shared-library-name entry-point-name*`))`

Here *macro-name* and *entry-point-name* are as before. In the first form, *entry-point-name* defaults to `(string` *macro-name*`)` with the leading underbar removed. *shared-library-name* is the name of the shared library in which to look for *entry-point-name*.

Examples

In MCL 4.0, the following examples are equivalent. The ones that don't explicitly specify the shared-library-name will be looked for in the libraries in the shared library search path.

```
(deftrap "_NewPtr" ((bytecount :signed-long))
   (:a0 :pointer)
   (:register-trap 41246 :d0 bytecount))
```

```
(deftrap (_newptr "NewPtr") ((bytecount :signed-long))
   (:a0 :pointer)
   (:register-trap 41246 :d0 bytecount))
(deftrap ("_NewPtr" ("InterfaceLib"))
         ((bytecount :signed-long))
   (:a0 :pointer)
   (:register-trap 41246 :d0 bytecount))
(deftrap ("_NewPtr" ("InterfaceLib" "NewPtr"))
         ((bytecount :signed-long))
   (:a0 :pointer)
   (:register-trap 41246 :d0 bytecount))
```

An example that actually uses the renaming is:

```
(deftrap ("_open" "PBOpenSync")
         ((paramblock (:pointer :paramblockrec)))
   (:D0 :signed-integer)
   (:register-trap 40960 :A0 paramblock))
```

Additional examples can be found in the interface files provided with MCL, which use the macro `deftrap` extensively.

Examples of calling entry points

This section gives several examples of calling entry points. All of these examples allocate temporary records with `rlet`, which is used to create a record for the duration of a body.

The following code creates a new window and draws inside it.

First, create the window.

```
? (defparameter *w* (make-instance 'window))
```

Within Lisp, call the `PaintRoundRect` procedure to draw and fill a rectangle inside the window. This *Inside Macintosh* definition of this procedure is as follows:

```
PROCEDURE PaintRoundRect (r: Rect; ovalWidth,
                                   ovalHeight: INTEGER);
```

The type of the first argument, `r`, is `:rect`, so you must define a record with `rlet` or `make-record`. Then call the entry point from within MCL using `#_PaintRoundRect`, which corresponds to the `PaintRoundRect` procedure.

```
(rlet ((r :rect
       :top 20
       :left 20
       :bottom 80
       :right 60))
  (with-focused-view *w*
    (#_paintroundrect r 30 30)))
```

A call to the entry point `PtToAngle` gets a result by passing an argument by reference:

```
? (rlet ((angle :integer)
         (r :rect :topleft #@(100 50)
                          :bottomright #@(120 70)))
        (with-focused-view *w*
          (#_Framerect r)
          (#_moveto 110 60)
          (#_lineto 150 20)
          (#_PtToAngle r #@(150 20) angle)
          (%get-word angle)))
```

This entry point call creates a record to hold the result of the call to `StuffHex`, which translates a Pascal string into binary data. It creates another record to call the rectangle required by `FillOval`, and finally it draws the oval with the pattern in the window:

```
? (rlet ((pat :pattern))
    (with-pstrs ((hex-string "1020408102040801"))
      (#_stuffhex pat hex-string)
      (rlet ((r :rect :topleft #@(200 20)
                      :bottomright #@(250 100)))
        (with-focused-view *w*
          (#_filloval r pat)))))
```

Entry point types and Lisp types

When you are calling an entry point, you must know the types of the arguments. You can determine them by consulting the trap definition in *Inside Macintosh.* The types of all arguments is shown at the end of each chapter.

Table 16-1 lists the MCL equivalents of the most frequently used Pascal types.

■ **Table 16-1** Pascal types and their equivalent MCL types

Pascal type	Lisp type
array	`macptr`
boolean	`t` or `nil`
byte	fixnum
char	Lisp character (#*char*)
handle	macptr
integer	fixnum
longword	integer
os-type	`fixnum`, four character string, or symbol with a four character p-name
point	`integer`
pointer	macptr
string	`macptr`
Anything passed by reference (VAR; can be made with `rlet` or `%stack-block`)	macptr
Other records, i.e., a record made with `make-record` or `rlet`	macptr

Records

Records can be viewed from two perspectives: how they are stored and used and how they are passed by Lisp.

Records keep track of blocks of Macintosh memory within Macintosh Common Lisp. As stored and used, a **record** is a contiguous structured block of memory of a specific size, stored on the stack or Macintosh heap. As passed around by Lisp, a record is a simple pointer to Macintosh memory, with no formatting or length information.

To use a record, a program must provide a record type. This record type tells the system how the data at the other end of the pointer should be interpreted. Your program must keep track of the types of all the records you create.

Records have no explicit type, so you can map over a single block of memory in several different ways, as if it were several different types of record. This is convenient, for example, in the case of a window pointer, whose first section is a GrafPort record. The system also allows you to use pointers returned by Macintosh traps as records.

In the following discussion, the word *record* can mean either a block of memory or a pointer to memory, depending on the context. For example, when you allocate a new record with `make-record`, a block of memory is allocated on the heap, and a pointer to the block is returned. For the sake of brevity, this process is described in this way: a record is allocated and returned.

Installing record definitions

In the Interfaces folder, a subfolder of Library, are source files giving definitions of many of the Macintosh records described in *Inside Macintosh*.

If you reference a known Macintosh record and the value of `*autoload-traps*` is true, the record definition is automatically loaded.

Other records may be defined with the macro `defrecord`.

The structure of records

A record has an associated set of **fields** that refer to different portions of the memory block. A record **definition** is a template that defines the fields for a specific type of record.

Each field has a name, a type, and a byte offset into the record. Field names are used to access portions of a record symbolically. Field types are used to determine the size of each field and the way the information in the field is encoded and decoded (for example, a field may itself be a record and therefore contain subfields). Field offsets indicate the position of the field inside the record.

Here is an example of a record definition. It has a name, `foo`, and two fields, `str` and `array`. The field `str` is a string of length 255 and the field `array` is an array of 100 integers:

```
(defrecord (foo :handle)
        (str (:string 255))
        (array (:array :integer 100)))
```

You can access the same portion of a record in different ways by using variant fields. See “Variant fields” on page 571.

Defining record types

Many standard record types are already defined in the Interface files. However, you can also define your own record types with `defrecord`.

`defrecord` [*Macro*]

Syntax `defrecord` *record-name* `&rest` *slot-descriptions*

Description The `defrecord` macro defines a new record type.

Arguments *record-name* Either a symbol that will be used to name the type of record, or a list whose `car` is the symbol used to name the record and whose `cadr` is a keyword that specifies the default type of storage used for the record. See the note at the end of this definition on overriding default record storage.

The package of the symbol used to name the record is ignored; all record names are converted to the keyword package.

The keyword should be either `:pointer` or `:handle`. If the keyword is `:pointer`, records of this record type are allocated and accessed as pointers. If the keyword is `:handle`, they are allocated and accessed as handles. If no keyword is given, the record type is assumed to be `:pointer`.

slot-descriptions

One or more slot descriptions. A standard slot description is a list of the form `(`*name type* `{`*option*`}*)`.

name The name used to access the field in the record. The name cannot be `variant`, which has special meaning (see the next section, "Variant Fields").

type The type of data the field contains; *type* is used to determine the field's length. The *type* must be one of the predefined types (see Table 16-2), a previously defined record type, an array, or a list whose `car` is the symbol `:string` and whose `cadr` is a fixnum from `1` to `255`, which is used to specify the length of the string.

An array type is of the form `(:array` *type dimensions*+`)`, where *type* is defined as in this example and *dimensions* are constant fixnums:

```
(defrecord (foo :handle)
        (str (:string 255))
        (array (:array :integer 100)))
```

option Zero or more of the following:

`:offset` *number*

A fixnum to offset the slot from the beginning of the record.

`:include` *t-or-nil*

A keyword, which can be used only when the slot type is another record. It indicates whether the fields of the other record can be accessed directly, as if they were fields of the record being defined. If the value of this keyword is `nil` (the default), they cannot be accessed directly. If the value is true, they can. In the first slot description, the value of `:include` is automatically true.

Examples

Here are two examples of the syntax of `defrecord`.

```
? (defrecord PenState
    (:pnLoc  point)
    (:pnSize point)
    (:pnMode integer)
    (:pnPat  pattern))
```

This call, one of the calls in the Interface files, creates a record type called `PenState` with four slots.

```
? (defrecord foo
    (field1 :integer :default 42)
    (field2 (array :longint 10))
    (field3 (array :byte 5 5))
    (field4 (some-record-type :handle))
    (field5 (:string 255)))
```

This call creates a new record type called `foo`. The fourth field of this record type is stored as a handle. Records stored as handles are less likely to cause fragmentation of the Macintosh heap, but you must be careful when using them.

△ **Important** The Macintosh ROM is very strict about whether records are passed to it by handle or by pointer. It is recommended that you explicitly specify the storage type by using the `:storage` keyword in calls to `make-record` and `rref` or by using the `href` or `pref` macros.△

MCL records correspond exactly to MPW Pascal packed records, except that Boolean fields always take up a full byte. Fields that are 2 or more bytes long always begin at word boundaries (that is, at even memory locations). Fields that are 1 byte long are padded to 2 bytes if necessary. See *Inside Macintosh* for more details on field size.

Table 16-2 lists the predefined record field types and their lengths.

■ **Table 16-2** Predefined record field types and their lengths

Lisp type	Length in bytes	Equivalent to
`array`	4	`pointer`
`boolean`	1	
`byte`	1	`unsigned-byte`
`signed-byte`	1	
`unsigned-byte`	1	
`character`	1	
`handle`	4	
`integer`	2	`signed-integer`
`signed-integer`	2	
`unsigned-integer`	2	

`long`	4	`signed-long`
`signed-long`	4	
`unsigned-long`	4	
`longint`	4	`signed-long`
`signed-longint`	4	`signed-long`
`unsigned-longint`	4	`unsigned-long`
`ostype`	4	
`point`	4	
`pointer`	4	
`short`	2	`signed-integer`
`signed-short`	2	`signed-integer`
`unsigned-short`	2	`unsigned-integer`
`string`	4	
`word`	2	`unsigned-integer`
`signed-word`	2	`signed-integer`
`unsigned-word`	2	`unsigned-integer`
`single-float`	4	
`double-float`	8	
`ptr`	4	`pointer`

Variant fields

You can use **variant fields** to access the same portion of a record in different ways. Variant fields allow an area of a record to be mapped to different sets of fields. For example, you can use variant fields to access one part of a record as a single longword or as 4 bytes. Variant fields (like records in general) are useful mnemonic aids and short cuts. The size of a variant field is equal to the total size of the largest set of fields in the variant portion.

If a field description contains variant fields, it will have the form

```
(:variant ({variant-field1}+)
          ({variant-field2}+)
          ...
          ({variant-fieldN}+))
```

The section of the record described by the variant may be accessed through *N* sets of fields. The size of the variant field is equal to the size of the largest set of fields.

You can specify an `:origin` keyword argument for a field. The keyword `:origin` simply sets the offset counter.

The following code indicates that a `rect` record may be accessed either as two points or as four coordinates:

```
(defrecord Rect
           (:variant ((top :integer)
                      (left :integer))
                     ((topleft :point :origin 0)))
           (:variant ((bottom :integer)
                      (right :integer))
                     ((bottomright :point :origin 4))))
```

A variant field list can itself use variants.

Creating records

Records may be created temporarily, for example, within a function, using `rlet`, or with indefinite extent, using `make-record`. Records with indefinite extent must be disposed of explicitly. Temporary records are much more efficient.

Creating temporary records with `rlet`

The macro `rlet` is used when memory needs to be allocated temporarily.

`rlet` [*Macro*]

Syntax `rlet ({`*symbol record-type init-forms*`*}`$^{+}$`) {`*form*`}*`

Description The `rlet` macro creates a temporary record on the stack and evaluates *form*. The value of the last *form* is returned. The `rlet` macro is the most efficient way to create temporary records.

The records are stored on the stack and are therefore ephemeral. When the evaluation of *forms* is done, all the records vanish irretrievably and should not be referenced. (This macro is similar to `%stack-block`, to which `rlet` macroexpands.)

The `rlet` macro has the same general form as the `let` macro. For every *symbol* a new binding is created. Space is allocated on the stack for a *record-type* record, and the record is initialized according to the *init-form*, which should be pairs of *field-keyword* and *value*. A pointer to the new record is the value of the corresponding *symbol*. The *forms* are evaluated in the resulting environment, and the value of the last *form* is returned.

If the value of `*autoload-traps*` is true, `rlet` automatically loads the definition of the record.

Arguments

symbol	A symbol.
record-type	A record type.
init-forms	Zero or more initial value forms.
forms	Zero or more forms.

Examples

This example of `rlet` allocates space on the stack for a record `r` of type `:rect`, initializes it with its `:topleft` and `:bottomright` values, and evaluates `(#_framerect r)`. It draws a rectangle into the current GrafPort, the window `foo`:

```
? (setq foo (make-instance 'window))
#<WINDOW "Untitled", #x352819>
? (with-focused-view foo
    (rlet ((r :rect
              :topleft #@(10 10)
              :bottomright #@(100 100)))
      (#_framerect r)))
NIL
```

The binding is ephemeral; `r` no longer has a binding after the value of the last form is returned:

```
? r
> Error: Unbound variable: R
> While executing: SYMBOL-VALUE
```

Here is an example of the expansion of `rlet`.

```
(rlet ((r :rect :topleft #@(10 10) :bottomright #@(100 100)))
  (#_framerect r))
```

macroexpands to

```
(%stack-block ((r 8))
  (ccl::%put-point r 655370 0)
  (ccl::%put-point r 6553700 4)
```

```
  (traps:_framerect r))
```

and then to

```
(let* ((r (ccl::%new-ptr 8)))
  (declare (dynamic-extent r))
  (ccl::%put-point r 655370 0)
  (ccl::%put-point r 6553700 4)
  (traps:_framerect r))
```

If you use the `rlet` macro to allocate a record with `:storage :handle`, it acts as though you overrode the allocation to `:storage :pointer`.

△ **Important** If you override the default storage with `:storage :pointer`, you should use the pointer-specific macros `pref` and `pset` to access the record (or be careful to always specify `:storage :pointer` in `rref` and `rset`). Doing otherwise may cause a crash. △

The *record-type* may also be a record field type. In that case, `rlet` allocates enough storage for one of the specified record fields. For example, the call

```
(rlet ((p :point))...)
```

allocates enough storage to hold a point (that is, 4 bytes). When you allocate storage using record field types, you cannot specify the initial contents.

Creating records with indefinite extent

When you want to return a record from a function, you must create a record that has indefinite extent. The memory this record takes up is not subject to automatic garbage collection; it uses space in the Macintosh heap until it is explicitly disposed of.

Some records, such as windows, must be created and initialized by specific Toolbox routines. Such records should be created not by using `make-record` but by using the appropriate Toolbox traps.

The following macro is used to create records with indefinite extent.

make-record [*Macro*]

Syntax `make-record` *record-name* `&rest` *initforms*

Description The `make-record` macro allocates space in the Macintosh heap for a record *record-name* and returns a pointer or a handle to it. This record has indefinite extent (as opposed to blocks allocated with `%stack-block` or `rlet`).

Arguments

record-name — Either a symbol that will be used to name the type of record, or a list whose `car` is the symbol used to name the record and whose `cadr` is a keyword that specifies the default type of storage used for the record. The keyword should be one of the following:

`:storage` — A keyword used to override the default storage method used by the record. If specified, this should be `:pointer` or `:handle`.

△ **Important** It is recommended that you always specify the storage explicitly, rather than relying on the default. A crash is very likely if a handle is used as a pointer or vice versa. △

`:clear` — A keyword determining how *record-name* is cleared. If the value of `:clear` is true, clears the entire record, using an efficient operating system call to allocate and clear the pointer or handle simultaneously.

`:length` — A fixnum. Used to override the default size used by the record. There is no error checking to see whether `:length` is long enough.

initforms — A list of keywords and values used to initialize the record.

`dispose-record` [*Macro*]

Syntax `dispose-record` *record* `&optional` *storage-type*

Description The `dispose-record` macro disposes of *record* and returns `nil`.

If *record* contains pointers to other records, `dispose-record` does not automatically dispose of these other records, since other pointers to those records may exist.

Any Macintosh Toolbox data structure that is allocated using a special trap (such as regions and controls) also needs to be deallocated using a special trap, rather than by `dispose-record`. In general, if you did not use `make-record` to create the data structure, you should not dispose of it with `dispose-record`.

The `dispose-record` macro has an effect only if *record* is a pointer or a handle to a Macintosh Memory Manager block.

Arguments

record — A record.

storage-type The type of the record being disposed. This may be either a record type or storage (`:pointer` or `:handle`). Supplying this argument allows the macro to expand into more efficient code.

Accessing records

The following macros and functions are used to access and modify records.

href [*Macro*]

Syntax `href` *handle accessor*

Description The `href` macro returns the contents of the specified field of *handle*. This macro is the most efficient way to access the fields of a record.

If the value of `*autoload-traps*` is true, `href` automatically loads the record definition.

An error is signaled at macroexpansion time if an attempt is made to get a handle to a record and the surrounding record is stored as a pointer.

The `href` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try experimenting with `href` expansions to see how this macro works.

The macro `href` may be combined with `setf` to modify a field of a record.

Arguments

handle A handle to a record.

accessor A form that describes which record type and field to access; *accessor* has the form *record-type{.field}*$^+$. An array reference has the form (*record-type {.field}*$^+$ *indices**), for example:

```
(defrecord (foo :handle)
   (str (:string 255))
   (array (:array :integer 100)))

(href f (foo.array 42))
```

record-type A previously defined record type.

field — A field in a record of type *record-type*. If *field* is also a record type, its fields may be accessed by appending an additional period and field name. If that field is a record type, the process can continue. There can be any number of *fields*. Every one but the last must be a record type; the last one may be a record type, but is not required to be.

In cases where the first field of *record-type* is also a record, then that field's fields may be referred to as if they were direct fields of *record-type*. For example, a QuickDraw GrafPort is the first field of a `windowRecord` record, so the GrafPort's `portrect` can be abbreviated from `windowRecord.grafport.portrect` to `windowRecord.portrect`.

If the final *field* of *accessor* is not a record, then the actual field value is returned. If the final *field* of *accessor* is itself a record, then a pointer to that record is returned.

If you do not specify enough array indices, Macintosh Common Lisp returns a pointer to the location in memory where the subarray would begin.

---

pref [*Macro*]

Syntax `pref` *pointer accessor*

Description The `pref` macro returns the contents of the specified field of *pointer*. This macro is the most efficient way to access the fields of a record.

If the value of `*autoload-traps*` is true, `pref` automatically loads the record definition.

An error is signaled at macroexpansion time if the surrounding record is stored as a handle.

The `pref` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try experimenting with `pref` expansions to see how this macro works.

The macro `pref` may be combined with `setf` to modify a field of a record.

Arguments

pointer — A pointer to a record.

accessor — A form that describes which record type and field to access; *accessor* has the form *record-type{.field}*$^+$. An array reference has the form `(`*record-type {.field}*$^+$ *indices*`*)`, for example:

```
(defrecord (foo :pointer)
   (str (:string 255))
   (array (:array :integer 100)))
```

```
(pref f (foo.array 42))
```

record-type — A previously defined record type.

field — A field in a record of type *record-type*. If *field* is also a record type, its fields may be accessed by appending an additional period and field name. If that field is a record type, the process can continue. There can be any number of *fields*. Every one but the last must be a record type; the last one may be a record type, but is not required to be.

In cases where the first field of *record-type* is also a record, then that field's fields may be referred to as if they were direct fields of *record-type*. For example, a QuickDraw GrafPort is the first field of a `windowRecord` record, so the GrafPort's `portrect` can be abbreviated from `windowRecord.grafport.portrect` to `windowRecord.portrect`.

If the final *field* of *accessor* is not a record, then the actual field value is returned. If the final *field* of *accessor* is itself a record, then a pointer to that record is returned.

If you do not specify enough array indices, Macintosh Common Lisp returns a pointer to the location in memory where the subarray would begin.

---

rref [*Macro*]

Syntax `rref` *record accessor* `&key :storage`

Description The `rref` macro returns the contents of the specified field of *record*. It determines the storage type of *record* (i.e. whether it is a pointer or handle) by checking the default storage type of the record definition. However, because records sometimes do not use the default storage type (e.g. a dereferenced handle), this operation is less safe than `pref` or `href`. For that reason, `href` and `pref` are recomended over `rref`.

If the value of `*autoload-traps*` is true, `rref` automatically loads the record definition.

An error is signaled at macroexpansion time if an attempt is made to get a pointer to a record and the surrounding record is stored as a handle. If this is desired, use a `with-dereferenced-handles` form around the call and specify `:storage` to be `:pointer`. Such a pointer to a record within a handle is valid for the duration of `with-dereferenced-handles`.

The `rref` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try experimenting with `rref` expansions to see how this macro works.

Combined with `setf`, `rref` is equivalent to `rset`.

Arguments *record* — A pointer or handle to a record.

accessor — A form that describes which record type and field to access; *accessor* has the form *record-type{.field}*$^{+}$. An array reference has the form (*record-type {.field}*$^{+}$ *indices**), for example:

```
(defrecord (foo :handle)
   (str (:string 255))
   (array (:array :integer 100)))

(rref f (foo.array 42))
```

record-type — A previously defined record type.

field — A field in a record of type *record-type*. If *field* is also a record type, its fields may be accessed by appending an additional period and field name. If that field is a record type, the process can continue. There can be any number of *fields*. Every one but the last must be a record type; the last one may be a record type, but is not required to be.

In cases where the first field of *record-type* is also a record, then that field's fields may be referred to as if they were direct fields of *record-type*. For example, a QuickDraw GrafPort is the first field of a `windowRecord` record, so the GrafPort's `portrect` can be abbreviated from `windowRecord.grafport.portrect` to `windowRecord.portrect`.

If the final *field* of *accessor* is not a record, then the actual field value is returned. If the final *field* of *accessor* is itself a record, then a pointer to that record is returned.

If you do not specify enough array indices, Macintosh Common Lisp returns a pointer to the location in memory where the subarray would begin.

`:storage` — The storage method used for the record. It should be `:pointer` or `:handle`. If omitted, the default storage type for the specified record type is used. It is recommended that you always explicitly specify the storage type.

△ **Important** The storage type used with `make-record` when a record is created must be the same as the storage type specified by any calls to `rref` or `rset` for that record. A crash is very likely if a handle is referenced as a pointer or vice versa. △

Examples

Here are some examples of using `rref`:

```
(rref my-rect :rect.top)
(rref wptr :windowRecord.portrect.bottomright)
(rref tePtr :terec.viewrect.left :storage :pointer)
(rref my-control :control.controlvalue)
```

`rset` [*Macro*]

Syntax `rset` *record accessor value* `&key :storage`

Description The `rset` macro sets the value of a field in a record. To ensure that the right storage type is used, it may be preferable to use `pref` or `href` with `setf`, rather than `rset`.

If the value of `*autoload-traps*` is true, `rset` automatically loads the record definition.

The `rset` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try expanding `rset` to see how it works.

Combined with `setf, rref` is equivalent to `rset`.

Arguments

record	A pointer or handle to a record.
accessor	A form that describes which record type and field to set. For a complete description of record accessors, see `rref`.
value	The new value to store in the field. If the final *field* of *accessor* is a record, *value* must also be a record (either a handle or a pointer) that is copied into the appropriate field of *record*. Attempting to put a new value into an underspecified array reference gets a run-time error.
`:storage`	The storage method used for the record. It should be either `:pointer` or `:handle`. If omitted, the default storage type for the specified record type is used. It is recommended that you always explicitly specify the storage type.

△ **Important** The storage type used with `make-record` when a record is created must be the same as the storage type specified by any calls to `rref` or `rset` for that record. A crash is very likely if a handle is referenced as a pointer or vice versa. △

Examples

```
(rset wptr :window.portrect.topleft #@(100 200))
(rset my-rect :rect.left -10)
(rset teptr :terec.viewrect.top 50 :storage :pointer)
(rset my-control :control.controlvalue 200)
```

`raref` [*Macro*]

Syntax `raref` *record array-descriptor* `&rest` *indices*

Description The `raref` macro accesses an array inside a record. If `raref` is not passed enough indices, it returns a pointer to the place in memory indicated by the indices you have passed. This macro operates on that place in memory and returns the contents of the specified field of the specified record.

This macro cannot access arrays in handles. The handle must be dereferenced before it is passed to `raref`.

Arguments

record A pointer to a record.

array-descriptor
A description of the type and dimensions of the array, of the form `(`*type* `{`*dimension*`}*)`.

indices A set of indices to a location in memory.

Example

Assume that you have a record type `foo`.

```
(defrecord foo
    (field1 :integer :default 42)
    (field2 (array :longint 10))
    (field3 (array :byte 5 5))
    (field4 (some-record-type :handle))
    (field5 (:string 255)))
```

The following three expressions are equivalent:

```
(rref ptr (foo.field3 2 4))
(raref (rref ptr foo.field3) (:byte 5 5) 2 4)
(raref (rref ptr (foo.field3 2)) (:byte 5) 4)
```

`rarset` [*Macro*]

Syntax `rarset` *record value array-descriptor* `&rest` *indices*

Description The `rarset` macro sets the value of an array index inside a record and returns the new contents.

Arguments

record A pointer to a record.

value The new value of the contents of the specified field.

array-descriptor
A description of the type and dimensions of the array, of the form `(`*type* `{`*dimension*`}*)`.

indices A set of indices to a location in memory.

`clear-record` [*Macro*]

Syntax `clear-record` *record* `&optional` *storage-type*

Description The `clear-record` macro clears *record* and returns `nil`.

Arguments

record A record.

storage-type The type of the record being cleared. This may be either a record type or storage (`:pointer` or `:handle`). Supplying this argument allows the macro to expand into more efficient code. If specified, the storage must match the storage given to or defaulted by `make-record`; otherwise, it may cause a crash.

`copy-record` [*Macro*]

Syntax `copy-record` *source-record* `&optional` *record-type dest-record*

Description The `copy-record` macro copies all of the fields of *source-record* (which should be of type *record-type*) into *dest-record* and returns *dest-record*.

Arguments

source-record A record of type *record-type*.

record-type Any previously defined record type.

If the default storage for *source-record* has been overridden, you should not specify *record-type* as well. If you specify *record-type*, then *source-record* and *dest-record* must have been allocated with the default storage. No error checking is performed.

If *dest-record* requires a value but *record-type* does not, give *record-type* the value `nil`.

dest-record A record of type *record-type*, if *record-type* is supplied; if not, a record of the same storage type as *source-record*.

If *dest-record* is not specified, a new one is allocated. If *dest-record* is specified, `copy-record` correctly copies the contents of *source-record* into *dest-record*, even when the storage types of *source-record* and *dest-record* are different.

`get-record-field` [*Function*]

Syntax `get-record-field` *record record-type field-name*

Description The `get-record-field` function returns the value of the *field-name* field of *record*. The `get-record-field` function is much less efficient than the `rref` macro. It should be used only when a function (rather than a macro) is needed, for example, when the record type or field name is not known at compile time.

Unlike the macros listed in this chapter, `get-record-field` needs access to record definitions at run time.

Arguments

record	A record.
record-type	Any record type, given as a keyword.
field-name	The field of the record to be accessed, given as a keyword.

`set-record-field` [*Function*]

Syntax `set-record-field` *record record-type field-name value*

Description The `set-record-field` function sets the value of the *field-name* field of *record* and returns `nil`. The `set-record-field` function is much less efficient than the `rset` macro. It should be used only when a function (rather than a macro) is needed, for example, when the record type or field name is not known at compile time.

Unlike the macros listed in this chapter, `set-record-field` needs access to record definitions at run time.

Arguments

record	A record.
record-type	Any record type, given as a keyword.
field-name	The field of the record to be accessed, given as a keyword.
value	The new value to be placed in the field.

Getting information about records

The following functions give information on records. They are provided primarily for use during development and debugging of programs that use records.

`*record-types*` [*Variable*]

Description The `*record-types*` variable contains a list of all the types of records that are currently defined in the MCL environment.

`*mactypes*` *[Variable]*

Description The `*mactypes*` variable contains a list of all the field types for use in records. (Note that this list does not include record types themselves, which can also be used as field types.)

`find-record-descriptor` [*Function*]

Syntax `find-record-descriptor` *record-type* `&optional` *errorp autoload*

Description The `find-record-descriptor` function returns the record descriptor of *record-type.*

Arguments

record-type A keyword naming a record type.

errorp An argument determining whether an error is signaled if the value of this argument is true and *record-type* is not a valid record type. The default value is true.

autoload An argument determining whether to load *record-type*. The default is true.

`find-mactype` [*Function*]

Syntax `find-mactype` *mactype* `&optional` *errorp autoload*

Description The `find-mactype` function returns the mactype descriptor of *mactype*.

Arguments

mactype A keyword naming a mactype.

errorp An argument determining whether an error is signaled if the value of this argument is true and *mactype* is not a valid record type. The default value is true.

autoload An argument determining whether to load *mactype*. The default is true.

`record-length` [*Macro*]

Syntax `record-length` *record-type*

Description The `record-length` macro returns an integer representing the length of *record-type*.

Argument *record-type* A keyword naming a record type.

`record-fields` [*Function*]

Syntax `record-fields` *record-type*

Description The `record-fields` function returns a list of the fields in *record-type*.

Argument

record-type	A keyword naming a record type.

`record-info` [*Function*]

Syntax `record-info` *record-type* `&optional` *error-p*

Description The `record-info` function returns a list of the offset, type, and length of each field in *record-type.*

Arguments

record-type	A keyword naming a record type.
error-p	An argument specifying the behavior of the function if *record-type* does not name a record type. If this parameter is specified and true, `record-info` signals an error; otherwise it returns `nil`.

`field-info` [*Function*]

Syntax `field-info` *record-type field-name*

Description The `field-info` function returns the offset, type, and length of the field *field-name* of *record-type.*

Arguments

record-type	A keyword naming a record type.
field-name	A field name.

`print-record` [*Function*]

Syntax `print-record` *record record-type* `&optional` *currlevel*

Description The `print-record` function prints the values of the fields of *record* of type *record-type*. No values are returned.

The `print-record` function uses the values of `*print-length*` and `*print-level*`.

Arguments

record	A record.

record-type	Any record type, given as a keyword.
currlevel	The current print level. The default is 0.

`handle-locked-p` [*Function*]

Syntax `handle-locked-p` *handle*

Description The `handle-locked-p` function returns `t` if *handle* is locked, `nil` if it is not.

Trap calls using `stack-trap` and `register-trap`

Most older Macintosh traps accept arguments either on the stack or in registers, but not in both places. Some newer traps accept arguments through both. In general, the operating system traps are register based and the Toolbox traps are stack based, but there are exceptions.

Within a single trap call, some arguments are passed as immediate values and some are passed by reference (that is, a pointer to the value is passed). In general, data 4 bytes long or less is passed by value, and data longer than 4 bytes is passed by reference. Check *Inside Macintosh* for the calling sequences of particular traps. Arguments passed by reference (Pascal VAR parameters) may be modified by the trap.

In MCL 3.1, trap calls can include explicit specifications of where the arguments should be placed, and where the return value should be retrieved from. This information is already encoded in the trap definition, so it is usually not necessary to specify it in the trap call.

Low-level stack trap calls

Here is the general format of a stack trap call.

_TrapName [*Macro*]

Syntax *_TrapName* {*type-keyword argument*}* [*return-value-keyword*]

Description The arguments are evaluated, coerced according to *type-keyword*, and passed as the arguments to the trap.

Arguments

_TrapName — The name of the trap.

type-keyword — A keyword signifying the type of coercion to be performed on the corresponding *argument*. Possible *type-keywords* are `:word`, `:long`, `:ostype`, `:ptr`, and `:d0`. The keywords operate on the subsequent *argument* according to the following list.

- `:word` — This keyword causes *argument* to be passed as a 16-bit word. Arguments passed as words should be fixnums; it is an error if *argument* has more than 16 significant bits.
- `:long` — This keyword causes *argument* to be passed as a 32-bit longword, and is equivalent to `:ostype`, next.
- `:ostype` — This keyword causes *argument*, as a four-character string or symbol with a four-character print name, to be passed as a 32-bit value (8 bits for each of the characters); os-types are used as identifiers by the Resource Manager, Scrap Manager, and other parts of the Macintosh Operating System.
- `:ptr` — This keyword causes *argument* to be passed as a 32-bit macptr.
- `:d0` — This keyword causes *argument* to be passed in the `d0` register.
- `:boolean` — This keyword causes *argument* to be passed as a Boolean value.

argument — An argument to be passed to the stack. As noted previously, *argument* should evaluate to 32-bit values or to macptrs to data on the Macintosh heap or the stack.

return-value-keyword — Indicates the type of value returned by the trap. If *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

- `:word` — A 16-bit result is read from the stack, sign-extended, and returned as a signed Lisp integer.
- `:long` — A 32-bit result is read from the stack and returned as a signed Lisp integer.
- `:ptr` — A 32-bit result is returned from the stack as a macptr.
- `:novalue` — The Macintosh trap does not return a value. The Lisp call returns `nil`.
- `:boolean` — A Boolean value is returned from the stack.

Examples

This form calls the trap `FrameRoundRect` with three arguments: a pointer and two words. The value `nil` is returned.

```
? (let ((oval-width 12)
```

```
    (oval-height 8))
   (%stack-block ((my-rect 8))        ;rectangles are 8 bytes
                 (%put-word my-rect 12 0) ;top=12
                 (%put-word my-rect 40 2) ;left=40
                 (%put-word my-rect 32 4) ;bottom=32
                 (%put-word my-rect 80 6) ;right=80
                 (_FrameRoundRect:ptr my-rect
                                      :word oval-width
                                      :word oval-height)))
```

The following two forms are equivalent, although the first is much easier to read:

```
? (_GetResource :ostype "STR#"    ;ASCII "STR#"
                :word 15          ;resource number 15
                :ptr)             ;return value is pointer

? (_GetResource :word #x5223      ;ASCII "R#"
                :word #x5354      ;ASCII "ST"
                :word 15          ;resource number 15
                :ptr)             ;return value is pointer
```

Note that in the second form, the components of the resource type are pushed in reverse order so that they will be in the correct order on the stack.

Low-level register trap calls

Here is the general format of a register trap call.

_TrapName [*Macro*]

Syntax *_TrapName* [`:check-error`]{*register-keyword argument*}* [*return-register-keyword*]

Description The arguments are evaluated, coerced according to *register-keyword*, and passed as the arguments to the trap.

Arguments *_TrapName* The name of the trap.

`:check-error`
: A symbol. If this symbol appears as the first argument to a register trap, then Macintosh Common Lisp signals an error if the trap returns a negative value in register `d0`. Before using `:check–error`, make sure that the trap in question uses this error-signaling protocol.

register-keyword
: A keyword that specifies the register that holds the subsequent *argument*.

argument
: An argument to be passed to the register. Arguments are evaluated in left-to-right order and placed in registers according to the *register-keywords*. Arguments to be placed in data registers should be 32-bit values. Arguments placed in address registers are not coerced in any way and should be macptrs.

return–register-keyword
: A keyword that specifies which register will hold the value returned by the trap (if any). Recognized register keywords are `:a0` through `:a6` and `:d0` through `:d7`. If *return-register-keyword* is not supplied, then `nil` is returned. Values returned from a data register are returned as 32-bit values. Values are returned from an address register as macptrs. There is no facility for returning multiple values from register traps.

Example

In this example, upon completion `my-pointer` holds a pointer to a 2000-byte block of memory on the current Macintosh heap. If the memory cannot be allocated, a Lisp error is signaled.

```
(setq my-pointer (_newptr :check-error
                          :d0 2000
                          :a0))
```

Macros for calling traps

The forms `stack-trap`, `register-trap`, and `%gen-trap` provide a generalized mechanism for calling Macintosh traps.

In MCL 4.0, these macros can be used to call traps through the 68K emulator.

`stack-trap` [*Macro*]

Syntax `stack-trap` *trap-number* {*trap-keyword argument*}*
[*return-value-keyword*]

Description The `stack-trap` macro expands into an efficient low-level system call to the stack.

Arguments

trap-number A value that evaluates to a 68000 A-trap instruction. These can be found in *Inside Macintosh*. If *trap-number* is specified as a compile-time constant expression, the trap call can be open-coded by the compiler.

type-keyword A value that signifies the type of coercion to be performed on the corresponding *argument*. The keywords operate on the subsequent *argument* according to the following list.

- `:word` A keyword that causes *argument* to be passed as a 16-bit word. Arguments passed as words should be fixnums; it is an error if *argument* has more than 16 significant bits.
- `:long` A keyword that causes *argument* to be passed as a 32-bit longword.
- `:ostype` A keyword that causes *argument*, a four-character string or a symbol with a four-character print name, to be passed as a 32-bit value (8 bits for each of the characters); ostypes are used as identifiers by the Resource Manager, Scrap Manager, and other parts of the Macintosh Operating System.
- `:ptr` A keyword that causes *argument* to be passed as a macptr.
- `:d0` A keyword that causes *argument* to be passed in the D0 register.
- `:boolean` A keyword that causes *argument* to be passed as a Boolean value.

argument An argument to be passed to the stack. As noted previously, *argument* should evaluate to 32-bit values or to macptrs to data on the Macintosh heap or the stack.

`:trap-modifier-bits` *bitmask*
A sequence. Anywhere that a *type-keyword*/*argument* pair may appear, the sequence `:trap-modifier-bits` *bitmask* is also allowed. The `:trap-modifier-bits` specifier causes the associated bitmask to be logical-ored with the value of `_TrapName` and the resulting value used as the trap number. The placement of any `:trap-modifier-bits` forms in the argument list is not significant.

return-value-keyword
Indicates the type of value returned by the trap. If *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

`:word` A keyword that causes a16-bit result to be read from the stack, sign-extended, and returned as a signed Lisp integer.

`:long` A keyword that causes a 32-bit result to be read from the stack and returned as a signed Lisp integer.

`:ptr` A keyword that causes a 32-bit result to be returned from the stack as a macptr.

`:novalue` A keyword that causes no value to be returned from the Macintosh trap. The Lisp call returns `nil.`

`:boolean` A Boolean value is returned.

`register-trap` [*Macro*]

Syntax `register-trap [:check-error ]` *trap-number* {*register-keyword argument*}* [*return-register-keyword*]

Description The `register-trap` macro expands into an efficient low-level system call to the register.

Arguments

`:check-error`
: Signals an error if the trap returns a negative value in register `d0`.

trap-number
: Evaluates to a 68000 A-trap instruction. These instructions can be found in *Inside Macintosh*. If *trap-number* is specified as a compile-time constant expression, the trap call can be open-coded by the compiler.

register-keyword
: Specifies the register that holds the subsequent *argument*. Recognized register keywords are `:a0` through `:a6` and `:d0` through `:d7`.

argument
: An argument to a register call. Arguments are evaluated in left-to-right order and placed in registers. Arguments to be placed in data registers should be 32-bit values. Arguments placed in address registers should be macptrs.

`:trap-modifier-bits` *bitmask*
: Anywhere that a *register-keyword*/*argument* pair may appear, the sequence `:trap-modifier-bits` *bitmask* is also allowed. The `:trap-modifier-bits` specifier causes the associated bitmask to be logical-ored with the value of `_TrapName` and the resulting value used as the trap number. The placement of any `:trap-modifier-bits` forms in the argument list is not significant.

return-register-keyword
: Specifies which register holds the value returned by the trap (if any). Recognized register keywords are `:a0` through `:a6` and `:d0` through `:d7`.

Examples

Here is an example of the use of a register-based trap call with `:trap-modifier-bits.` The trap instruction `_NewPtr` has #x0200 logical-ored into it, causing the ROM to zero out the contents of the newly created pointer.

```
? (defconstant #$trap-clear-bitmask #x0200)
#$TRAP-CLEAR-BITMASK
? (_NewPtr :trap-modifier-bits #$trap-clear-bitmask
           :d0 10 :a0)
#<A Mac Zone Pointer Size 10 #x65F9A>
```

The following forms are equivalent to the previous form:

```
(register-trap _newptr :trap-modifier-bits 512 :d0 10 :a0)
(register-trap (logior _NewPtr 512) :d0 10 :a0))
```

Placement is not significant; this call is equivalent to any of the preceding ones.

```
(_NewPtr :d0 10 :trap-modifier-bits #$trap-clear-bitmask :a0)
```

%gen-trap [*Function*]

Syntax `%gen-trap` *trap* {*type-keyword argument*}* [`:return-block` *pointer* | *return-value-keyword*]

Description The `%gen-trap` function executes a low-level system call to a trap with parameters on the stack or in registers. This function is not available in MCL 4.0.

It returns a value according to `:return-block` or *return-value-keyword*. If the `:return-block` keyword is present, `%gen-trap` returns `nil`. If it is not present, `%gen-trap` returns the value appropriate for *return-value-keyword*.

The compiler open-codes this function.

Arguments *trap* A trap call to a trap word in the range `$A000-$AFFF.`

type-keyword A keyword that indicates what type of coercion to perform on the subsequent argument and where to place the argument. The possible values of *type-keyword* are the following *arguments*:

`:word` The *argument* parameter (which should be a fixnum) is truncated to 16 bits, and the resulting word value is pushed on the stack.

`:long` An object coercible to a 32-bit immediate quantity. The resulting longword value is pushed on the stack.

`:ptr` The *argument* parameter is pushed on the stack as a macptr.

`:d0–:d7` An object coercible to a 32-bit immediate quantity. The resulting longword value is placed in the indicated data register.

`:a0–:a4` The *argument* parameter should be a macptr. Its address is put in the indicated address register.

`:boolean` This keyword causes *argument* to be passed as a Boolean value.

The stack-based arguments are pushed on the stack in the order in which they appear in the call.

`:return-block`
If this keyword appears in a trap call, it should be followed by a pointer to a block of memory where the returned values are placed. This mechanism lets a trap return values from multiple registers and positions on the stack. If this keyword is present, `%gen-trap` returns `nil`.

pointer A pointer.

return-value-keyword
Indicates the type of value the function returns if `:return-block` is not present. If the value of *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

`:word` A 16-bit result is read from the top of the stack, sign-extended, and returned as a signed Lisp integer.

`:long` A 32-bit result is read from the top of the stack and returned as a signed Lisp integer.

`:ptr` A 32-bit result is read from the top of the stack and returned as a macptr.

`:d0-:d7` The value of the indicated data register is returned as a signed Lisp integer.

`:a0-:a4` The value of the indicated address register is returned as a macptr.

`:boolean` A Boolean value is returned.

`:novalue` No value is returned and the trap call returns `nil`.

Notes on trap calls

The following sections discuss 32-bit immediate quantities and Boolean true and false values.

32-bit immediate quantities

When interfacing to the Macintosh ROMs, it is necessary to be able to specify 32-bit immediate quantities. These quantities are used to represent numerical values. When viewed as a sequence of 4 consecutive bytes, they are sometimes used to denote os-types.

The following Lisp data types may be mapped to 32-bit values in contexts where a trap or memory-access primitive requires a 32-bit immediate quantity:

- Any fixnum. Fixnums use only the low 29 bits (in MCL 3.1) or 30 bits (in MCL 4.0).
- Any other integer. The 32 least significant bits of the integer are used. It is an error to pass an integer larger than 32 bits, but the error is not detected.
- A character of type `base-character`. The 32-bit value of the character code is used as the value.
- A string whose length is four characters. Macintosh Common Lisp views such a string as a sequence of 4 bytes; this allows specifying a 32-bit os-type.
- A symbol whose print name is a string whose length is four characters.

The constants `nil` and `t` are not acceptable as arguments to functions (such as `%put-long`) that require a 32-bit value.

Boolean values: Pascal true and false

Pascal parses Boolean values as words (2 bytes) with bit 8 set. Macintosh Common Lisp automatically converts between this representation and the MCL values `nil` and `t`. Thus,

```
(stack-trap _button :boolean)
```

is equivalent to

```
(logbitp 8 (stack-trap _button :word))
```

and

```
(stack-trap foo :boolean x)
```

is equivalent to

```
(stack-trap foo :word (if x -1 0))
```

Chapter 17:

Foreign Function Interface

Contents

This chapter describes Macintosh Common Lisp's Foreign Function Interface (FFI), which permits calls from within Macintosh Common Lisp to functions written in C, Pascal, assembly language, and other languages. Such functions are called foreign functions. Foreign functions can in turn make calls back to Macintosh Common Lisp.

You should read this chapter if you plan to include calls to foreign functions within Macintosh Common Lisp.

You should also be familiar with MPW object files, which are documented in *MPW: Macintosh Programmer's Workshop Development Environment*.

Accessing Foreign Code in MCL 4.0 and 3.1

The mechanisms used to access foreign code in MCL 4.0 and 3.1 are quite different.

In MCL 4.0, access to foreign code is quite simple. The foreign code must be available as a shared library. The entry points of this library are accessed in exactly the same way as Macintosh OS entry points.

In MCL 3.1, foreign code must be available as an MPW object-code file. This file is linked into MCL, and entry points to it are defined.

Foreign Code in MCL 4.0

MCL searches for foreign code libraries using exactly the same search path that it uses for Macintosh OS libraries.

Defining foreign code entry points

You can use `deftrap` to define the entry points for shared libraries created from your own or third-party code, but doing so has a few drawbacks. `deftrap` always defines symbols in the `traps` package, so if any of your entry points have the same name as system calls or entry points from other shared libraries, you will have to rename them appropriately. Also, `deftrap` requires a body, which will serve no purpose in the entry points. (The body could always be specified as `(:no-trap nil)` or some such, but this is just extra work.)

`define-entry-point` automatically generates body code that signals an error at run-time if the entry point was not found at compile time.

`define-entry-point` [*Macro*]

Syntax `define-entry-point` *name arglist*
`({`*return-place*`}{`*mactype*`}) | nil))`

Description The `define-entry-point` macro defines *name* in the current package as an entry point to a shared library.

Arguments

name The name of the entry point. The syntax is as for `deftrap`. In particular, the name can specify the library in which the entry point will be found.

arglist The arglist is as for `deftrap`.

return-type Any valid mactype.

(`define-entry-point`*name arglist*
return-type)

Unlike `deftrap`, all symbols are read in the current package. Like `deftrap`, the defined macro is exported from that package.

The following are equivalent. They all define a macro named `newptr` that invokes the “NewPtr” entry point in the “InterfaceLib” library.

```
(define-entry-point “NewPtr" ((bytecount :signed-long))
   :pointer)
(define-entry-point (newptr "NewPtr")
                    ((bytecount :signed-long))
   :pointer)
(define-entry-point ("NewPtr" ("InterfaceLib"))
                    ((bytecount :signed-long))
   :pointer)
(define-entry-point ("NewPtr" ("InterfaceLib" "NewPtr"))
                    ((bytecount :signed-long))
   pointer)
```

To use the macro defined by one of the forms above, you could write:

```
(newptr 5)
```

Foreign Code in MCL 3.1

The remainder of this chapter describes how to access foreign code in MCL 3.1. These facilities can also be used to access 68K object code in MCL 4.0, through the 68K emulator.

Using the MCL 3.1 foreign function interface

To use foreign functions from Macintosh Common Lisp, do the following:

- Write and compile the foreign functions using a compiler that produces MPW object files.
- Run Macintosh Common Lisp and load the Foreign Function Interface files.
- Load the MPW object files with the function `ff-load`.
- Define an interface for each foreign function you wish to call. (This is done with `defffun`, `deffcfun`, or `deffpfun`.)
- Call the foreign functions from Macintosh Common Lisp using MCL syntax.

A call from Macintosh Common Lisp to a foreign function looks exactly like a call to another MCL function. The MCL function that makes the call (and, for that matter, the programmer) doesn't even need to know that the function called was written in a different language.

To use the Foreign Function Interface you must load the file `ff.fasl`, included in the MCL Library folder, or execute the expression

```
?  (require "ff")
```

High-level Foreign Function Interface operations

The following high-level operations are used with the Foreign Function Interface. They can be used only on object files in the MPW object file format.

`ff-load` [*Function*]

Syntax `ff-load` *files* `&key :entry-names :libraries :library-entry-names :ffenv-name :replace`

Description The `ff-load` function loads the MPW object files specified by *files* and returns a foreign function environment.

The foreign function environment returned consists of code segments, a jump table, a static data area, and a collection of active entry point names. Dead code is removed so that only code and data reachable from the active entry points are included in the environment.

Each call to `ff-load` produces a distinct foreign function environment, with its own global space, function code, and so on. There is no sharing of code or data between environments produced by separate calls to `ff-load`. For example, if two different calls to `ff-load` require a library function `atoi`, they will each get their own copy of `atoi`. If they each refer to a global variable `errno`, they will get their own copy of `errno`. To share data and library code between routines, you must link the routines in a single call to `ff-load`.

Arguments

files A filename, a pathname, or a list of filenames and pathnames of MPW object files.

`:entry-names`
A list of strings naming all the entry points in *files* that should be active. If `:entry-names` is not specified or its value is `nil`, all entry points in *files* are active. Note that these strings are case sensitive.

`:libraries` A list of additional object files to load. These differ from the files in *files* in that, by default, entry points in libraries are not considered active (so that code from libraries is not included in the link unless the code is needed by other functions).

`:library-entry-names`
A list of active entry point names in libraries. This overrides the default for libraries of only including those entry points used by other functions.

`:ffenv-name` A symbol. If `:ffenv-name` is given and its value is not `nil`, `ff-load` checks to see whether an environment with the given name is already loaded. If so, the action taken depends on the value of the `:replace` argument (described next). If not, the specified files are loaded and the resulting environment is given the name passed in this argument. This argument can be used to make `ff-load` behave somewhat like `require`.

`:replace` If `:replace` is given and its value is not `nil`, the files are always loaded and any previously loaded environment of the same name (as specified by the `:ffenv-name` argument) is disposed of. (The previously existing environment is disposed of only if the loading is successful.)

Example

```
(setf (logical-pathname-translations "mpw")
      '(("clib;**;*.*" "hd:mpw:libraries:clibraries:**:*.*")
        ("lib;**;*.*" "hd:mpw:libraries:libraries:**.*.*")
        ("**;*.*" "hd:mpw:**:*.*")))
(defparameter *c-libraries*
 '("mpw:clib;stdclib.o""mpw:lib;interface.o"))

(ff-load "c-hacks;utils.c.o"
         :ffenv-name  'c-utils
         :libraries *c-libraries*
         :library-entry-points
         '("atoi" "strcmp"))

(deffcfun (frob "frob") ...)   ;frob is defined in utils.c
(deffcfun (atoi "atoi") ...)
(deffcfun (strcmp "strcmp") ...)
```

For information on setting up logical pathnames, look in the file `ff-example.lisp` in the FF Examples subfolder of your Examples folder.

---

dispose-ffenv [*Function*]

Syntax `dispose-ffenv` *ffenv*

Description The `dispose-ffenv` function disposes of the heap storage used by the environment *ffenv*. If you call a foreign function residing in an environment that has been disposed of, you will almost certainly crash.

Argument *ffenv* A foreign function environment, as returned by `ff-load`, or a symbol naming a foreign function environment.

Example

See “A Short example” on page 609.

`defffun` [*Macro*]

`deffcfun` [*Macro*]

`deffpfun` [*Macro*]

Syntax

`defffun (`*lisp-name entry-name* `{`*option*`}*) ({`*argspec*`}*) {`*result-flag*`}*`
`deffcfun (`*lisp-name entry-name* `{`*option*`}*) ({`*argspec*`}*) {`*result-flag*`}*`
`deffpfun (`*lisp-name entry-name* `{`*option*`}*) ({`*argspec*`}*) {`*result-flag*`}*`

Description

These macros help you define an MCL interface to a foreign function. You describe the arguments the function takes and the result it returns, and `defffun` defines an MCL function that performs appropriate coercions and type checks on the arguments and calls the foreign function. The `deffcfun` and `deffpfun` macros are identical to `defffun` except that they set the `:language` option.

Arguments

lisp-name The name of the MCL function that is defined by the macro. This must be a symbol.

entry-name The name of an active entry point in a foreign function environment. It should be a string. Entry point names are case sensitive. If *entry-name* exists in more than one loaded environment, the environment used is undefined.

option The entry name may be followed by options that further describe the foreign function. The options to `deffun`, `deffcfun`, and `deffpfun` provide information on the syntax and calling sequence of the function being defined. These options are `:language`, `:check-args`, and `:reverse-args`.

`:language`
The value of `:language` indicates the language used to define the foreign function, which in turn regulates defaults for other options. This option is currently used in the macroexpansion of `deffcfun` and `deffpfun`. Future extensions will support other language types.

`:check-args`
If the value of `:check-args` is non-`nil` (the default), the function performs run-time checks to ensure that the actual argument types match the declared expectations. If its value is `nil`, this type checking is skipped. This option may be overridden by individual *argspecs.*

`:reverse-args`
If the value of `:reverse-args` is non-`nil`, the arguments are pushed on the stack in reverse order from that specified in the *argspec* list. This is the default if the language is C.

argspec The description of a single argument to the foreign function. Argument specifications have the general form (*lisp-type* {*flag*}*). They are described in detail in the next section, "Argument Specifications."

result-flag The value returned by the function. Result flags are described in "Result flags" on page 608.

Example

See "A Short example" on page 609.

Argument specifications

The argument specifications of foreign function calls give information about each argument the foreign function will receive, including the MCL type to expect and a series of flags. The flags give information on the foreign type, the argument-passing method, and the necessity for argument checking.

lisp-type

Any valid MCL type specifier. It declares that the corresponding argument to the MCL function will be of that type. If argument checking is requested, a `check-type` form is included in the MCL function. In addition, *lisp-type* is used to select the argument-passing convention and foreign type, if these are not explicitly specified. If *lisp-type* is a symbol (not a list) and there are no *flags*, the parentheses around *argspec* may be omitted.

flags

Specify the format (foreign type) in which the corresponding argument should be passed, the method for passing, and the necessity for performing a type check of the argument. The following flags describe the format (foreign type) of the argument to be passed to the foreign function. They are mutually exclusive; that is, you may choose only one of the possible values.

`:long`

The foreign function is expecting a longword value. The MCL argument should be an integer or a character (in which case its `char-code` is used). It can also be a four-character string or a symbol.

`:word`

The foreign function is expecting a word value. The MCL argument should be a fixnum. The low 16 bits of the value constitute the foreign argument.

`:double`

The foreign function is expecting a floating-point value in the machine double format (8 bytes). The MCL argument should be of type `double-float`.

`:extended`

The foreign function will receive a floating-point value in SANE extended format (10 bytes). The MCL argument should be of type `double-float`. The `:extended` flag is the default floating-point type for C and Pascal.

`:ptr`

The foreign function will receive a longword value. The argument should be an object of type `macptr`. (See Chapter 15: Low-Level OS Interface for a description of this MCL data type.)

`:Lisp-ref`

The foreign function is expecting a pointer to an MCL value. A temporary location is reserved (for the duration of the call) in nonrelocatable memory, a pointer to the MCL argument is placed in that location, and the address of the location is passed to the foreign function. The foreign function can access the MCL data using double indirection, just as it would a Macintosh handle. The contents of the location are updated whenever the MCL data is relocated.

`:cstring`

The foreign function is expecting a null-terminated string. The MCL argument should be an MCL string. The foreign function must not modify any locations beyond the end of the string. The string may be of any length.

`(:cstring size)`

The foreign function is expecting a null-terminated string in a buffer of *size* bytes (not including the null). The MCL argument should be a string of *size* characters or less. The foreign function must not modify any locations beyond the end of the buffer.

`:pstring`

The foreign function is expecting a Pascal string (a string preceded by a count byte). The MCL argument should be a string. The foreign function must not modify any locations beyond the end of the string. If the argument is longer than 255 characters, an error is signaled.

`(:pstring size)`

The foreign function is expecting a string preceded by a length byte in a buffer of *size* bytes (not including the length byte). The MCL argument should be a string *size* characters long or less. The foreign function must not modify any locations beyond the end of the buffer.

If no foreign format flag is specified, a default is chosen based on the *lisp-type* and the `:language` option, according to Table 17-1. If *lisp-type* is not listed in the table, there is no default (and the foreign type must be specified).

■ **Table 17-1** Foreign type defaults

MCL type	In C	In Pascal
Integer	`:long`	`:word`
Character	`:long`	`:word`
String	`:cstring`	`:pstring`
Float	`:extended`	`:extended`

The following *flags* are used to specify the argument-passing method. They are mutually exclusive; that is, you may choose only one of the possible values.

`:by-value`

The value of the argument is passed to the foreign function (pushed on the stack). This method may not be used to pass strings (that is, the argument format must not be `:cstring` or `:pstring`).

`:by-address`

The address of a location containing a copy of the argument is passed to the foreign function (pushed on the stack). If the foreign function modifies the argument, the changes will not be visible to Macintosh Common Lisp. Use `:by-reference` if you want Macintosh Common Lisp to see changes.

`:by-reference`

The address of a location containing a copy of the argument is passed to the foreign function (pushed on the stack). In addition, arrangements are made so that upon return from the foreign function, any changes in the copy are reflected in the MCL argument itself. Thus foreign functions may destructively modify MCL data structures. Only floating-point values and strings may be passed by reference (that is, the argument format must be `:cstring`, `:pstring`, `:double`, or `:extended`). When a string is passed by reference and the foreign function changes the size of the string, an error is signaled unless the MCL string is adjustable.

When no passing method is specified, the default is to pass `:long`, `:word`, `:ptr`, and `:Lisp-ref` arguments by value and others by address. In addition, if the language is C, `:extended` arguments are passed by value.

The following flags specify whether type checking should be performed on *lisp-type*. Using one of these flags lets you override the `:check-args` option for the function as a whole. The two flags are mutually exclusive.

`:check-arg`

The argument is checked at run time to ensure it is of type *lisp-type*.

`:no-check-arg`

The argument type is not checked at run time.

Result flags

The value returned by the foreign function is described by *result-flags*. These flags describe the type and location of the returned value.

The type of the returned value is described by one of the following keywords. The type determines how the result is coerced before it is passed back to Macintosh Common Lisp.

`:long`
: The result is interpreted as a 32-bit signed integer and returned as a signed MCL integer.

`:full-long`
: The result is equivalent to `:long`; maintained for backward compatibility.

`:word`
: The returned value is a 16-bit word. It is interpreted as a signed integer and returned as a fixnum.

`:double`
: The foreign result is a double float. It is coerced to an MCL double float.

`:extended`
: The foreign result is a float in extended format. It is coerced to
an MCL double float.

`:float`
: This is a synonym for `:extended`.

`:char`
: The foreign result is a character. The low 8 bits of the returned value are interpreted as a character code and returned as an MCL character.

`:ptr`
: The returned value is a 32-bit integer of type `macptr`.

`:novalue`
: The foreign function returns no value. The MCL function returns `nil`. This is the default if the language is Pascal.

The location of the returned value is described by one of the following keywords:

`:d0–:d7` The foreign function returns the value in the specified data register as a signed MCL integer. The default is `:d0` if the language is `:c`.

`:a0–:a4` The value in the specified address register is returned as a macptr. (See Chapter 15: Low-Level OS Interface for a description of macptrs.).

`:stack` The foreign function returns the value on top of the stack as a macptr.

A Short example

Assume that the file `test.c` contains the following C function:

```
#include <ctype.h>
#include <memory.h>

digitval (ch)
  char ch;
{  if isdigit(ch) return ch - '0';
  else return -1;
}
```

After compiling this file in MPW, you can use it from within Macintosh Common Lisp as follows:

```
? (ff-load "test.c.o" :ffenv-name 'test
                      :libraries *c-libraries*)
#<A FF-ENV>
? (deffcfun (digit-value "digitval")
            (character) :long)
DIGIT-VALUE
? (digit-value #\7)
7
? (digit-value #\A)
-1
```

Low-level functions

The following low-level functions are used to implement the higher-level functions described earlier in this chapter. You may want to use the low-level functions if you require increased speed or flexibility. The `ff-call` function is faster than higher-level functions because it is open-coded and the arguments are not coerced or checked for types. This gives more flexibility at the cost of some error checking.

If you use these functions, it is your responsibility to pass the right types; if you pass the wrong types, the system will probably crash.

ff-call [*Function*]

Syntax `ff-call` *pointer* {*type-keyword argument*}* [`:return-block` *pointer*| *return-value-keyword*]

Description The `ff-call` function transfers control to the address *pointer*, passing arguments according to the *type-keyword*/*argument* pairs.

In MCL 4.0, this can be used to call a universal proc through the `CallUniversalProc` trap.

The `ff-call` function returns a value according to `:return-block` or *return-value-keyword*. If the `:return-block` keyword is present, `ff-call` returns `nil`. If it is not present, `ff-call` returns the value appropriate for *return-value-keyword*.

This function is useful if you have `'CODE'` resources with entry points at known offsets. Keep in mind, however, that no type checking is performed on the arguments and that the system will probably crash if bad arguments are passed.

The stack-based arguments are pushed on the stack in the order in which they appear in the call.

If the returned value is to be taken from the stack, room is left on the stack for this value before any stack-based arguments are pushed. The foreign function is entered at *pointer* with the return address on the top of the stack. The function does not have to obey strict stack discipline, but it must never lower the stack beyond its arguments (in other words, it should never pop anything more than its arguments, and it should not modify any data on the stack beyond the arguments).

Arguments *pointer* A pointer to the entry point of the foreign function to be called.

type-keyword Indicates what type of coercion to perform on the subsequent argument and where to place the argument. The possible values of *type-keyword* are the following:

`:word` The MCL argument should be a fixnum. The low 16 bits of the argument are pushed on the stack.

`:long` The MCL argument should be an integer or a character, in which case its `char-code` is used. It can also be a four-character string or a symbol. It is coerced to a longword and the result is pushed on the stack. (See th"32-bit immediate quantities" on page 594.)

`:ptr` An object of type `macptr`. Its associated address is pushed on the stack. (See Chapter 15: Low-Level OS Interface for a description of macptrs.)

`:d0-:d7` The MCL argument should be an integer or character, in which case its `char-code` is used. It can also be a four-character string or a symbol. It is coerced to a longword and the result is placed in the indicated data register. The same constraints apply as in `:long`, described previously.

`:a0-:a4` A macptr. Its associated address is placed in the indicated address register.

`:a5` A macptr. Its associated address is placed in the A5 register and in the Macintosh low memory global `CurrentA5` for the duration of the call.

argument An argument.

`:return-block`
A keyword. If this keyword appears in a call to `ff-call`, it should be followed by a pointer to a block of memory where the returned values are placed. This mechanism lets a foreign function return values from multiple registers and positions on the stack. If this keyword is present, `ff-call` returns `nil`.

return-value-keyword
Indicates the type of value the function returns if `:return-block` is not present. If the value of *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

`:word` A 16-bit result is read from the top of the stack, sign extended, and returned as a signed MCL integer.

`:long` A 32-bit result is read from the top of the stack and returned as a signed MCL integer.

`:ptr` A 32-bit result is read from the top of the stack and returned as an object of type `macptr`.

`:d0-:d7` The value in the indicated data register is returned as a signed MCL integer.

`:a0-:a4`	The value in the indicated data register is returned as a macptr. (See Chapter 15: Low-Level OS Interface for a description of macptrs.)
`:novalue`	No value is returned from the function. The `ff-call` function returns `nil`.

---

ff-lookup-entry [*Function*]

Syntax `ff-lookup-entry` *entry-name*

Description The `ff-lookup-entry` function returns two values describing the entry point named *entry-name*. The entry point must be an active entry point in some previously loaded environment. Note that entry point names are case sensitive. The first value returned is a pointer to the entry point. The second value is the A5 pointer for the environment where the entry point was found. If *entry-name* does not exist, `nil` is returned. If *entry-name* exists in more than one loaded environment, the specific environment returned is undefined.

Using the `ff-lookup-entry` function is a relatively slow operation. You normally call it once at load time and store the results in some easily accessible place, rather than calling it every time you need to reference the entry point. Note that the values returned by `ff-lookup-entry` are pointers and thus cannot be maintained across image saves and restarts. When restarting a Lisp image, you must reinitialize all entry information by calling `ff-lookup-entry` again.

Argument *entry-name* A string giving the name of an entry point in a foreign function environment. The string is case sensitive.

Examples

Here are two examples.

```
;frob is a foreign function:
? (multiple-value-bind (entry a5)
                       (ff-lookup-entry "frob")
                       (ff-call entry :a5 a5))
;FrobCount is a static integer variable:
? (format T "There are ~D frobs."
          (%get-long
           (ff-lookup-entry "FrobCount")))
```

%word-to-int [*Function*]

Syntax `%word-to-int` *fixnum*

Description The `%word-to-int` function sign-extends the low 16 bits of *fixnum* to a full integer value.

Argument *fixnum* A fixnum.

Example

```
? (= (%word-to-int #xFFFF) -1)
T
? (= (%word-to-int 1) 1)
T
```

%copy-float [*Function*]

Syntax `%copy-float` *float*

Description The `%copy-float` function returns a copy of *float*, that is, a newly consed floating-point number that is `eql` but not `eq` to *float*. The argument can be an MCL `double-float` or a macptr to a memory location that contains a 64-bit floating-point number in machine format (see 68881 or SANE documentation for details).

Argument *float* A floating-point number.

Calling Macintosh Common Lisp from foreign functions

A foreign function may call an MCL function, receive a returned value, and do further processing before returning to Macintosh Common Lisp. An MCL function that is to be called by a foreign function must be defined with one of the following macros. The macros create a pointer, which can then be passed to the foreign function.

The `defccallable` macro is used to define MCL functions that can be called from C. The `defpascal` macro is used to define MCL functions that can be called from the Macintosh Toolbox or from user-written Pascal code. Both of these macros put an MCL function in the function cell of a symbol and a pointer to the C or Pascal entry point in the value cell of a symbol.

The `defpascal` and `defccallable` macros are described in Chapter 15: Low-Level OS Interface.

The following example uses `defccallable`. It uses a C function, `addthree`, that takes two arguments: an integer and a pointer to an MCL function. The C function adds 1 to its first argument, then calls the MCL function pointed to by its second argument. This MCL function is passed the incremented first argument.

The MCL function increments its argument and returns it, whereupon the C function increments it again and returns the value. Here control passes from Macintosh Common Lisp to C to Macintosh Common Lisp to C and finally back to Macintosh Common Lisp.

The `addthree` function is accurately named only if it is passed a pointer to an MCL function that takes a `fixnum` argument, increments it, and returns the incremented value.

Here is the C function:

```
int addthree (i, Lispfn)
    int i, (*Lispfn) ();
    {
    i = i + 1;
    i = (*Lispfn) (i);
    return i + 1;
    }
```

The MCL `defccallable` macro sets the value of its first argument to the entry point for the function that it defines. Here is the MCL macro:

```
? (defccallable add-one
            (:long i :long))
            (+ i 1))
ADD-ONE
```

The pointer to the MCL function is not an MCL data type, so use `t` as the type specifier.

```
? (deffcfun (add-three "addthree")
            ((fixnum :long) (t :ptr))
            :long)
ADD-THREE
```

The pointer to the MCL function is stored in the value cell of `add-one`, so you don’t need to quote the symbol or use the special form `function`.

```
? (add-three 5 add-one)
8
```

Extended example

The files `ff-example.c`, `ff-example.Lisp`, and `ff-example.test` in the FF Examples folder in your Examples folder contain an expression-by-expression example of how to use the Foreign Function Interface with C.

Perform the following steps.

- Boot MPW.
- Edit your foreign language code. The examples use a set of C functions defined in the file `example.c` on the Foreign Function Interface disk.
- Compile your code. Use the MPW Build facility to make sure you get all of the right library files.
- Test your code in MPW. This stage isn't strictly necessary but will ensure that you pass MCL-proven working code. If you don't test your code in MPW, it isn't necessary to link it. Macintosh Common Lisp needs only the object files.
- Start Macintosh Common Lisp and load the Foreign Function Interface.
- MCL code for loading the foreign function files and defining an MCL interface to the functions is given in the file `example.lisp` in the FF Examples subfolder of your MCL Examples folder.
- Call the foreign functions from Macintosh Common Lisp.

Examples for testing the code are contained in the file `example.test` in the FF Examples subfolder.

Appendix A:

Implementation Notes

Contents

This appendix describes details of the implementation of Common Lisp by Macintosh Common Lisp. It includes information on cases that are ambiguous in Common Lisp and provides technical information on memory management, the compiler, and other aspects of the Macintosh Common Lisp system.

The Metaobject Protocol

Macintosh Common Lisp version 2 implements CLOS as documented in the second edition of *Common Lisp: The Language*. It also contains some informational functions that are part of the Metaobject Protocol (MOP) as described in *The Art of the Metaobject Protocol* by Gregor Kiczales and others (MIT Press, 1991).

Metaobject classes defined in Macintosh Common Lisp

Table A-1 shows the class structure of the metaobject classes defined in Macintosh Common Lisp version 2. All the metaobject classes are instances of `standard-class` except `generic-function` and `standard-generic-function`, which are instances of `funcallable-standard-class`. They are not documented in *Common Lisp: The Language,* but some of them are documented in *The Art of the Metaobject Protocol.*

■ **Table A-1** Structure of metaobject classes defined in Macintosh Common Lisp version 2

Class	Direct superclasses
`standard-object`	`t`
`structure-object`	`t`
`metaobject`	`standard-object`
`method-combination`	`metaobject`
`long-method-combination`	`method-combination`
`short-method-combination`	`method-combination`
`standard-method-combination`	`method-combination`
`method`	`metaobject`
`standard-method`	`method`
`standard-accessor-method`	`standard-method`
`standard-writer-method`	`standard-accessor-method`
`standard-reader-method`	`standard-accessor-method`

■ **Table A-1** Structure of metaobject classes defined in Macintosh Common Lisp version 2 (continued)

Class	Direct superclasses
`generic-function`	`metaobject` `ccl::funcallable-standard-object`
`standard-generic-function`	`generic-function`
`specializer`	`metaobject`
`class`	`specializer`
`ccl::compile-time-class`	`class`
`structure-class`	`class`
`built-in-class`	`class`
`ccl::std-class`	`class`
`funcallable-standard-class`	`std-class`
`standard-class`	`std-class`

During compilation, if a class definition is encountered for a previously unknown class, an instance of the class named `ccl::compile-time-class` is added to the compilation environment. This instance is a stub only. The Common Lisp generic function `class-name` returns its name and `find-class` finds it if given the compile-time environment as its third argument, but none of the other MOP functions returns any kind of useful information. For example, `class-precedence-list` signals an error when called with an instance of `ccl::compile-time-class`. This way of handling `defclass` at compile time is very likely to change in future versions of Macintosh Common Lisp.

The class named `ccl::std-class` is an implementation detail that may change in future versions of Macintosh Common Lisp; hence its name is not exported. It is included in the above table for completeness.

Unsupported metaobject classes

The following metaobject classes do not exist in Macintosh Common Lisp version 2.0:

```
eql-specializer
forward-referenced-class
slot-definition
standard-slot-definition
standard-direct-slot-definition
standard-effective-slot-definition
```

Unsupported Introspective MOP functions

The following functions, which are part of the de facto Introspective MOP standard, are not supported by Macintosh Common Lisp verson 2.0:

```
class-default-initargs
class-direct-default-initargs
generic-function-argument-precedence-order
generic-function-declarations
generic-function-initial-methods
generic-function-lambda-list
method-lambda-list
slot-boundp-using-class
slot-definition-class
slot-definition-allocation
slot-definition-initargs
slot-definition-initform
slot-definition-initfunction
slot-definition-name
slot-definition-readers
slot-definition-type
slot-definition-writers
slot-exists-p-using-class
slot-makunbound-using-class
slot-value-using-class
```

MCL functions relating to the Metaobject Protocol

The following MOP functions are supported in Macintosh Common Lisp.

`class-direct-subclasses` [*Generic function*]

Syntax `class-direct-subclasses` (*class* `class`)

Description The `class-direct-subclasses` generic function returns a list of the direct subclasses of the given class, that is, all classes that mention this class in their `defclass` forms.

Argument *class* A class.

Example

```
? (defclass foo () ())
#<STANDARD-CLASS FOO>
? (defclass bratch (foo) ())
#<STANDARD-CLASS BRATCH>
? (defclass gronk (foo) ())
#<STANDARD-CLASS GRONK>
? (class-direct-subclasses (find-class 'foo))
(#<STANDARD-CLASS GRONK> #<STANDARD-CLASS BRATCH>)
? ? (class-direct-subclasses (find-class 'standard-object))
(#<STANDARD-CLASS FOO>
 #<STANDARD-CLASS INSPECTOR::ERROR-FRAME>
 #<STANDARD-CLASS INSPECTOR::UNDO-VIEW-MIXIN>
 #<STANDARD-CLASS INSPECTOR::BOTTOM-LINE-MIXIN>
 #<STANDARD-CLASS INSPECTOR::CACHE-ENTRY>
 #<STANDARD-CLASS INSPECTOR::BASICS-FIRST-MIXIN>
 #<STANDARD-CLASS INSPECTOR::OBJECT-FIRST-MIXIN>
 #<STANDARD-CLASS INSPECTOR::UNBOUND-MARKER>
 #<STANDARD-CLASS INSPECTOR::INSPECTOR> #<STANDARD-CLASS
MENUBAR>
 #<STANDARD-CLASS KEY-HANDLER-MIXIN> #<STANDARD-CLASS
APPLICATION>
 #<STANDARD-CLASS CONDITION> #<STANDARD-CLASS SCRAP-HANDLER>
 #<STANDARD-CLASS CCL::LISP-WDEF-MIXIN>
 #<STANDARD-CLASS CCL::INSTANCE-INITIALIZE-MIXIN>
 #<STANDARD-CLASS FUNCALLABLE-STANDARD-OBJECT>
 #<STANDARD-CLASS METAOBJECT>)
```

The file `grapher.lisp` in your MCL Examples folder contains a good example of the use of `class-direct-subclasses`.

---

class-direct-superclasses [*Generic function*]

Syntax `class-direct-superclasses` (*class* `class`)

Description The `class-direct-superclasses` generic function returns a list of the direct superclasses of the given class, that is, all classes that are specified in the class's `defclass` form.

Argument *class* A class.

Example

```
? (defclass my-io-stream (input-stream output-stream) ())
#<STANDARD-CLASS MY-IO-STREAM>
? (class-direct-superclasses *)
(#<STANDARD-CLASS INPUT-STREAM> #<STANDARD-CLASS OUTPUT-
STREAM>)
```

---

class-precedence-list [*Generic function*]

Syntax `class-precedence-list` (*class* `class`)
`class-precedence-list` (*class* `standard-class`)

Description The `class-precedence-list` generic function returns the class precedence list for the given class. This list is used by `compute-applicable-methods` to determine the order of precedence of methods specialized on the class.

Argument *class* A class.

Example

```
? (defclass foo () ())
#<STANDARD-CLASS FOO>
? (class-precedence-list *)
(#<STANDARD-CLASS FOO> #<STANDARD-CLASS STANDARD-OBJECT>
#<BUILT-IN-CLASS T>)
? (defclass bar () ())
#<STANDARD-CLASS BAR>
? (class-precedence-list *)
(#<STANDARD-CLASS BAR> #<STANDARD-CLASS STANDARD-OBJECT>
#<BUILT-IN-CLASS T>)
? (defclass gronk (foo bar) ())
```

```
#<STANDARD-CLASS GRONK>
? (class-precedence-list *)
(#<STANDARD-CLASS GRONK> #<STANDARD-CLASS FOO> #<STANDARD-
CLASS BAR>
 #<STANDARD-CLASS STANDARD-OBJECT> #<BUILT-IN-CLASS T>)
```

`class-prototype` [*Generic function*]

Syntax

`class-prototype` (*class* `ccl::std-class`)
`class-prototype` (*class* `structure-class`)

Description

The `class-prototype` generic function returns a prototype instance of the given class. The contents of the instance are undefined, though it has the same number of instance slots as an instance created with `make-instance` (or a structure creator function), and all class slots are accessible.

Argument

class A class.

Example

In this example, `y` is bound only because of `:allocation :class`.

```
? (defclass foo ()
            ((x :initform 1 :accessor foo-x)
             (y :allocation :class
                :initform 2 :accessor foo-y)))
#<STANDARD-CLASS FOO>
? (foo-y (class-prototype (find-class 'foo)))
2
```

`class-direct-instance-slots` [*Generic function*]

Syntax

`class-direct-instance-slots` (*class* `ccl::std-class`)

Description

The `class-direct-instance-slots` generic function returns a list of slot definition objects describing the instance slots that were declared in the class's `defclass` forms. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name`.

Argument

class A class.

Example

See the example in the definition of `slot-definition-name` on page 629.

`class-direct-class-slots` [*Generic function*]

Syntax `class-direct-class-slots` (*class* `ccl::std-class`)

Description The `class-direct-class-slots` generic function returns a list of slot definition objects describing the class slots that were declared in the class's `defclass` forms. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name`.

Argument *class* A class.

`class-instance-slots` [*Generic function*]

Syntax `class-instance-slots` (*class* `ccl::std-class`)

Description The `class-instance-slots` generic function returns a list of slot definition objects describing all the instance slots, direct and inherited, that were declared in the `defclass` for the class. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name`.

Argument *class* A class.

`class-class-slots` [*Generic function*]

Syntax `class-class-slots` (*class* `ccl::std-class`)

Description The `class-class-slots` generic function returns a list of slot definition objects describing all the class slots, direct and inherited, that were declared in the `defclass` for the class. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name`.

Argument *class* A class.

Example

```
? (defclass foo ()
    ((x :accessor foo-x
        :initarg :x
```

```
        :initform 1)
     (y :allocation :class
        :accessor foo-y
        :initarg :y
        :initform 2)))
#<STANDARD-CLASS FOO>
? (defclass bar (foo)
    ((m :accessor bar-m
        :initarg :m
        :initform 3)
     (n :allocation :class
        :accessor bar-n
        :initarg :n
        :initform (log 4))))
#<STANDARD-CLASS BAR>
? (class-direct-instance-slots (find-class 'bar))
((M (3) (:M)))
? (class-direct-class-slots (find-class 'bar))
((N #<Anonymous Function #xDF2EA6> (:N)))
? (class-instance-slots (find-class 'bar))
((M (3) (:M)) (X (1) (:X)))
? (class-class-slots (find-class 'bar))
((N (1.3862943611198906) (:N)) (Y (2) (:Y)))
```

---

specializer-direct-methods [*Generic function*]

Syntax

`specializer-direct-methods` (*specializer* `class`)
`specializer-direct-methods` (*specializer* `list`)

Description

The `specializer-direct-methods` generic function returns a list of all methods that specialize on the given specializer. An `eql` specializer is represented as a list of length 2 whose `car` is the symbol `eql` and whose `cadr` is an MCL object.

In the default world, the `specializer-direct-methods` lists are not cached. The first time you call `specializer-direct-methods` or `specializer-direct-generic-functions`, it maps over all the generic functions, computing the direct methods lists for all specializers. It also enables caching of this information for subsequent calls to `add-method` and `remove-method`. This caching is preserved across calls to `save-application`. The function `clear-class-direct-methods-caches` clears all the cached information and stops `add-method` from keeping track of it until the next call to `specializer-direct-methods` or `specializer-direct-generic-functions`.

Argument *specializer* A class or a list of the form `(eql` *object*`)`.

specializer-direct-generic-functions [*Generic function*]

Syntax `specializer-direct-generic-functions` (*specializer* `class`)
`specializer-direct-generic-functions` (*specializer* `list`)

Description The `specializer-direct-generic-functions` generic function returns a list of all generic functions that specialize on the given specializer. An `eql` specializer is represented as a list of length 2 whose `car` is the symbol `eql` and whose `cadr` is an MCL object.

In the default world, the `specializer-direct-generic-functions` lists are not cached. The first time you call `specializer-direct-methods` or `specializer-direct-generic-functions`, it maps over all the generic functions, computing the direct methods lists for all specializers. It also enables caching of this information for subsequent calls to `add-method` and `remove-method`. This caching is preserved across calls to `save-application`. The function `clear-class-direct-methods-caches` clears all the cached information and stops `add-method` from keeping track of it until the next call to `specializer-direct-methods` or `specializer-direct-generic-functions`.

Argument *specializer* A class or a list of the form `(eql` *object*`)`.

generic-function-methods [*Generic function*]

Syntax `generic-function-methods` (*generic-function* `standard-generic-function`)

Description The `generic-function-methods` generic function returns a list of the methods for *generic-function*.

Argument *generic-function* A generic function.

Example

```
? (defmethod foo ((x integer)) x)
#<STANDARD-METHOD FOO (INTEGER)>
? (defmethod foo ((x fixnum)) (+ x (call-next-method)))
#<STANDARD-METHOD FOO (FIXNUM)>
? (generic-function-methods #'foo)
(#<STANDARD-METHOD FOO (FIXNUM)> #<STANDARD-METHOD FOO
(INTEGER)>)
```

method-function [*Generic function*]

Syntax `method-function` (*method* `standard-method`)

Description The `method-function` generic function returns the function that runs when *method* is invoked. This function takes the same number of arguments as the generic function. If it was generated from code containing a call to `call-next-method`, function-calling it with `funcall` will cause Macintosh Common Lisp to crash. (Otherwise it can be function-called safely.)

Argument *method* A standard method.

method-generic-function [*Generic function*]

Syntax `method-generic-function` (*method* `standard-method`)

Description The `method-generic-function` generic function returns the generic function associated with *method*, or `nil` if there is none.

Argument *method* A standard method.

Example

```
? (setq m (defmethod foo ((x integer)) x))
#<STANDARD-METHOD FOO (INTEGER)>
? (method-generic-function m)
#<STANDARD-GENERIC-FUNCTION FOO #xD61B66>
? (remove-method (method-generic-function m) m)
#<STANDARD-GENERIC-FUNCTION FOO #xD61B66>
? (method-generic-function m)
NIL
```

method-name [*Generic function*]

Syntax `method-name` (*method* `standard-method`)

Description The `method-name` generic function returns the name of *method*

Argument *method* A standard method.

Example

```
? (defmethod foo ((x integer)) x)
#<STANDARD-METHOD FOO (INTEGER)>
```

```
? (method-name *)
FOO
```

---

method-qualifiers [*Generic function*]

Syntax `method-qualifiers` (*method* `standard-method`)

Description The `method-qualifiers` generic function returns a list of the qualifiers for *method*. (See *Common Lisp: The Language,* pages 839, 849.)

Argument *method* A standard method.

---

method-specializers [*Generic function*]

Syntax `method-specializers` (*method* `standard-method`)

Description The `method-specializers` generic function returns a list of the specializers for *method*.

Argument *method* A standard method.

Example

```
? (defmethod bar ((x integer) (y list)) (cons x y))
#<STANDARD-METHOD BAR (INTEGER LIST)>
? (method-specializers *)
(#<BUILT-IN-CLASS INTEGER> #<BUILT-IN-CLASS LIST>)
```

---

slot-definition-name [*Generic function*]

Syntax `slot-definition-name` (*slot-definition* `list`)

Description The `slot-definition-name` generic function returns the name of *slot-definition*. Future versions of Macintosh Common Lisp will fully support the `slot-definition` class.

Argument *slot-definition* A slot definition object.

Example

```
? (defclass foo () (x y))
#<STANDARD-CLASS FOO>
? (mapcar 'slot-definition-name
     (class-direct-instance-slots *))
```

```
(X Y)
```

`copy-instance` [*Generic function*]

Syntax `copy-instance` (*instance* `standard-object`)

Description The `copy-instance` generic function returns a copy of the given instance. The default method merely copies the vector used to store the instance slots for the instance. Users may add methods to perform additional initialization for the copied method.

Argument *instance* An instance of `standard-object` or one of its subclasses.

Example

There are examples of `copy-instance` in the Interface Toolkit source code.

`clear-specializer-direct-methods-caches` [*Function*]

Syntax `clear-specializer-direct-methods-caches`

Description The `clear-specializer-direct-methods-caches` function clears all the cached information returned by `specializer-direct-methods` and `specializer-direct-generic-functions`, preventing subsequent calls to `add-method` from caching this information. The next call to either of these functions recomputes the caches and reenables maintenance of them by `add-method`.

`clear-clos-caches` [*Function*]

Syntax `clear-clos-caches`

Description The `clear-clos-caches` function clears CLOS caches in preparation for doing a `save-application` if the value of the `:clear-clos-caches` keyword argument to `save-application` is true (the default). (See Appendix B: Workspace Images.) This function clears the effective method caches stored inside generic functions and the valid initialization argument caches stored inside classes.

`clear-gf-cache` [*Function*]

Syntax `clear-gf-cache` *generic-function*

Description The `clear-gf-cache` function clears the cached effective methods for *generic-function*. This function saves space but causes subsequent invocations of the generic function to be slower until the cache is filled again.

Argument *generic-function*
A generic function.

`clear-all-gf-caches` [*Function*]

Syntax `clear-all-gf-caches`

Description The `clear-all-gf-caches` function calls `clear-gf-caches` on all generic functions. This function is called by `clear-clos-caches`.

`method-exists-p` [*Function*]

Syntax `method-exists-p` *generic-function* `&rest` *args*

Description The `method-exists-p` function returns `nil` if *generic-function* is not a generic function or a symbol naming a generic function, or if `(apply` *generic-function args*`)` would cause an error because there are no applicable primary methods for the given arguments to the generic function. Otherwise, it returns one of the applicable primary methods for *generic-function*. This function is faster than `compute-applicable-methods` and does not `cons`.

Arguments *generic-function*
A generic function or a symbol naming a generic function.
args One or more arguments to the generic function.

`*check-call-next-method-with-args*` [*Variable*]

Description The `*check-call-next-method-with-args*` variable determines whether a run-time check is made during calls to `call-next-method`.

When the value of this variable is true (the default), then the check is made to ensure that new arguments do not change the set of methods that are applicable for the generic function.

When the value of this variable is `nil`, *then* no check is made.

The checking is not completely strict. If the required arguments that are passed to `call-next-method` are `eq` to the original required arguments passed to the generic function, then the test passes.

For effective methods that have already been cached, changes to `*check-call-next-method-with-args*` will not take effect until `clear-all-gf-caches` is invoked.

`*defmethod-congruency-override*` [*Variable*]

Description

The `*defmethod-congruency-override*` variable allows you to override standard MCL behavior when you define global generic functions.

When the value of this variable is `nil` *(the default), then* an error is signaled.

When the value of this variable is true, then Macintosh Common Lisp does not signal an error if the function binding of the generic function's name is not a generic function or if a method's lambda list is not congruent to its generic function's lambda list.

If `*defmethod-congruency-override*` is a function, then it is called with two arguments as described next.

If an attempt is made by `defmethod` or `ensure-generic-function` to redefine a regular function, macro, or special form, `*defmethod-congruency-override*` is called with two arguments, the function name (a symbol) and `nil`. If `nil` is returned, an error is signaled. Otherwise, the redefinition is performed.

If `add-method` is instructed to add a method to a generic function and the lambda lists of the method and the generic function are not congruent, `*defmethod-congruency-override*` is called with two arguments, the generic function and the method. If it returns `nil`, an error is signaled. Otherwise, all methods are removed from the generic function, the generic function's lambda list is redefined to be congruent with the method's lambda list, and the method is added.

If `*defmethod-congruency-override*` is not `nil` and not a function, it behaves as if it were a function that always returns non-`nil`. Hence, redefinitions are performed silently. This is very dangerous and should usually be done only by patch files.

MCL class hierarchy

The file `print-class-tree.lisp` in the MCL Examples folder contains functions to print the class hierarchy of an MCL class in a way that makes the direct superclasses and the class precedence list apparent. It includes, as a comment, a hierarchy diagram for every class in the MCL system, sorted by class name.

Types and tag values

MCL uses low tags to indicate the basic types of objects. The mapping between tags and Common Lisp types is an implementation detail that is likely to change in future version of Macintosh Common Lisp.

Tags in MCL 3.1

In MCL 3.1, references to Lisp objects are encoded in 32-bit 680x0 longwords. The 3 least significant bits of the longword are referred to as the object's **tag** and determine the type of the object (see the list of tag values that follows). In the case of **immediate objects** such as fixnums, characters, and short floats, the value of the object is contained in the remaining 29 bits. In other cases, the 32-bit longword constitutes a **tagged pointer** to the associated object.

A consequence of this tagging scheme is that all nonimmediate Lisp objects are allocated on 8-byte boundaries.

- The tag value of 0 is used to represent fixnums; the two's-complement value of the fixnum is stored in the upper 29 bits of the longword. Fixnums can therefore store values in the range `-2^28` through `(2^28)-1`, inclusive. Note that this representation allows the direct use of machine arithmetic instructions where applicable.
- The tag value of 1 is used to represent variable-length objects called **uvectors.** Objects with this tag include all arrays, CLOS instances, structure instances, bignums, ratios, complex numbers, `macptr` pointers, packages, and a few more internal types. A pointer that contains this tag points 1 byte beyond the beginning of the storage occupied by the object it points to.
- The tag value of 2 represents symbols; symbol pointers therefore point 2 bytes into the storage allocated to the symbol.

- The tag value of 3 represents double-precision floating-point values; such pointers point 3 bytes into the 64-bit double-float.
- The tag value of 4 represents `cons` cells and `nil`. Since the `car` of a `cons` cell occupies the first of two longwords allocated to that cell, **cons-tagged** objects point at the `cdr` of the `cons` cell.
- The tag value of 5 is used to represent instances of the **short-float** data type; the upper 29 bits of the longword encode a sign bit, a 5-bit exponent, and a
23-bit significand.
- The tag value of 6 is used to denote functions; all valid Lisp objects with this tag point to executable machine code.
- The tag value of 7 is used to represent small immediate objects, including characters. The least significant byte of a character contains the value `#xF`.

  If bits 8 to 15 of the character contain `#xF`, then the character is a **base character** (a Lisp object of type `base-character`). The character code of the character is contained in the most significant word of the object; if the character is a base character, then this value must be in the range 0 through 255 inclusive.

  The Lisp character type `extended-character` is not implemented in this release of Macintosh Common Lisp. For now at least, all Lisp objects of type `character` are of type `base-character`.

  Other immediate objects whose tag is 7 and whose low byte is *not* `#xF` are used to represent various constants used by the Memory Manager and the run-time system.

Tags in MCL 4.0

MCL 4.0 uses the low three bits of an object as a tag. The low two bits identify all objects that user code can get ahold of. Bit 2 (value 4) is used along with the low two bits to additionally distinguish between user objects and internal objects (e.g. uvector headers). The four kinds of user objects and their low 2 bits are:

tag	**Object Type**
0	fixnum
1	list (cons or `nil`)
2	uvector
3	immediate (e.g. character)

Uvectors (tag 2) are further sub-tagged in the header of their representation in memory. For more details on the tagging scheme, see the file "compiler;ppc;ppc-arch.lisp".

Raw Object Access

The following functions provide low-level access to objects that are tagged as uvectors.

`uvectorp` [*Function*]

Syntax `uvectorp` *object*

Description The `uvectorp` function returns true if *object* is tagged as a uvector.

Argument *object* A variable-length `uvector` object.

`uvsize` [*Function*]

Syntax `uvsize` *object*

Description The `uvsize` function returns the size of *object* as a fixnum.

Argument *object* A variable-length `uvector` object.

`uvref` [*Function*]

Syntax `uvref` *object index*

Description The `uvref` function returns the element of *object* at *index*. The function signals an error unless `(<= 1` *index* `(uvsize` *object*`))`.

The function `setf` may be used with `uvref` to modify an element of a uvector. If *object* is a simple array, `uvref` is the same as `aref`.

Arguments *object* A variable-length `uvector` object.

index An index into *object*.

Reader macros undefined in Common Lisp

In addition to supporting the standard Common Lisp reader macro characters, Macintosh Common Lisp defines the following dispatching reader macros, which are undefined in Common Lisp:

`#@` Transforms the subsequent list of two fixnums into a point.

`#$` Should be followed by a symbol *symbol*. Interns `$`*symbol* in the `traps` package and, if `*autoload-traps*` is true, attempts to load an interface constant definition. See Chapter 16: OS Entry Points and Records.

`#_` Should be followed by a symbol *symbol*. Interns `_`*symbol* in the `traps` package and, if `*autoload-traps*` is true, attempts to load a trap definition. See Chapter 16: OS Entry Points and Records.

Numeric arguments in pathnames

Macintosh Common Lisp uses the CLOS `#P` syntax for pathnames, but it also has a numeric argument that specifies one of four possible unusual conditions in the pathname. These numeric arguments are an error in Common Lisp and should not be used in portable code:

`#1P` Means that the type of a pathname is `:unspecific`.

`#2P` Means that the name of a pathname is `""`.

`#3P` Means that the type of a pathname is `:unspecific` and its name is `""`.

`#4P` Means that the namestring represents a logical pathname.

Numbers

Fixnums are stored as immediate data using a two's-complement representation. They are 29 bits long in MCL 3.1 and 30 bits long in MCL 4.0 (see discussion of the tagging scheme, in "Types and tag values" on page 633). Note that `eql` fixnums are `eq` (although portable code should not rely on this fact).

In MCL 3.1, two internal floating-point data formats are supported. The format used to represent instances of the Common Lisp data types `single-float`, `double-float`, and `long-float` corresponds to IEEE double or 64-bit floating-point format. Each such floating-point number is 64 bits (8 bytes) long and consists of a sign bit, an 11-bit binary exponent (allowing exponents in the range –1022 through 1023), and a 52-bit significand. The significand precision is 53 bits. Floating point numbers in MCL 4.0 are discussed below.

The format used to represent instances of the Common Lisp data type `short-float` consists of 3 tag bits, a sign bit, a 5-bit binary exponent, and a 23-bit significand (for tags, see discussion of the tagging scheme, in “Types and tag values” on page 633). This format is similar to the IEEE single format, but the smaller exponent restricts the range of representable numbers (for example, the values of `least-positive-short-float` and `least-negative-short-float`) and does not allow the representation of denormalized numbers.

On Macintosh computers that do not have floating-point hardware, MCL 3.1 emulates that portion of the floating-point instruction set that it uses.

The following functions are extensions to Common Lisp.

`bignump` [*Function*]

Syntax `bignump` *number*

Description The `bignump` function returns a Boolean indicating whether *number* is a bignum.

Argument *number* A number.

`fixnump` [*Function*]

Syntax `fixnump` *number*

Description The `fixnump` function returns a Boolean value, `t` if *number* is a fixnum, `nil` if it is not.

Argument *number* A number.

`lsh` [*Function*]

Syntax `lsh` *fixnum count*

Description The `lsh` function logically shifts *fixnum* by *count* and returns the result of the operation, which must also be a fixnum. This is the same as the Common Lisp `ash` function, except that any bits shifted out of the (currently) 29 bits of a fixnum are lost.

Arguments

fixnum	A fixnum.
count	An integer.

Floating point numbers in MCL 4.0

MCL 4.0 does not support short floats. The Common Lisp type `short-float` maps to the same type of object as `double-float`.

The compiler inlines the operations `+`, `-`, `*`, `/` when the operands are known to be double-floats.

Floating-point exceptions are, for the most part, enabled and detected. By default, all threads start up with `overflow`, `underflow`, `division-by-zero`, and `invalid-operation` enabled, `inexact-result` disabled, and rounding-mode set to `nearest`. The functions `get-fpu-mode` and `set-fpu-mode` provide higher-level control over floating point behavior.

To simplify floating point exception signaling, `arithmetic-error` is now a subclass of `error`, rather than of `simple-error`. It is provided with a `:report` method.

`get-fpu-mode` [*Function*]

Syntax `get-fpu-mode`

Description Returns a list of keyword/value pairs which describe the floating-point exception-enable and rounding-mode control flags for the current stack-group or process. The list is of the form:

```
(:rounding-mode rounding-mode-keyword
:overflow boolean
:underflow boolean
:division-by-zero boolean
:invalid boolean
:inexact boolean)
```

rounding-mode-keyword must be one of `:nearest`, `:zero`, `:positive`, or `:negative`. The boolean values indicate whether the corresponding IEEE exception is enabled or not. Each MCL thread begins execution with the rounding mode set to `:nearest`, the `:overflow`, `:division-by-zero`, and `:invalid` exceptions enabled and the `:inexact` and `:underflow` exceptions disabled.

Arguments *no arguments*

set-fpu-mode [*Function*]

Syntax `set-fpu-mode &key` *rounding-mode overflow underflow division-by-zero invalid inexact*

Description Sets the current thread's exception-enable and rounding-mode control flags to the indicated values for the arguments that are supplied and preserves the values associated with those that aren't supplied.

`set-fpu-mode` returns the value that would be returned by `get-fpu-mode` after these changes have been made.

If supplied, the value of *rounding-mode* must be one of `:nearest`, `:zero`, `:positive`, or `:negative`.

Arguments

`rounding-mode`	*rounding-mode-keyword*
`overflow`	*boolean*
`underflow`	*boolean*
`division-by-zero`	*boolean*
`invalid`	*boolean*
`inexact`	*boolean*)

The following useful macros could be written with `get-fpu-mode` and `set-fpu-mode`:

```
(defmacro with-fpu-mode ((&rest options) &body body)
  (let* ((old-mode (gensym)))
    `(let* ((,old-mode (get-fpu-mode)))
       (unwind-protect
         (progn
           (set-fpu-mode ,@options)
           ,@body)
         (apply #'set-fpu-mode ,old-mode)))))

(defmacro with-overflow-disabled (&body body)
  `(with-fpu-mode (:overflow nil) ,@body))
```

Characters and strings

MCL has built-in classes for characters and strings. The classes `base-character` and `extended-character` are subclasses of `character`. The classes `base-string` and `extended-string` are subclasses of `string`.

MCL 4.0 follows the Common Lisp standard in that the `:element-type` argument to the function `make-string` defaults to `character`. However in MCL 3.1 `:element-type` defaults to `*default-character-type*.` The initial value of `*default-character-type*` is `base-character`.

If `:element-type` is not specified and `:initial-element` is specified as an `extended-character`, the resulting string is an `extended-string`.

An `extended-string` allocates 16 bits for each character in the string. However, the `schar` function with an `extended-string` will not return an `extended-character` if the `character` at the specified position only requires 8 bits. In this case, a `base-character` is returned.

Ordering and case of characters and strings

MCL has various functions that order strings and characters, as well as functions that transform strings and characters from upper case to lower case and from lower case to upper case. The correct ordering and changing the case of characters are functions of the script in which the string or character is interpreted.

The new special variable `*string-compare-script*` determines how to order strings or characters. The following functions are extended to use the variable `*string-compare-script*`:

```
string-equalchar-equallower-case-p
string-greaterpchar-greaterpupper-case-p
string-lesspchar-lesspalpha-char-p
string-not-equalchar>alphanumericp
string-not-greaterpchar<
string-not-lesspchar>=
string>char<=
string<char-upcase
string>=char-downcase
string<=
string-upcase
string-downcase
string-capitalize
```

A description of the special variable `*string-compare-script*` follows.

`*string-compare-script*` [*Variable*]

Description The value of `*string-compare-script*` is an integer, for example, the value of a script constant such as `#$smRoman` or the value of the system script currently in effect (i.e., `#$smSystemScript`). The default is `#$smSystemScript`. Your system must have the specified script installed.

The script manager

The Macintosh script manager stores strings as a mixture of 8-bit and 16-bit characters in a string, whereas Lisp can not. To account for this, all MCL 3.0 functions that move characters between macptrs and Lisp strings take an optional script argument. The script determines which 8-bit characters in the string referenced by the macptr are the first byte of a 2-byte character. (For more information on 2-byte characters, see the section "2-byte Character Encodings" in Chapter 1 of *Inside Macintosh: Text*.)

If a character in a Lisp string requires more than 8 bits to represent it and the first byte of that character is a valid first byte in the specified script, 2 bytes are moved to the destination macptr. If the character requires more than 8 bits to represent it and the first byte is not a valid first byte, only the lower 8 bits are moved to the destination macptr.

The functions affected by this change are `%get-string`, `%get-cstring`, `%put-string`, and `%put-cstring`.

Script manager utilities

The following functions and variable are used when working with the script manager.

`set-extended-string-script` [*Function*]

Syntax `set-extended-string-script` *script*

Sets the script to use for printing extended-strings. If the script is not set explicitly, the default is the system script if it is a 2 byte script; otherwise the default is an installed 2 byte script. If there are no installed 2 byte scripts, the default is `nil`.

Arguments *script* A script, as described by Inside Macintosh.

`set-extended-string-font` [*Function*]

Syntax `set-extended-string-font` *font-spec*

Sets the font to use for printing extended strings. If not set explicitly, the default is the `#$smScriptAppFond` for the extended-string script.

Arguments *font-spec* A font spec, as described by the MCL Reference Manual.

`convert-kanji-fred` [*Function*]

Syntax `convert-kanji-fred` *oldpath* `&optional` *newpath* `(`*if-exists* `:supersede)`

This function is used to convert files produced by KanjiFred to a format that can be used by MCL 4.0 and 3.1.

Arguments

oldpath The pathname of the file to convert.

newpath The pathname at which to store the converted file. The default is *oldpath*.

if-exists A keyword describing what action to take if *newpath* already exists. The allowed keywords are the same as for `copy-file`.

`*input-file-script*` [*Variable*]

This variable can be set to a 2 byte script to be used when reading a text file that was not created with Fred and is known to contain text in a single 2 byte script.

String lengths

The following functions return string lengths.

`byte-length` [*Function*]

Syntax `byte-length` *string* `&optional` *script start end*

Description The function `byte-length` returns the length in 8-bit bytes of *string*. The arguments *start* and *end* specify a subset of *string*.

Arguments *string* A Lisp string.

script	The script in which the string is interpreted. The default is `#$smSystemScript`, which is the system script.
start	The starting position of the string count. The default is `0`.
end	The ending position of the string count. The default is `(length string)`.

`pointer-char-length` [*Function*]

Syntax `pointer-char-length` *macptr length* `&optional` *script*

Description The function `pointer-char-length` returns the length in characters of the string pointed to by *macptr*.

This function returns three values. The first is the length in characters required to represent the string as a Lisp structure. The second value is a boolean value that indicates whether any 2-byte characters are necessary to represent the string in Lisp. If the value is true, at least one 2-byte character is necessary. The third value is a boolean that is true if *length* falls after the first byte of a 2-byte character in the string pointed to by *macptr*.

Arguments

macptr	A Macintosh pointer.
length	The length in bytes of the string pointed to by *macptr*.
script	The script in which the string is represented. The default is the system script.

`%str-from-ptr-in-script` [*Function*]

Syntax `%str-from-ptr-in-script` *pointer length* `&optional` *script*

Description Gets a Lisp string from a `macptr` pointer interpreted in *script*. The result is an extended string if any of the characters in the source are 16 bits wide.

pointer	A pointer of type `macptr`.
length	The length in bytes of the source string.
script	The script in which the string is represented. The default is `#$SmSysScript.`

Arrays

Default array contents

The `:initial-element` argument to `make-array` has no defined default. In particular, code should not rely on the `:initial-element` argument defaulting to `nil`.

When an array is grown using `vector-push-extend` or `adjust-array`, the contents of newly added elements is undefined. Newly added elements are not initialized to `nil`.

Array element types and sizes

Table A-2 lists the distinct types of array element that are supported.

■ **Table A-2** Types of array element

Type	Length (bits per element)
`bit`	1
`character`	8
`double-float`	64
`(unsigned-byte 8)`	8
`(signed-byte 8)`	8
`(unsigned-byte 16)`	16
`(signed-byte 16)`	16
`(unsigned-byte 32)`	32
`(signed-byte 32)`	32
`t`	One node (32 bits per element)

Only simple vectors are supported directly. All arrays of rank other than 1 are implemented as displaced arrays. In addition to the memory needed to store its elements, a simple vector requires 8 bytes of overhead; a bit vector requires 9 bytes. A complex (displaced) array has about `32+ (4*` *rank*`)` bytes of overhead. The rank of an array must be less than `#x2000` (8K).

No array may have more elements than the number equal to `most-positive-fixnum` (that is, `2*28-1`); therefore, only fixnums are valid array indices.

Table A-3 gives the theoretical limits on the sizes of arrays.

■ **Table A-3** Theoretical limits on array length

Type	Length limit
`bit`	`most-positive-fixnum`
`character`	`#xFFFFF8`
`double-float`	`#x1FFFFF`
`(unsigned-byte 8)`	`#xFFFFF8`
`(signed-byte 8)`	`#xFFFFF8`
`(unsigned-byte 16)`	`#x7FFFFC`
`(signed-byte 16)`	`#x7FFFFC`
`(unsigned-byte 32)`	`#x3FFFFE`
`(signed-byte 32)`	`#x3FFFFE`
`t`	`#x3FFFFE`

All these limits represent arrays requiring approximately 16 MB of contiguous memory.

There is no limit on the size of individual dimensions of an array except the limits imposed by the total array size.

Multidimensional arrays and arrays that were created with non-`nil` values for the `:displaced-to` and/or the `:fill-pointer` arguments to `make-array` are stored as two vectors, a header and a storage vector.

`displaced-array-p` [*Function*]

Syntax `displaced-array-p` *array*

Description The `displaced-array-p` function returns `nil` if *array* is not a displaced array. If it is a multidimensional array or an array created with non-`nil` values for the `:displaced-to` and/or the `:fill-pointer` arguments to `make-array`, the function returns two values, the storage vector and the offset from the beginning of the storage vector to the beginning of the storage for the array.

Argument *array* An array.

Example

```
? (setq a (make-array 10))
```

```
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
? (setq b (make-array 5
                    :displaced-to a
                    :displaced-index-offset 3))
#(NIL NIL NIL NIL NIL)
? (displaced-array-p b)
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
3
? (eq * a)
T
```

Packages

Macintosh Common Lisp, following the forthcoming ANSI Common Lisp standard, uses the package name `common-lisp` instead of `lisp`. The only external symbols of the `COMMON-LISP` package are the approximately 900 symbols of Common Lisp.

The `CCL` package uses the `COMMON-LISP` package. Its exported symbols consist of extensions to Common Lisp provided by Macintosh Common Lisp. The `CCL` package shadows none of the Common Lisp symbols.

The `COMMON-LISP-USER` package uses both the `CCL` and the `COMMON-LISP` packages.

The default value of the `:use` argument to `make-package` and to `defpackage` is the value of the variable `*make-package-use-defaults*`. The initial value of this variable is `("COMMON-LISP" "CCL")`. (See *Common Lisp: The Language,* page 263.)

Macintosh Common Lisp includes a `lisp` package that behaves similarly to the one described in the first edition of *Common Lisp: The Language*. However, full compatibility is not guaranteed.

The variable `*autoload-lisp-package*` determines whether the `LISP` package is loaded when it is first referenced. The value of `*autoload-lisp-package*` is `nil`. If you are running your own code that depends on the `LISP` package, or using code such as PCL or Richard Waters's pretty printer (see *Common Lisp: The Language,* Chapter 27), you may need to do one or more of the following:

- Set the value of `*autoload-lisp-package*` to `t`. You can use the Environment dialog on the Tools menu. When the value of this variable is true, the `:lisp` package is automatically loaded when it is required.

- Load the file `lisp-package.lisp` or `lisp-package.fasl` from the Library folder. This source file defines the `lisp` package.

If you are running your own code, convert it if possible.

When you run code that depends on the `lisp` package and it is not loaded, a restart provides the opportunity to load it.

Additional printing variables

In addition to the standard Common Lisp printer variables (see *Common Lisp: The Language),* Macintosh Common Lisp uses the variables in Table A-4 to control printing.

Table A-4 Additional printing variables

Variable	Purpose
`*print-simple-vector*`	Determines how simple vectors are printed. Default is `nil`; prints simple vectors according to the value of `*print-array*`. If true, prints simple vectors readably. If an integer, prints simple vectors with a length less than the integer readably.
`*print-simple-bit-vector*`	Determines how simple bit vectors are printed. Default is `nil`; prints simple bit vectors according to the value of `*print-array*`. If true, prints simple bit vectors readably. If an integer, prints simple bit vectors with a length less than the integer readably.
`*print-string-length*`	Determines how strings are printed. Default is `nil`; all strings are printed in full. If an integer, strings with a length greater than the integer are printed using an abbreviated format.
`*print-structure*`	Determines whether structures are printed readably using `#S` syntax. Default is `t`; structures are printed in an abbreviated format. If true, structures are printed readably.

`*print-abbreviate-quote*` Determines whether lists whose first element is the symbol quote or the symbol function are printed specially.

Default is true, the lists are printed specially. If the value is `nil`, they are not..

Memory management

Macintosh Common Lisp divides the application heap into two areas: a Lisp heap and a Macintosh heap. Most Lisp data structures (such as `cons` cells, symbols, arrays, and functions) are stored in the Lisp heap; most Macintosh data structures (such as Window records, bitmaps, and CODE resources) are stored in the Macintosh heap.

Garbage collection

In some programming languages, memory management can be a problem. One of the advantages of Macintosh Common Lisp (and Lisp in general) is that you do not need to explicitly deallocate storage for variables or other data structures; Macintosh Common Lisp handles this automatically for you.

To implement memory management, Macintosh Common Lisp provides a **garbage collector,** a small routine that periodically recycles the memory from unneeded data structures.

Macintosh Common Lisp has two kinds of automatic memory management, called ephemeral garbage collection and full garbage collection.

Ephemeral garbage collection

In general, most heap-allocated objects become inaccessible soon after they are created. The ephemeral garbage collector exploits this by concentrating its efforts on reclaiming memory allocated to newly created objects.

The ephemeral garbage collector partitions the population of all dynamically allocated objects into three sets, called generations. Generations are divided roughly into ages by time of creation. Objects first go into the space allocated to the youngest generation. When that space fills up, Macintosh Common Lisp performs an ephemeral garbage collection on only that space, clearing it of all objects. Objects that cannot be reclaimed are promoted to the middle generation. When the middle generation fills up, Macintosh Common Lisp reclaims space within its partition, promoting surviving objects to the oldest generation.

Only when all three sets are full is a full garbage collection invoked.

The function `gc-thermometer`, defined in `thermometer.lisp` in your Examples folder, provides a graphic display of the EGC's behavior.

Guidelines for enabling the EGC

Ephemeral garbage collection can be enabled and disabled by calling the function `egc`.

If you experience disruptive pauses while interacting with Macintosh Common Lisp, you should consider enabling the EGC. A full garbage collections take longer than an ephemeral garbage collection, and is more disruptive.

However, while ephemeral collections are much briefer, they are also much more frequent than full collections. Overall, garbage collection uses more system resources when the EGC is enabled. Because of this, EGC is only recomended when you need to increase your interactivity. Compilations and other time-consuming non-interactive computations are more appropriately performed with the EGC is disabled.

EGC in MCL 3.1

In MCL 3.1, the EGC is most effective when it can work cooperatively with a hardware Memory Management Unit (MMU). It will be able to do this on a 68040-based Macintosh, or on a 68030-based Macintosh with virtual memory or the PTable system extension installed.

If MMU support is unavailable, the ephemeral garbage collector scans all older generations to find the occasional case where such an assignment has taken place. This overhead can be very significant in virtual memory environments; whether or not it is acceptable in real memory environments depends on the speed of the processor, the size limits associated with the ephemeral generations, and the behavior and needs of the application.

The EGC in MCL 4.0 does not require MMU support.

Controlling the EGC

The ephemeral garbage collector is said to be enabled when Macintosh Common Lisp has been asked to use it; it is said to be active when Macintosh Common Lisp is in fact using it. (It may be enabled but inactive when, for instance, free space in the heap is less than the size limit of the youngest generation.)

The following functions can be used to control and configure the ephemeral garbage collector.

`egc` [*Function*]

Syntax `egc` *enable-p*

Description The `egc` function attempts to enable and activate the ephemeral garbage collector if the value of *enable-p* is true. (See the discussion of enabling and activation preceding this definition.) If the value of *enable-p* is `nil`, `egc` disables the ephemeral garbage collector. The function returns `t` if the ephemeral garbage collector is enabled after performing the operation, `nil` otherwise.

Argument *enable-p* A Boolean value.

`egc-enabled-p` [*Function*]

Syntax `egc-enabled-p`

Description The `egc-enabled-p` function returns `t` if the ephemeral garbage collector is enabled and returns `nil` otherwise.

`egc-active-p` [*Function*]

Syntax `egc-active-p`

Description The `egc-active-p` function returns `t` if the ephemeral garbage collector is active and returns `nil` otherwise.

`configure-egc` [*Function*]

Syntax `configure-egc` *generation-0-size generation-1-size generation-2-size*

Description If the ephemeral garbage collector is not currently enabled, the `configure-egc` function sets the size limits of the ephemeral generations as indicated and `t` is returned. If the ephemeral garbage collector is enabled, the current values of the size limits are not affected and `nil` is returned.

The arguments should be nonnegative integers that specify the size limits in kilobytes that the ephemeral garbage collector should use for the three ephemeral generations.

Arguments

generation-0-size	A positive integer.
generation-1-size	A nonnegative integer.
generation-2-size	A nonnegative integer.

`egc-configuration` [*Function*]

Syntax `egc-configuration`

Description The `egc-configuration` function returns three integer values that express the size limits in kilobytes associated with ephemeral generations 0, 1, and 2.

Enabling the EGC programmatically

You can decide programmatically whether to enable the EGC using the MCL function `egc-mmu-support-available-p`. This function is useful in applications intended for users with unknown Macintosh configurations.

`egc-mmu-support-available-p` [*Function*]

Syntax `egc-mmu-support-available-p`

Description In MCL 4.0, this function always returns true.

In MCL 3.1, This function returns true if MCL determines that the system has a 4K or 8K page size.

Full garbage collection

Macintosh Common Lisp uses a mark/compact/forward garbage collector. Garbage collection occurs automatically as memory is needed. This can happen in response to a Macintosh Operating System call or to a memory request by Macintosh Common Lisp. You can invoke garbage collection manually through the function `gc`.

The garbage collector in MCL 3.1 optionally performs a limited amount of event processing, sufficient to partially handle `suspend` and `resume` events and to allow background tasks to run. The garbage collector's event handling does not handle window update events. It simply draws a gray pattern into regions it is expected to update and notifies Lisp's low-level event dispatcher that windows need to be updated. It also does not handle the conversion of the Clipboard on MultiFinder context switches.

`gc-event-check-enabled-p` [*Function*]

Syntax `gc-event-check-enabled-p`

Description The `gc-event-check-enabled-p` function returns a Boolean value, indicating whether Macintosh Common Lisp performs event processing during garbage collection. A value of `t`, the default, means that event processing is turned on during garbage collection.

`set-gc-event-check-enabled-p` [*Function*]

Syntax `set-gc-event-check-enabled-p` *boolean*

Description The `set-gc-event-check-enabled-p` function turns garbage-collector event processing on or off according to the value of *boolean.*

Argument *boolean* A flag. If the value of *boolean* is true, Macintosh Common Lisp performs event processing during garbage collection.

Garbage Collection Statistics

The following functions provide information on the garbage collections that have been performed in the course of a Lisp session.

`gctime` [*Function*]

Syntax `gctime`

Description The `gctime` function returns five integer values:

- the total number of milliseconds spent in all full and ephemeral garbage collections in the current session
- the total number of milliseconds spent in all full garbage collections in the current session
- if the ephemeral garbage collector is enabled, the total number of milliseconds spent in all ephemeral collections of generation 2 in the current session. If the EGC is not enabled, this value is 0.
- if the ephemeral garbage collector is enabled, the total number of milliseconds spent in all ephemeral collections of generation 1 in the current session. If the EGC is not enabled, this value is 0.
- if the ephemeral garbage collector is enabled, the total number of milliseconds spent in all ephemeral collections of generation 0 in the current session. If the EGC is not enabled, this value is 0.

`gccounts` [*Function*]

Syntax `gccounts`

Description The `gccounts` function returns five integer values:

- the total number of full and ephemeral garbage collector invocations in the current session
- the total number of full garbage collector invocations in the current session
- if the ephemeral garbage collector is enabled, the total number of times the ephemeral garbage collector has been invoked on generation 2 in the current session. If the EGC is not enabled, this value is 0.
- if the ephemeral garbage collector is enabled, the total number of times the ephemeral garbage collector has been invoked on generation 1 in the current session. If the EGC is not enabled, this value is 0.
- if the ephemeral garbage collector is enabled, the total number of times the ephemeral garbage collector has been invoked on generation 0 in the current session. If the EGC is not enabled, this value is 0.

Termination

Termination[1] is a facility for running an action when an object is about to be garbage-collected. This action can perform auxiliary clean-up operations associated with the disposal of the object.

MCL 4.0 provides a full termination facility. MCL 3.1 provides a modest termination facility that works only for macptrs.

Termination in MCL 4.0

Termination of an object in MCL 4.0 proceeds in four stages:

- 1. The object is registered for termination. This is accomplished by calling `terminate-when-unreachable` on the object and on a termination function.
- 2. During garbage collection, it is noticed that the object has become unreachable. The object is moved to the termination queue, and removed from any weak hash-tables which contain it.
- 3. Sometime after garbage collection, the termination queue is drained, by calling the termination functions on the corresponding objects in the termination queue. This may be done automatically or under program control.
- 4. On the next garbage collection, if the object is still unreachable (i.e. if the termination functions have not generated live pointers to the object), it is garbage collected. If the object has been made reachable by one or more of the termination functions, it will not be garbage collected, and it will no longer be registered for termination; it must be reregistered for termination if that is desired.

Note that termination is a property of an object, not a class. If you want all the instances of a class to subject to termination, you must register each of the instances individually, for example in an `initialize-instance` method.

1. In some languages, this functionality is termed "finalization." MCL uses the term "termination" to avoid confusion with the Common Lisp concept of class finalization. The MCL termination mechanism is modeled on the mechanism designed and implemented for Apple Dylan.

`terminate-when-unreachable` [*Function*]

Syntax `terminate-when-unreachable` *object* `&optional` (*function* `terminate`)

Description Registers *object* for termination with the termination function. *function* should be a function of one argument. *function* will be called with *object* as its argument when *object* becomes unreachable and the termination queue is drained.

Each call of `terminate-when-unreachable` on a single (`eq`) object registers a new termination function. All will be called when the object becomes unreachable. The order in which they will be called is unspecified. If `terminate-when-unreachable` is called multiple times with the same object and same termination function, it is undefined whether the termination function will be called once or multiple times.

The ability to associate multiple termination functions with a single object may be removed in future versions of MCL.

Arguments

object	Any object.
function	A function of one argument.

`terminate` [*Generic Function*]

Syntax `terminate` *object*

Description The default termination function. A predefined method on `t` does nothing. Programmers should add methods for their own objects, as needed.

In normal operation, this function is called by `drain-termination-queue`. It should not generally be called explicitly by the programmer.

`terminate` [*Method*]

Syntax `terminate` (*object* `t`)

Description The default method ignores *object* and returns `nil`.

`drain-termination-queue` [*Function*]

Syntax `drain-termination-queue`

Description Drains the termination queue. That is, calls the termination functions for every object that has become unreachable.

If `*enable-automatic-termination*` is true (the default), `drain-termination-queue` is called automatically on the first event-check following each garbage collection. In this case, it does not need to be called by the user program.

`*enable-automatic-termination*` [*Variable*]

Description If true (the default) `drain-termination-queue` will be automatically called on the first event check after the garbage collector runs. If you set this to false, you are responsible for calling `drain-termination-queue`.

Note that in the future built-in features of MCL may rely on termination, so you shouldn't simply shut it off if you decide it's no longer needed for your objects.

`cancel-terminate-when-unreachable` [*Function*]

Syntax `cancel-terminate-when-unreachable` *object* `&optional` *function*

Description Removes the effect of the last call to `terminate-when-unreachable` for *object* and *function* (both tested with `eq`). Returns true if it found a match. If the object has been moved to the termination queue since `terminate-when-unreachable` was called, a match will not be found.

If *function* is `nil` or unspecified, then it is not used in determining a match. Instead, the most recent termination function installed for the object is removed.

`termination-function` [*Function*]

Syntax `termination-function` *object*

Description Returns the function passed to the last call of `terminate-when-unreachable` for *object*. If the object has been moved to the termination queue since `terminate-when-unreachable` was called, `nil` is returned.

Termination in MCL 3.1

The file "`macptr-termination.lisp`" in the Library folder provides a simple termination mechanism for MCL 3.1.

This mechanism works with pre- and post-gc hooks. Each hook is a series of functions which are called just before or just after a garbage collection.

The pre-gc hooks are not guaranteed to be called. They will only be called if an event-dispatch occurs between the time memory is depleted and the time the garbage collection occurs.

By default, the hooks are only called on full garbage collections.

add-pre-gc-hook [*Function*]

delete-pre-gc-hook

add-post-gc-hook

delete-post-gc-hook

Syntax `add-pre-gc-hook` *hook*

Description These functions add and remove pre- and post-gc hooks. Because hooks are compared with `EQ`, it is best to pass a symbol that has a global function definition.

set-post-egc-hook-enabled-p [*Function*]

Syntax `set-post-egc-hook-enabled-p` *value*

Description Enables the running of post-gc hooks on ephemeral collections.

`post-egc-hook-enabled-p` [*Function*]

Syntax `post-egc-hook-enabled-p`

Description Returns `true` if the post gc hook is to be called after EGC as well as after full GC.¨

Macptrs and termination in MCL 3.1

The post-gc hook facility can be used to create terminable Macptrs in MCL 3.1.

`make-terminable-macptr` [*Function*]

Syntax `make-terminable-macptr` *macptr termination-function*

Description Creates and returns a terminable `macptr`. It points at the same Mac Heap address as the `macptr` argument. When the return value becomes scavengeable (that is no longer accessible in the Lisp heap), it calls the *termination-function* with a single argument, the returned `macptr`. If the *termination-function*'s return value is non-`nil`, it frees the `macptr`. Otherwise, it assumes that you decided not to terminate it, and calls the termination-function again the next time the GC runs and it is scavengeable.

`deactivate-macptr` [*Function*]

Syntax `deactivate-macptr` *macptr*

Description If *macptr* is an active gc-able `macptr` or terminable `macptr`, make it inactive by disabling its intention to take termination action when it is reclaimed, and return t. If it is either an ordinary `macptr` or an already-inactive gc-able `macptr` or a terminable `macptr`, return `nil`. If it is not a `macptr`, signal an error.

Evaluation

Macintosh Common Lisp offers two evaluator options: a standard evaluator and a compiling evaluator.

- The standard evaluator conforms to Common Lisp standards as described in the second edition of *Common Lisp: The Language*, Chapter 20. However, `evalhook` and `applyhook` were removed from the Common Lisp standard by vote of the X3J13 committee in November 1989 (after *Common Lisp: The Language* went to press). Macintosh Common Lisp still supports them, but they are deprecated.
- The compiling evaluator compiles nontrivial expressions and then runs them. For looping or self-recursive constructs, the compiling evaluator is much faster (up to several hundred times). The compiling evaluator is used when the variable `*compile-definitions*` is non-`nil`.

In the default environment, the system uses the compiling evaluator; that is, the value of `*compile-definitions*` is `t`.

The following variable governs the behavior of the evaluator.

`*compile-definitions*` [*Variable*]

Description The `*compile-definitions*` variable determines whether MCL expressions are compiled. (See the introductory remarks in this section.)

If the value of this variable is true (the default), then all function definitions and most top-level forms are compiled.

If the value of this variable is `nil`, *then* no compilation is performed.

The value of this variable can be toggled in the Environment dialog box on the Tools menu.

Compilation

This section describes some of the behavior of the MCL compiler and describes some means of influencing that behavior.

Tail recursion elimination

The MCL compiler attempts to minimize the stack usage of compiled functions by being properly **tail recursive.** A function is tail recursive if it returns the value(s) of the last function it calls as its own. In that case, the stack space allocated for the function's returned value(s) can be deallocated before it begins execution.

One side effect of the elimination of tail recursion is that, in general, the Stack Backtrace tools display only a portion of the execution history, since those function calls in which tail recursion was eliminated are no longer awaiting return values.

The compiler can be advised that tail recursion should never be eliminated from calls to certain single-valued global functions. Do this by adding the names of those functions to the list that is the value of the variable `ccl::*nx-never-tail-call*`. You can also use customized compiler policy objects to control when the compiler eliminates tail recursion. (See "Compiler policy objects" on page 662.)

Self-referential calls

Within a named function, the compiler may assume that a call to a function of the same name refers to the same function (unless that function has been declared not inline). Although this approach allows such calls to be compiled slightly more efficiently, debugging tools such as `trace` and `advise` violate this assumption.

This aspect of the compiler's behavior can also be controlled through appropriate use of compiler policy objects.

Compiler policy objects

A `compiler-policy` object is a data structure whose components advise the compiler of the desirability of performing (or avoiding) certain optimizations. Usually, compiler policy objects specify how `optimize` declarations are to be interpreted. (For `optimize` declarations, see *Common Lisp: The Language.*)

Separate compiler policy objects are used for file compilation and for interactive compilation, although the default values of these objects specify identical behavior.

The function `new-compiler-policy` is used to create a compiler policy object and to override the implementation's default behavior. The functions `set-compiler-policy` and `current-compiler-policy` set and return the compiler policy used for interactive compilation (including the use of the `compile` function). The functions `set-current-file-compiler-policy` and `current-file-compiler-policy` set and return the policy object used to compile functions that will be saved in `fasl` files.

---

compiler-policy [*Class name*]

Description The `compiler-policy` class is the class of compiler policy objects.

---

new-compiler-policy [*Function*]

Syntax

```
new-compiler-policy &key :allow-tail-recursion-
elimination :inhibit-register-allocation :trust-
declarations :open-code-inline :inhibit-safety-
checking :inhibit-event-polling :inline-self-calls
:allow-transforms :force-boundp-checks :allow-
constant-substitution
```

Description The `new-compiler-policy` function creates and returns a new compiler policy in which the default specifications of behavior are overridden by the values associated with the indicated keyword arguments.

Each of these keywords may take one of the following values:

- `nil`, which specifies that the associated behavior is suppressed
- `t`, which specifies that the associated behavior is performed
- a function that takes arguments as described here and returns a Boolean value

Unless otherwise noted, the functions are called with a (possibly null) lexical environment as their lone argument. To determine what value to return, they may reasonably use functions such as `declaration-information` to extract information about the `optimize` declarations (and other declarations) in effect in that environment.

Setting a new compiler policy completely shadows any existing policy.

Arguments `:allow-tail-recursion-elimination`

When this value is `nil` or a function that inspects the environment and returns `nil`, the compiler does not eliminate tail recursion. The default value is a function that returns true unless the value of the `debug` optimize quantity in the environment is 3.

`:inhibit-register-allocation`

When this value is true or a function that returns true, the compiler does not allocate frequently used values in registers. The default value is a function that returns true when the value of the `debug` optimize quantity in the environment is 3.

`:trust-declarations`

When this value is true or a function that returns true and the value of the `safety` optimize quantity in the environment is not 3, the compiler attempts to exploit type declarations to produce faster and/or smaller code. If those declarations are incorrect, the resulting code may show unpredictable behavior. The default value is a function that returns true if, within the environment, the value of the `speed` optimize quantity is not less than the value of the `safety` optimize quantity.

`:open-code-inline`

When this value is true or a function that returns true and the compiler sees a call to a function that has been declared inline or a call to a primitive operation implemented in the MCL kernel, the compiler may replace that call with a larger (but possibly faster) sequence of instructions. The default value is a function that returns `t` if, within the environment, the value of the `speed` optimize quantity is two or more units greater than the value of the `space` optimize quantity.

`:inhibit-safety-checking`

When this value is true or a function that returns true and the value of the `safety` optimize quantity in the environment is not 3, the compiler is licensed to omit safety checks. (When the compiler performs safety checks, incorrect programs cause errors to be signaled.) The default value is a function that returns `t` if, within the environment, the value of the `speed` optimize quantity is 3 and the value of the
`safety` optimize quantity is 0.

`:inhibit-event-polling`
: When this value is true or a function that returns true, the compiler may omit instruction sequences that poll for events from loops that are otherwise uninterruptible. The default value is a function that returns `t` if, within the environment, the value of the `speed` optimize quantity is 3 and the value of the `safety` optimize quantity is 0.

`:inline-self-calls`
: When this value is true or a function that returns true, the compiler may assume that within a globally named function, calls to a global function of the same name may be compiled without reference to the function cell of the symbol that names that function. The default value is a function that returns `t` unless the value of the `debug` optimize quantity in the environment is 3.

`:allow-transforms`
: When this value is true or a function that returns true, the compiler expands compiler macros and may perform other source-to-source transforms. The default value is a function that returns `t` unless the value of the `compilation-speed` optimize quantity in the environment is 3 or the value of the `debug` optimize quantity in the environment is 3.

`:force-boundp-checks`
: When this value is true or a function that returns true or when the value of the `safety` optimize quantity is 3, the compiler ensures that variables are bound before referencing them. If a function is provided, it should take two arguments, a symbol that names a variable and a lexical environment. Ordinarily, the compiler omits checking the binding of the variable with `boundp` if the variable reference appears within the scope of a special binding of that variable, or if the reference appears in a file that is being compiled with `compile-file` and appears after a `defvar` or `defparameter` form that defined that variable.

`:allow-constant-substitution`
: When this value is true or a function that returns true, the compiler is allowed to substitute the value of a named constant for a reference to the constant. The default value is a function of three arguments: a symbol that names a constant, the value of that constant, and the current lexical environment. The function ignores those arguments and returns `t`.

`current-compiler-policy` [*Function*]

Syntax `current-compiler-policy`

Description The `current-compiler-policy` function returns the current `compiler-policy` used by interactive compilation.

`set-current-compiler-policy` [*Function*]

Syntax `set-current-compiler-policy &optional` *policy*

Description The `set-current-compiler-policy` function sets the default `compiler-policy` used by interactive compilation to *policy*. If *policy* is `nil` or unsupplied, a copy of the default compiler policy is used.

Argument *policy* A compiler policy.

`current-file-compiler-policy` [*Function*]

Syntax `current-file-compiler-policy`

Description The `current-file-compiler-policy` function returns the current `compiler-policy` used by file compilation.

`set-current-file-compiler-policy` [*Function*]

Syntax `set-current-file-compiler-policy &optional` *policy*

Description The `set-current-file-compiler-policy` function sets the default `compiler-policy` used by file compilation to *policy*. If *policy* is `nil` or unsupplied, a copy of the default compiler policy is used.

Argument *policy* A compiler policy.

`ignore-if-unused` [*Declaration*]

Syntax `ignore-if-unused`

Description The `ignore-if-unused` declaration behaves the same way as `ignore`, but does not signal a warning if the variable is used. This declaration is usually used in macroexpansions.

Listener Variables

The following variables are related to the behavior of the Lisp Listener.

`*top-listener*` [*Variable*]

Description The `*top-listener*` variable the Listener of the current process.

`*listener-default-font-spec*` [*Variable*]

Description The `*listener-default-font-spec*` variable specifies which font is used when new Listener windows are opened. The initial value is `("Monaco" 9 :PLAIN)`.

`*listener-window-position*` [*Variable*]

Description The `*listener-window-position*` variable specifies a point indicating the position used when new Listener windows are created. The user may set this variable.

Example

Here is an example of setting this variable. (The `point-string` and `make-point` functions are documented in "MCL functions relating to points" on page 71.)

```
? *listener-window-position*
19660850
? (point-string *listener-window-position*)
"#@(50 300)"
? (setf *listener-window-position* (make-point 20 300))
19660820
```

`*listener-window-size*` [*Variable*]

Description The `*listener-window-size*` variable specifies a point indicating the size used when new Listener windows are created. The user may set this variable.

`*terminal-io*` [*Variable*]

DescriptionThe initial binding of this stream prints to the Listener which is the value of `*top-listener*.` If there is no Listener, any attempt to write to `*terminal-io*` creates a new Listener.

Patches

The following functions are used to load MCL patches.

`load-patches` [*Function*]

Syntax `load-patches &optional` *source-dir all*

Loads some or all of the compiled files in the patch file directory, and optionally sets a patch version number which determines the version specified in the `vers 1` resource created when `save-application` is called. The patches directory is a folder whose name is of the form “Patches *x.y*”, where *x* and *y* are the major and minor version numbers of MCL (for example, “Patches 3.1b1” or “Patches 4.0”).

If *all* is `nil`, only new patches are loaded. A patch is considered to be new if its name (excluding file extension) ends in “p*n*”, where *n* is a number greater than the current patch version. The current patch version is determined from the `vers 1` resource. The patch version number will be set to the highest value of *n* encountered, and is returned by `load-patches` if set.

If *all* is true, all patches are loaded and the patch version is not set.

Arguments *source-dir* The directory containing the patch file directory. The default value for this argument is the value of the form `(full-pathname "ccl:" :no-error nil)`.

all — If true, load all compiled files in alphabetical order, and don't set the patch version number. If `nil` load only compiled files with names as specified above, and set the patch version number. The default is `nil`.

`load-all-patches optional source-dir` [*Function*]

Syntax `load-all-patches &optional` *source-dir*

Loads all compiled files from a patches directory by executing `(load-patches` *source-dir* `t)` and resets the current patch version to `nil`. Returns `nil`.

Arguments *source-dir* — The directory containing the patch file directory. The default value for this argument is the value of the form `(full-pathname "ccl:" :no-error nil)`.

Miscellaneous MCL expressions

The following MCL expressions provide miscellaneous useful functionality not in Common Lisp.

`*.fasl-pathname*` [*Variable*]

Description The `*.fasl-pathname*` variable contains the default pathname extension to use for compiled files. In MCL 4.0 it is`#P".pfsl"` and in MCL 3.1 it is`#P".fasl"`.

`*always-eval-user-defvars*` [*Variable*]

Description The `*always-eval-user-defvars*` variable determines how Macintosh Common Lisp treats the evaluation of `defvar`.

If an entire buffer or a selection in a buffer is evaluated, `defvar` is never equivalent to `defparameter`.

If the value of this variable is true, then `defvar` is equivalent to `defparameter` when evaluated as a single expression from a Fred buffer or when typed to the Listener.

If the value of this variable is `nil` *(the default), then* `defvar` acts in the normal Common Lisp way (see *Common Lisp: The Language,* pages 86–87).

`require-type` [*Function*]

Syntax `require-type` *argument type*

Description The `require-type` function is like the Common Lisp `check-type` macro, except that it returns a value rather than using `setf`, and so can be done entirely not inline. If *argument* is of the same type as *type,* `require-type` returns *argument*. If not, it signals an error.

Arguments

argument	Any argument.
type	A type.

Example

```
? (require-type (front-window) 'window)
#<LISTENER "Listener" #x42DDB1>
? (require-type (target) 'listener)
> Error: value #<WINDOW "Inspector Central" #x4618A1> is not
of the expected type LISTENER.
```

`structure-typep` [*Function*]

Syntax `structure-typep` *form type*

Description The `structure-typep` function returns `t` if *form* is of the given structure type *type* or if it includes *type.* Otherwise it returns `nil`. This function is used by `defstruct` predicates.

Arguments

form	Any form.
type	A type.

`structurep` [*Function*]

Syntax `structurep` *form*

Description The `structurep` function returns `t` if the given object is a named structure; otherwise it returns *nil*.

Argument *form* An MCL form.

loading-file-source-file [*Variable*]

Description

The `*loading-file-source-file*` variable is bound to the namestring of the file containing the source code while `load` is loading a source code file or a `fasl` file. The value of this variable is either the name of the file itself if it is a source code file, or the name of the file that was compiled to create it if it is a `fasl` file. Its default value is `nil`.

hide-windoids-on-suspend [*Variable*]

Description

Controls whether windoids are hidden automatically when MCL is suspended. If set to true, they are hidden on suspend events and shown on resume events. The default value is `t`.

machine-owner [*Function*]

Syntax

`machine-owner`

Description

returns the “Owner Name” from the Sharing Setup control panel if it can be determined, otherwise returns `"unspecified"`.

pascal-full-longs [*Variable*]

Description

Controls whether or not `defpascal` functions use bignums to get full 32-bit (signed) arguments. If set to true, bignums are used. The default value is `nil.`

preferences-file-name [*Variable*]

Description

The name of the preferences file, normally "MCL Preferences".

tool-back-color [*Variable*]

Description

Controls the background color of the tools dialog boxes. It can be set to any value returned by `user-set-color` or any value suitable as an argument to `make-color`.

`*tool-line-color*` [*Variable*]

Description Controls the color of the lines dividing the tools dialog boxes. It can be set to any value returned by `user-set-color` or any value suitable as an argument to `make-color`.

`gestalt` [*Function*]

Syntax `gestalt` *selector* `&optional` *bitnum*

Description If bitnum is supplied and non-`nil`, `gestalt` returns true if that bit is set in the attribute flags; if `nil` or not supplied, `gestalt` returns the attribute flags as usual.

Appendix B:

Workspace Images

Contents

This appendix describes a utility that you can use to save images of running MCL environments. These images can be customized MCL development environments or prototype stand-alone applications.

The Image Facility

This chapter describes a utility that you can use to save images of running MCL environments. These images can be customized MCL development environments or prototype stand-alone applications.

◆ *Note*: The MCL license agreement does not allow redistribution of applications created with the image facility. The MCL Redistribution Kit is used for creating distributable applications. It includes a number of additional tools for optimizing these stand-alone applications.

To create an image, you first arrange your Lisp environment just as you want it, by loading files, etc. You then select the Save Application… or Extensions/Save Image… command from the Tools menu, or call the `save-application` function.

Some state cannot be saved and restored automatically. In particular, data on the Macintosh heap, and pointers to such data, cannot be saved and restored. Such data must be disposed of in the process of creating the image, and then recreated when the image is launched. `*lisp-cleanup-functions*`, `*save-exit-functions*`, `def-load-pointers`, and `*lisp-startup-functions*` are used for this purpose.

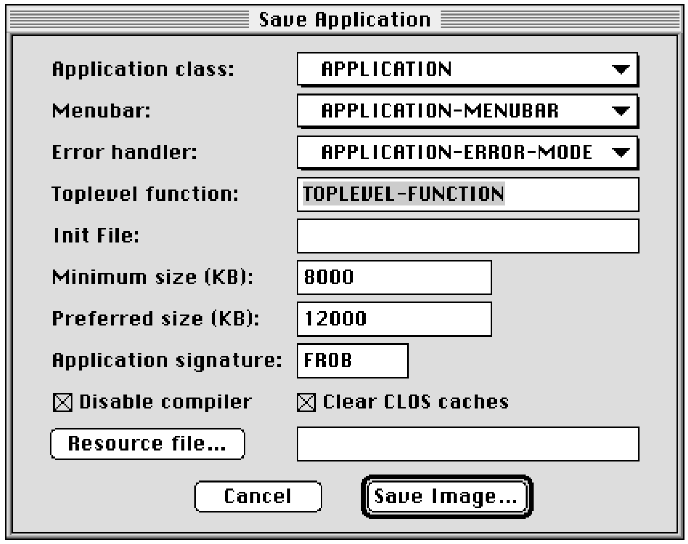
In addition to using the image facility, you can customize your Lisp environment with the Preferences dialogs and with an init file.

An image which is not intended to be used as a Lisp development environment will probably want to specify a different application class (as described in "Application class and built-in methods" on page 394), and may also want to define a new `toplevel-function` method for that class. For example, the built-in `toplevel-function` method for the `lisp-development-system` application class loads an init-file and MCL preferences file, actions which are likely inappropriate for your application.

The Save Application tool

The Save Application tool provides a graphical interface and slightly different options than the `save-application` function. When the Save Image button in selected, an image is saved and MCL returns to the Finder.

■ **Figure 1-10** The Save Application dialog box



- The application class provides a value for `*application*` in the image. `lisp-development-system` is for making customized MCL environments. Other subclasses of `application` are used for stand-alone application prototypes. See "Applications and Apple Events" on page 392 for a description of these variables and classes.
- The menubar allows you to specify the menubar to install when the image is restarted. The default is the MCL menubar, but if you have defined your own menubar using the Interface Toolkit, you can add it.
- The error handler specifies the response to unhandled errors. The choices are to pop up an error dialog, to pop up a Listener, or to Quit the application. The error dialog is the default.
- The toplevel function is a function to call when the image has been restarted and Macintosh pointers have been restored. In general, this should be a function of no arguments. As a special case, if `toplevel-function` is specified, it will be called on the current application class and init-file when the image is restarted.
- The init file specifies the name of a file to load when the image is restarted.
- The minimum size and preferred sizes specify components of the `size` resource, which control how much memory the Finder will allocate for the image when it is restarted. MCL calculates and suggests default values for these numbers.
- The Application signature signature is used by the OS to identify the application, associate it with icons and document files, etc.

- Disable compiler, if checked, disables the MCL compiler in the image. This allows you to test whether your application can run without the compiler, while still allowing you to use the development tools for debugging. This option is useful when testing and preparing an application for standalone distribution.
- Clear CLOS caches, if checked, flushes the CLOS caches before saving the image. This makes the image somewhat smaller, and makes it restart somewhat more quickly. On the downside, the image will run more slowly when it first restarts, as the caches get refilled.
- The resource file specifies a file of resources which should be copied into the resource fork of the image (along with the standard MCL resources) when the image is saved.

See the documentation of the `save-application` function page 677 for more details on the saving of process involved in saving an image.

The Save Image Command

The Save Image… command on the Extensions submenu of the Tools menu provides a shortcut for saving an image of a Lisp session. It may be used when you want to save out a snapshot of your current Lisp session quickly and without much customization.

The command prompts the user to choose a file name for the saved image. It then calls `save-application` with the following arguments:

pathname	The file name chosen by the user.
	`:size` A value computed by using the largest numbered SIZE resource values, and adding the amount of memory that has been consumed since the first MCL extension was loaded.

Forms Related to Images

The following functions, variables, and macros are used to programmatically create saved images, and to control the exact behavior of images when the are created and restarted. See also the description of `toplevel-function` on page 396.

`save-application` [*Function*]

Syntax `save-application` *pathname* `&key :toplevel-function :creator :excise-compiler :size :resources :init-file :clear-clos-caches :menubar :error-handler :application-class :memory-options`

Description The `save-application` function creates a stand-alone image containing the functionality of the current Lisp environment. When `save-application` is finished, Macintosh Common Lisp exits to the Finder.

Before saving an image, `save-application` closes all windows, takes down and remembers the current menu bar, and disposes of other pointers to the Macintosh heap. It then executes all the functions on the list `*save-exit-functions*`. Finally, Macintosh Common Lisp performs a garbage collection and saves the heap image.

You can add functions to the list `*save-exit-functions*`. You may wish to do this if you want to save and restore a certain state in a particular way.

When a heap image is restarted, Macintosh Common Lisp restores Macintosh pointers used by the system, resets the logical hosts `"ccl:"` and `"home:"`, and reinitializes some system configuration variables. Then it runs all the functions specified by `def-load-pointers` in the order they were specified.

Arguments

pathname A pathname for the image to be created. If a file with that name already exists, Macintosh Common Lisp deletes it before `save-application` is performed.

`:toplevel-function`
A function of no arguments to call when the image restarts. The default is a function which calls `toplevel-function` on the current application class and init-file.

`:creator` The `mac-file-creator` os-type for the saved application. The default is `:CCL2`. Set it to something else if you do not want the Finder to consider your saved application the creator of all your MCL files.

`:excise-compiler`
An argument specifying whether to disable the compiler in the resulting application. If the value of this keyword is true, the compiler is disabled. Its default value is `nil`.

Note: Code that calls external functions needs to be compiled if it is to run in an application with the compiler excised. Attempting to intrepret such functions will invoke the compiler, and error if the compiler is not present.

`:size` A size specification, which is either a nonnegative integer or a list of two nonnegative integers: `(`*preferred-size minimum-size*`)`. If present, this argument sets the preferred and minimum partition sizes (in bytes) in the application's `SIZE (-1)` resource. Any `SIZE(0)` resources that the Finder may have added to Macintosh Common Lisp are not copied to the resulting application.

`:resources` A list of resource specifications, where each resource specification is a list of the form `(`*data resource-type resource-id* `&optional` *resource-name*`)`.
If `:resources` is specified, any resources matching *resource-type* and *resource-id* are not copied from Macintosh Common Lisp to the resulting application. If *data* is non-`nil`, this specification causes *data* to be added as a resource of the specified type and ID. A resource specification can also be a symbol or function, in which case `funcall` will be run on it with a single argument, the name of the file being saved. When the function is called, the current resource file will be the resource file of the application being saved.

`:init-file` An argument specifying the pathname of an `init` file to load when the MCL image is started, or `nil` (the default). If the argument is `nil`, the result of calling `(application-init-file *application*)` is used. `application-init-file` returns `"init"` when called on `lisp-development-environment`; it returns `nil` when called on `application`. The `init` file need not be in the same folder as Macintosh Common Lisp; you can specify any pathname you wish.

`:clear-clos-caches`
An argument specifying whether caches are cleared when the application is saved. The default value is true.

`:menubar` A list of menu objects. `set-menubar` will be called with the specified menubar before the image is saved

`:error-handler`
one of the keywords `:dialog` `:listener` `:quit` `:quit-quietly`. If this argument is specified, the hide-listener-support module will be loaded, and the application-error method will perform the specified action when errors occur.

`:application-class`
A class or class name. `*application*` will be set to an instance of the specified class before the image is saved

`:memory-options`
A list of keyword/value pairs spcifying the contents of the `LSIZ 1` resource for the application. The following keys and default values are supported:

```
:mac-heap-minimum       102400
:mac-heap-maximum       409600
:mac-heap-percentage    5
```

`:low-memory-threshold`	`24576`
`:copying-gc-threshold`	`2147483648`
`:stack-maximum`	`184320`
`:stack-minimum`	`32768`
`:stack-percentage`	`6`

MCL 4.0 ignores the `:copying-gc-threshold`, `:stack-minimum`, `:stack-maximum`, and `:stack-percentage` arguments. They are stored in the `LSIZ` resource, but never used. MCL 3.0 ignores the `:copying-gc-threshold` argument.

Example

Here is an example of saving an application using the `:resources` keyword.

```
(eval-when (:compile-toplevel :execute :load-toplevel)
  (require :resources))

(defun copy-my-apps-resources (resource-file)
  (declare (ignore resource-file))
  (let ((refnum (#_CurResFile)))
    (with-open-resource-file (my-refnum "My App.r")
      ...)))

(save-application "My App" :resources
                           'copy-my-apps-resources)
```

Removing Macintosh pointers

An important restriction on saved images is that no data on the Macintosh heap is preserved across saves and restarts. When you save an application, any pointers or handles to the Macintosh heap become invalid. For this reason, you should dispose of all Macintosh handles and pointers before doing `save-application`.

If your program maintains pointers to the Macintosh heap, you should deallocate these with a function included on the list `*save-exit-functions*`. You can then reinitialize the pointers and handles with functions specified by `def-load-pointers`.

◆ *Note:* Leftover Macintosh pointers in a heap image can cause system crashes and other erratic behavior.

The `def-load-pointers` macro can be used to allocate memory on the heap during startup.

`*lisp-cleanup-functions*` [*Variable*]

Description The `*lisp-cleanup-functions*` variable contains a list of functions of no arguments on which `funcall` is run just before Macintosh Common Lisp exits (via `quit` or `save-application`). These functions are called just after the windows are closed.

When saving an application, the functions in `*lisp-cleanup-functions*` are run, then the functions in `save-exit-functions*` are run.

`*save-exit-functions*` [*Variable*]

Description The `*save-exit-functions*` variable contains a list of functions to be called when an image is saved. These functions should perform any preparation necessary for the image saving. The functions are called in the order in which they appear in
the list.

When saving an application, the functions in `*lisp-cleanup-functions*` are run, then the functions in `*save-exit-functions*` are run.

`def-load-pointers` [*Macro*]

Syntax `def-load-pointers` *name arglist* `&body` *body*

Description The `def-load-pointers` macro is usually used to allocate memory on the Macintosh heap. It associates *name* with `#'(lambda` *arglist . body*`)` in a list. If *name* is already on the list, the macro replaces it. If it is not, `def-load-pointers` adds *name* and its function to the list and runs `funcall` on it.

When Macintosh Common Lisp starts up, it calls the functions specified by `def-load-pointers` in the order in which they were specified on the list. This occurs before the `init` file is loaded.

Arguments *name* The name to associate with a function.

arglist	The argument list of the function. The function is called with no arguments, hence this argument should always be `nil`.
body	The body of the function.

`*lisp-startup-functions*` [*Variable*]

Description The `*lisp-startup-functions*` variable contains a list of functions of no arguments on which `funcall` is run after Macintosh Common Lisp starts, just before it enters the top-level function (usually the Listener's read loop). The functions contained in `*lisp-startup-functions*` are run after the functions specified by `def-load-pointers` and before the `init` file is loaded. The functions are called in reverse order from the order in which they appear in the list.

Appendix C:

SourceServer

Contents

This appendix describes an MCL interface to SourceServer, a source code control system.

SourceServer

SourceServer is an application that allows development environments and other applications to access MPW Projector project management capabilities via Apple Events. Development environments and applications have access to the full functionality of Projector including creating project databases, as well as checking in and out individual files. MCL, MPW and other development environments can share the same project database using SourceServer. A copy of SourceServer can be found in the "Developer Essentials" folder on the MCL 3.9 CD. The latest version is always available on E.T.O. (Essentials - Tools - Objects), a CD-ROM subscription series distributed by Apple through APDA. See MPW Projector documentation for an overview of what Projector and, by implication, SourceServer are all about.

The interface to SourceServer is adapted from a version in use by the Dylan team at Apple. It was created for a demo at WWDC and included in "Other goodies from Apple" on the MCL 2.0 CD. It has been improved by the folks at RSTAR, Inc. and by Digitool. It is an application in progress, but it is nonetheless useful. The best tested and most stable work style for SourceServer is to check files out read-only and then make them "modify read only" on your local disk,.

Setting up SourceServer

There are two files in the SourceServer folder that you should customize. The file `initialize-user.lisp` sets your user name and initials and the logical pathname translations for the SourceServer database and the file hierarchy on your local disk. The logical host for the SourceServer database is `SSRemote`. The host for the local files is `SSLocal`. The file `initialize-projects.lisp` sets the list of projects. The projects do not need to exist initially, but an error occurs if you attempt to mount a project that is not in the database.

To try out SourceServer, edit the two files, then load or execute the file `load-sourceserver.lisp` in the SourceServer folder.

The version of the SourceServer application in this folder is 1.0.1; it can reside anywhere on your system.

The SourceServer menu

The first four menu items on the SourceServer menu apply to the active (front most) window. If the active window corresponds to a file in the local directory of one of `*all-projects*` some or all of these menu items are enabled.

Checkout Active is enabled if the corresponding project is mounted and the file is read-only on the local disk. It checks out the file for modification, preventing other users from modifying it.

Checkin Active is enabled if the project is mounted and the file is modifiable on the local disk. It checks in the file and makes it read-only.

ModifyRead0nly Active is enabled if the file is read only. It makes the file modifiable on the local disk.

Other Active is always enabled and provides a variety of other options.

Mount Projects allows you to mount all projects or just some selected projects. Use shift click in the dialog to select multiple projects.

New Project creates a new project. A dialog asks for the name of the database file for the project. If the corresponding local directory exists in `SSLocal:`, that is used. If it does not exist, a dialog lets you create the local directory.

Update Current Project gets the most recent versions of all the project files from the SourceServer database. If any of the files are modifiable on the local disk, a dialog asks if you want to move the local files to a merge directory or to specify other action. It is recommended to choose "merge."

The modified files are moved to a directory like `hd:my-project merge0:` and the newer ones replace them in the project directory. You can use **Merge Directories** to merge your changes with the newer files.

Merge Directories is used to merge a selected file in one directory with the correspondingly named file in another directory. In the dialog `Main dir`: is generally the local project directory, for example. `SSlocal:project;` and `Merge dir:` is the directory containing those files that were moved to a merge directory, for example, `SSlocal:project Merge0`;. Choosing the **List Files** button lists the contents of the merge directory. Select a file from the list, then click **Mergge File.** Both versions of the selected file open and dialog that controls the merge is displayed. **Note:** Be sure you have a file selected before you click **Merge File.**

Merge Directories can be used to merge any directories not just those containing project files.

- To add a single file to a project, make its window active and choose **Checkin Active**. A dialog asks if you want to add it to the project.
- To add several files to a project use **New Project Files** in the submenu of **Other File**. This brings up a dialog that lists all the files in the chosen directory. The filter at the top can be used to select a subset of the files. The filter string is passed to the directory function so, for example, `*.lisp` selects all the `.lisp` files. Use the Shift and Command keys to select and deselect more than one file.
- To delete a file from a project choose **Delete** in the submenu of **Other File**. This just removes the file from the project database. It does not delete it from your local disk.

Notes

If you attempt to modify a fred-window for a read-only file, a dialog asks whether you want to make the buffer (and file) modifiable. This happens even if SourceServer is not loaded as long as the SourceServer folder is in the expected place in the MCL folder.

It may be the case that whereas this SourceServer interface supports project hierarchies, MPW does not. So switching between MPW and MCL SourceServer for source control of a hierarchical project may not work.

Appendix D:

QuickDraw Graphics

Contents

This appendix documents a set of CLOS methods that create an interface with QuickDraw. The code that implements these functions serves as an extended example of CLOS programming and is included as an example file. You should read it if you plan to use QuickDraw extensively in Macintosh Common Lisp, or if you are planning to create your own high-level methods to interface with traps. However, you may prefer to use the traps functionality documented in Chapter 16: OS Entry Points and Records.

This appendix assumes some familiarity with the various discussions of QuickDraw in *Inside Macintosh*. You should also be familiar with the MCL implementation of points, as discussed in "Points" on page 70 and with the MCL implementation of records, described in Chapter 16: OS Entry Points and Records.

QuickDraw in Macintosh Common Lisp

Macintosh Common Lisp allows you to call QuickDraw traps directly (see Chapter 16: OS Entry Points and Records). The interface routines support all of the functionality found in the original (64K ROM) Macintosh packages.

The arguments to the MCL QuickDraw functions generally parallel the arguments to the Pascal QuickDraw functions given in *Inside Macintosh*. In several places Pascal functionality has been extended by taking advantage of the optional arguments provided by Macintosh Common Lisp. In some places the order of arguments has been changed to make the mapping of the optional arguments more effective. Last *var* arguments have sometimes been eliminated, and instead a value is returned.

Some QuickDraw functions may be performed only as methods on views. The view must be a window or must be contained in a window. The functions depend on the existence of a graphics port (GrafPort). All other functions may be performed globally.

Before calling any of the functions described in this appendix, you must load the QuickDraw file, which is in your MCL Library directory.

Windows, GrafPorts, and PortRects

All drawing on the Macintosh computer takes place inside **GrafPorts,** the structures upon which a program builds windows. (See *Inside Macintosh* for a complete description of GrafPorts.)

In low-level Macintosh drawing, several levels of initialization are used to set up windows and GrafPorts for drawings. Once they have been created, you must keep track of the current GrafPort when you do any drawing.

This process is simplified for the graphics routines given in this appendix.

- When you create a window, an initialized GrafPort is automatically created.
- Drawing commands are defined as methods on views, which must be windows or contained in windows; when you call a method to perform one of the commands in a window, GrafPorts are set and restored automatically with `with-focused-view`.

- Drawing inside windows is automatically cropped to fit inside the window and the portions of the window that are visible (that is, not covered by other windows).

Drawing is also affected by the clip region (described later) and the PortRect. The **PortRect** is an arbitrary rectangle designating the outermost bounds in which drawing can occur. (See Figure D-2.) It supplies a frame of reference for the window. The default PortRect is infinitely large; you can set it using low-level calls (although you usually won't need to worry about this at all).

Since all the drawing functions use `with-focused-view`, you can speed up drawing considerably if you wrap `with-focused-view` around all calls to multiple drawing functions.

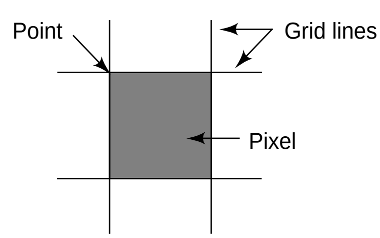
Points and rectangles

In QuickDraw, points are specified by two coordinates, the horizontal coordinate (called *h*) and the vertical coordinate (called *v*). The horizontal coordinate increases as it moves to the right, and the vertical coordinate increases as it moves down. The upper-left corner of a window (called the *origin*) is usually the point (0,0), but the origin may be changed by using the `set-origin` generic function.

Points are stored as encoded integers. Points lie at the intersection of two grid lines on the QuickDraw plane. Note that points and pixels are not equivalent. The point associated with a given pixel is at the upper-left corner of the pixel. (See Figure D-1.)

See "Points" on page 70 for a general description of the MCL point data format.

■ **Figure D-1** Location of point at upper-left corner of pixel

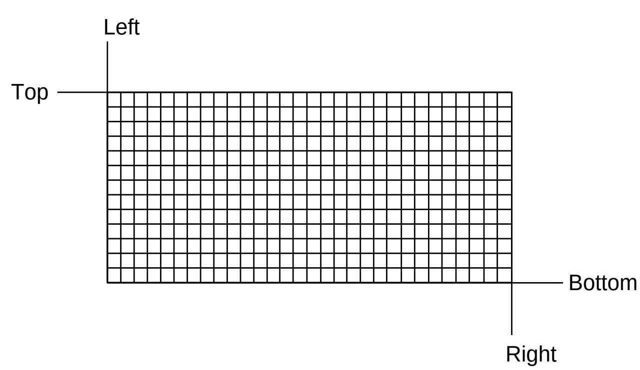


A Macintosh computer stores rectangles as 8-byte records. (Records are blocks of non-Lisp data stored on the Macintosh heap or on the stack; see Chapter 16: OS Entry Points and Records for details.)

Rectangle records can be thought of as two points (upper-left and lower-right), or four edges (left, top, right, and bottom). Allocating memory for rectangle records can be inefficient, and so Macintosh Common Lisp provides several forms of memory allocation. The `make-record` function is used to allocate memory for long-lived rectangles, and the `rlet` function is used to allocate records for short-lived rectangles (see Chapter 16: OS Entry Points and Records for details).

For many of the MCL QuickDraw functions that use rectangles, you do not need to allocate rectangle records explicitly at all. The rectangles can be specified as four coordinates, or as two points, or as a rectangle record (see Figure D-2). In general, if you use a rectangle only once, it is all right to pass it as two points or four coordinates. However, if you use it several times, it is more efficient to create and pass an actual rectangle record.

■ **Figure D-2** A PortRect



When alternative forms of a point or a rectangle are accepted as arguments, the flexible argument appears last. This order prevents ambiguity about which argument is which and explains why the order of arguments sometimes differs from the order given in *Inside Macintosh*.

Window state functions

The following functions operate on the window containing the view asked to perform a function.

`origin` [*Generic function*]

Syntax `origin` (*view* `view`)

Description The `origin` generic function returns the coordinates of the upper-left point in the window's content region. This is usually `#@(0 0)` but may be different if it is set by user-written code.

Argument *view* A window or a view contained in a window.

`set-origin` [*Generic function*]

Syntax `set-origin` (*view* `view`) *h* `&optional` *v*

Description The `set-origin` generic function sets the origin to the point specified by *h* and *v*.

The contents of the window are not moved; only future drawing is affected.

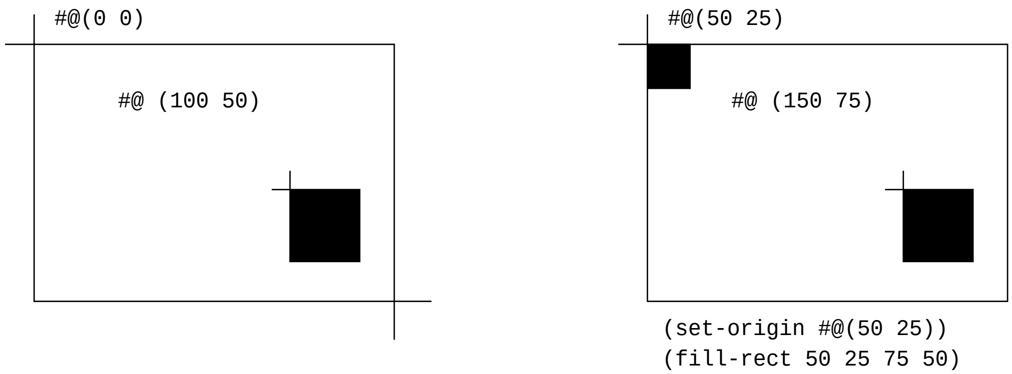
Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If the value of *v* is `nil` (the default), *h* is assumed to represent a point.

Example

Figure D-3 shows an example of using `set-origin`. In that figure you can see that rectangles can be passed as two points, four coordinates, or one rectangle record.

■ **Figure D-3** Multiple methods of passing rectangles



A clip region allows drawing in a window to be restricted to an arbitrary region. Drawing occurs only in the clip region. The default clip region is arbitrarily large, so no clipping takes place. Note that regions must be explicitly disposed of; they are not subject to automatic garbage collection.

`clip-region` [*Generic function*]

Syntax `clip-region` (*view* `view`) `&optional` *save-region*

Description The `clip-region` generic function returns the window's current clip region.

Arguments

view	A window or a view contained in a window.
save-region	The region in which the window's clip region is returned; otherwise, the clip region is returned in a newly allocated region.

`set-clip-region` [*Generic function*]

Syntax `set-clip-region` (*view* `view`) *new-region*

Description The `set-clip-region` generic function sets the window's clip region to *new-region* and returns *new-region*.

Arguments

view	A window or a view contained in a window.
new-region	A region.

See the “Regions” on page 718 for functions that allocate and manipulate regions.

`clip-rect` [*Generic function*]

Syntax `clip-rect` (*view* `view`) *left* `&optional` *top right bottom*

Description The `clip-rect` generic function makes the window’s clip region a rectangular region equivalent to the rectangle determined by *arg*. It returns `nil`.

Arguments

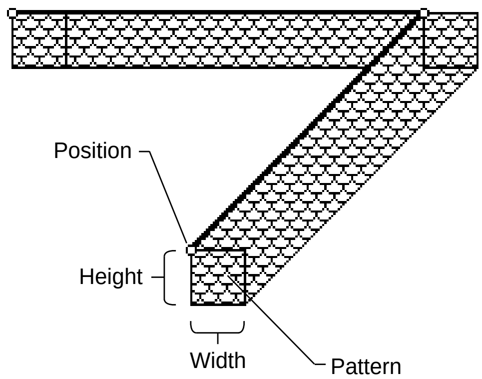
view A window or a view contained in a window.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Pen and line-drawing routines

Every window has its own pen. The state of the pen determines how drawing occurs in the window. For example, if the pen is hidden, drawing commands have no effect on the screen. In addition to its state as hidden or shown, a pen has a size (height and width), a position in the window, and a pattern used for drawing. (See Figure D-4.)

■ **Figure D-4** Attributes of a graphics pen



The following functions operate on the window containing the view asked to perform a function.

`pen-show` [*Generic function*]

Syntax `pen-show` (*view* `view`)

Description The `pen-show` generic function shows the pen. Drawing occurs only when the pen is shown.

Argument *view* A window or a view contained in a window.

`pen-hide` [*Generic function*]

Syntax `pen-hide` (*view* `view`)

Description The `pen-hide` generic function hides the pen. If the pen is hidden, no drawing occurs.

Argument *view* A window or a view contained in a window.

`pen-shown-p` [*Generic function*]

Syntax `pen-shown-p` (*view* `view`)

Description The `pen-shown-p` generic function returns `t` if the pen is shown and `nil` if the pen is hidden.

Argument *view* A window or a view contained in a window.

---

pen-position [*Generic function*]

Syntax `pen-position` (*view* `view`)

Description The `pen-position` generic function returns a point corresponding to the pen position in local coordinates.

Argument *view* A window or a view contained in a window.

---

pen-size [*Generic function*]

Syntax `pen-size` (*view* `view`)

Description The `pen-size` generic function returns the current pen size as a point (expressing a width and height).

Argument *view* A window or a view contained in a window.

---

set-pen-size [*Generic function*]

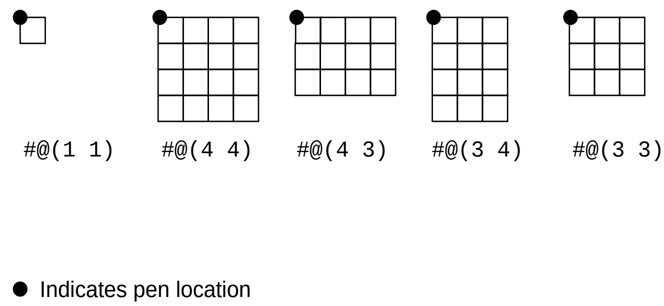
Syntax `set-pen-size` (*view* `view`) *h* `&optional` *v*

Description The `set-pen-size` generic function sets the pen size to the point indicated by *h* and *v*. Figure D-5 shows QuickDraw pen sizes.

Arguments

view	A window or a view contained in a window.
h	The width of the new pen size (or a point representing the width and height, if *v* is not given).
v	The height of the new pen size.

■ **Figure D-5** QuickDraw pen sizes



pen-pattern [*Generic function*]

Syntax `pen-pattern` (*view* `view`) `&optional` *save-pattern*

Description The `pen-pattern` generic function returns the window’s pen pattern.

Arguments

view A window or a view contained in a window.

save-pattern A pattern record; the pattern is returned in this record. If *save-pattern* is not given, a new pattern record is allocated to hold the returned pattern.

set-pen-pattern [*Generic function*]

Syntax `set-pen-pattern` (*view* `view`) *new-pattern*

Description The `set-pen-pattern` generic function sets the window’s pen pattern.

Arguments

view A window or a view contained in a window.

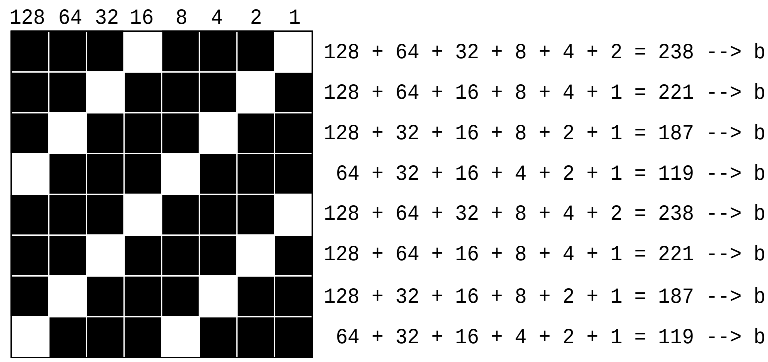
new-pattern A pattern record.

A pattern is stored as a 64-bit block of memory (see Figure D-6). The definition of a pattern record allows patterns to be accessed as 8 bytes or four words.

Pattern records, like all records, continue to take up space on the Macintosh heap until they are explicitly disposed of.

```
(defrecord pattern
  (variant ((b0 byte) (b1 byte) (b2 byte) (b3 byte)
            (b4 byte) (b5 byte) (b6 byte) (b7 byte))
           ((w0 integer) (w1 integer)
            (w2 integer) (w3 integer))))
```

■ **Figure D-6** Pen pattern stored as a 64-bit block of memory



Macintosh Common Lisp stores five patterns as constants: `*white-pattern*`, `*black-pattern*`, `*gray-pattern*`, `*light-gray-pattern*`, and `*dark-gray-pattern*`.

Pen modes affect the way drawing occurs in the window. They provide a logical mapping between the current state of pixels in the window and the state of the pixels being drawn.

`pen-mode` [*Generic function*]

Syntax `pen-mode` (*view* `view`)

Description The `pen-mode` generic function returns a keyword indicating the window's current pen mode.

Argument *view* A window or a view contained in a window.

`set-pen-mode` [*Generic function*]

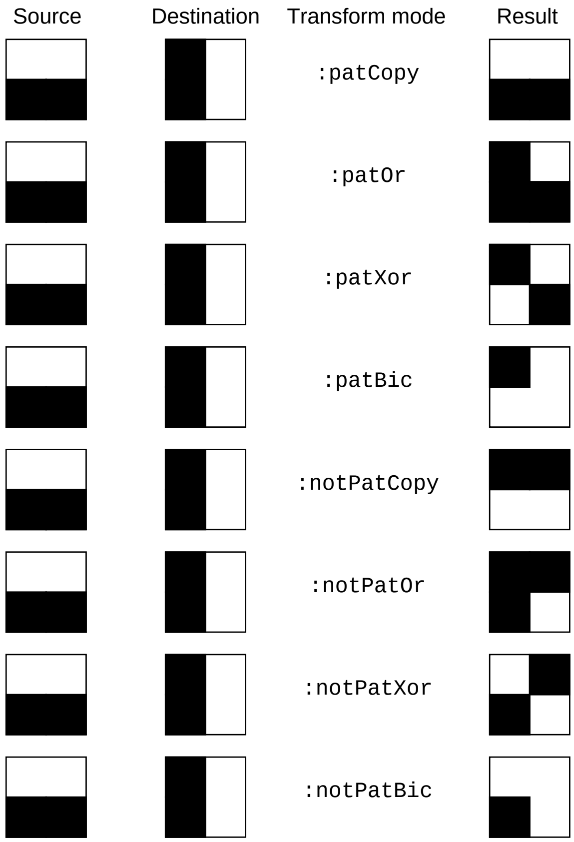
Syntax `set-pen-mode` (*view* `view`) *new-mode*

Description The `set-pen-mode` generic function sets the window's current pen mode. (See Figure D-7.)

Arguments *view* A window or a view contained in a window.

new-mode The new pen mode. This value should be one of the following keywords: `:patCopy`, `:patOr`, `:patXor`, `:patBic`, `:notPatCopy`, `:notPatOr`, `:notPatXor`, `:notPatBic.`

■ **Figure D-7** Effect of pen modes on pixels being drawn



Pen-state records represent the pen mode as an integer. This integer is equal to the position of the corresponding pen-mode keyword in the list `*pen-modes*`. To translate an integer into a keyword, make the call (`elt *pen-state*` *mode-integer*). To translate a keyword into an integer, make the call (`position` *mode-keyword* `*pen-state*`).

Here is the definition of a pen-state record.

```
(defrecord PenState
(pnLoc point)
(pnSize point)
(pnMode integer)
(pnPat pattern))
```

`pen-state` [*Generic function*]

Syntax `pen-mode` (*view* `view`) `&optional` *save-pen-state*

Description The `pen-state` generic function returns the current pen state, a record containing the pen's location, size, mode (as an integer), and pattern.

Pen-state records, like all records, continue to take up space on the Macintosh heap until they are explicitly disposed of.

Arguments

view	A window or a view contained in a window.
save-pen-state	A pointer to a pen-state record; the returned state is stored in this record. If *save-pen-state* is not given, the pen state is returned in a newly allocated record.

`set-pen-state` [*Generic function*]

Syntax `set-pen-mode` (*view* `view`) *new-pen-state*

Description The `set-pen-state` generic function sets the window's pen state.

Arguments

view	A window or a view contained in a window.
new-pen-state	A pen-state record.

`pen-normal` [*Generic function*]

Syntax `pen-normal` (*view* `view`)

Description The `pen-normal` generic function sets the pen size to `#@(1 1)`, the pen mode to :`patCopy`, and the pen pattern to `*black-pattern*`. The pen location is not changed.

Argument

view	A window or a view contained in a window.

`move-to` [*Generic function*]

Syntax `move-to` (*view* `view`) *h* `&optional` *v*

Description The `move-to` generic function moves the pen to the point specified by *h* and *v* without doing any drawing. It returns the point to which the pen moved.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

`move` [*Generic function*]

Syntax `move` (*view* `view`) *h* `&optional` *v*

Description The `move` generic function moves the pen *h* points to the right and *v* points down without doing any drawing.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

`line-to` [*Generic function*]

Syntax `line-to` (*view* `view`) *h* `&optional` *v*

Description The `line-to` generic function draws a line from the pen's current position to the point represented by *h* and *v*.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

`line` [*Generic function*]

Syntax `line` (*view* `view`) *h* `&optional` *v*

Description The `line` generic function draws a line to a point *h* points to the right and *v* points down from the current pen position.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

Drawing text

Macintosh Common Lisp draws text in windows by using a window as an output stream. Drawing of text takes place starting at the current pen position using the window's current font, size, style, and mode. The initial pen position determines the placement of the lower-left corner of the first character drawn, and the pen is moved to the right the width of each character after it is drawn. Special characters, such as carriage returns and backspaces, have no effect.

When a window is created, its pen position is `#@(0 0)`. This means that any text drawn in it will be above the visible portion of the window until the pen position is lowered.

`stream-tyo` [*Generic function*]

Syntax `stream-tyo` (*view* `view`) *char*

Description The `stream-tyo` generic function draws *char* at the current pen position, in the current font, using the current text transfer mode. It then moves the pen to the right the width of the character. Because windows are streams, all stream output functions (such as `prin1`) can be performed on them. The `stream-tyo` function is not normally called directly but instead by stream output functions.

Arguments

view	A window or a view contained in a window.
char	A character.

Calculations with rectangles

The following functions do not draw; they simply perform calculations. They do not depend on a GrafPort, and so they are defined globally rather than as generic functions.

`offset-rect` [*Function*]

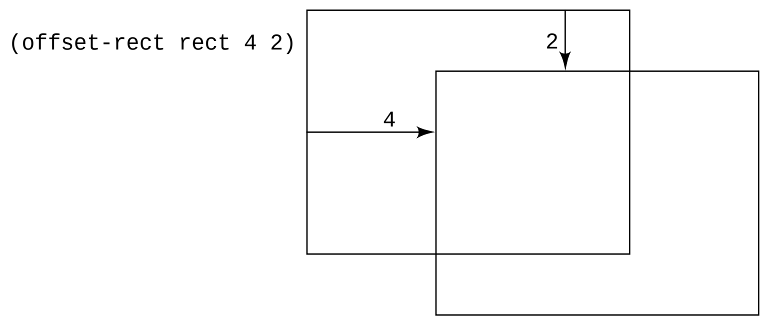
Syntax `offset-rect` *rectangle h* `&optional` *v*

Description The `offset-rect` function moves *rectangle h* to the right and *v* down. (See Figure D-8.) It returns the destructively modified rectangle.

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

■ **Figure D-8** Offset rectangle, with *h* equal to 4 and *v* equal to 2



`inset-rect` [*Function*]

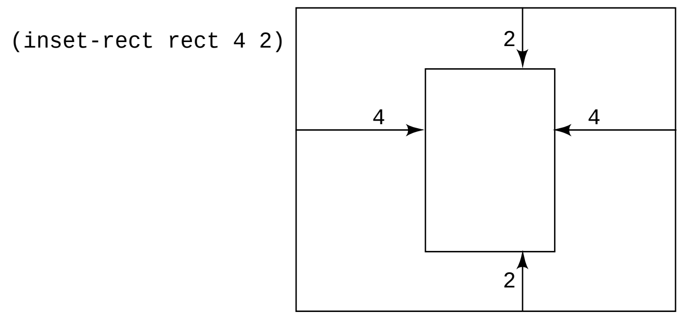
Syntax `inset-rect` *rectangle h* `&optional` *v*

Description The `inset-rect` function shrinks or expands *rectangle* by *h* and *v.* It returns the destructively modified rectangle. If *h* and *v* are positive, the left and right sides and the top and bottom move toward the center. If *h* and *v* are negative, the sides move outward. See Figure D-9.

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

■ **Figure D-9** Inset rectangle, with *h* equal to 4 and *v* equal to 2



`intersect-rect` [*Function*]

Syntax `intersect-rect` *rect1 rect2 dest-rect*

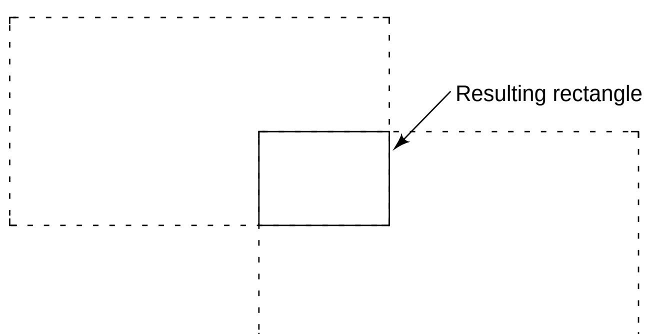
Description The `intersect-rect` function stores in *dest-rect* the rectangle created by the intersection of *rect1* and *rect2* and returns *dest-rect.* (See Figure D-10.)

A single rectangle may be passed as *dest-rect* and as *rect1* or *rect2*, making it unnecessary to allocate one extra rectangle.

Arguments

rect1	A rectangle.
rect2	A rectangle.
dest-rect	A rectangle record used to hold the intersection of *rect1* and *rect2*.

■ **Figure D-10** Rectangle resulting from the intersection of two others



`union-rect` [*Function*]

Syntax `union-rect` *rect1 rect2 dest-rect*

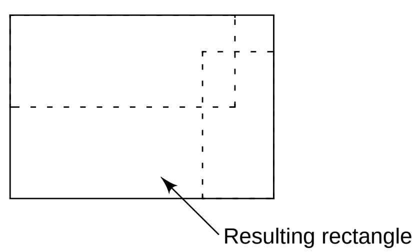
Description The `union-rect` function stores in *dest-rect* the smallest rectangle that encloses both *rect1 and rect2* and returns *dest-rect.* (See Figure D-11.)

A single rectangle may be passed as *dest-rect* and as *rect1* or *rect2*, making it unnecessary to allocate one extra rectangle.

Arguments

rect1	A rectangle.
rect2	A rectangle.
dest-rect	A rectangle record used to hold the rectangle enclosing *rect1* and *rect2*.

■ **Figure D-11** Smallest rectangle completely enclosing two others



`point-in-rect-p` [*Function*]

Syntax `point-in-rect-p` *rectangle h* `&optional` *v*

Description The `point-in-rect-p` function returns `t` if the point specified by *h* and *v* is inside *rectangle;* otherwise, it returns `nil`.

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

`points-to-rect` [*Function*]

Syntax `points-to-rect` *point1 point2 dest-rect*

Description The `points-to-rect` function stores in *dest-rect* the smallest rectangle that encloses both *point1* and *point2*, and returns *dest-rect*.

The `points-to-rect` function is useful when you have two corner points but don't know which one is the top left and which one is the bottom right.

Arguments

point1	A point that specifies one of the corners of the rectangle.
point2	A point that represents the other corner of the rectangle.
dest-rect	A rectangle record used to hold the result of the calculations.

`point-to-angle` [*Function*]

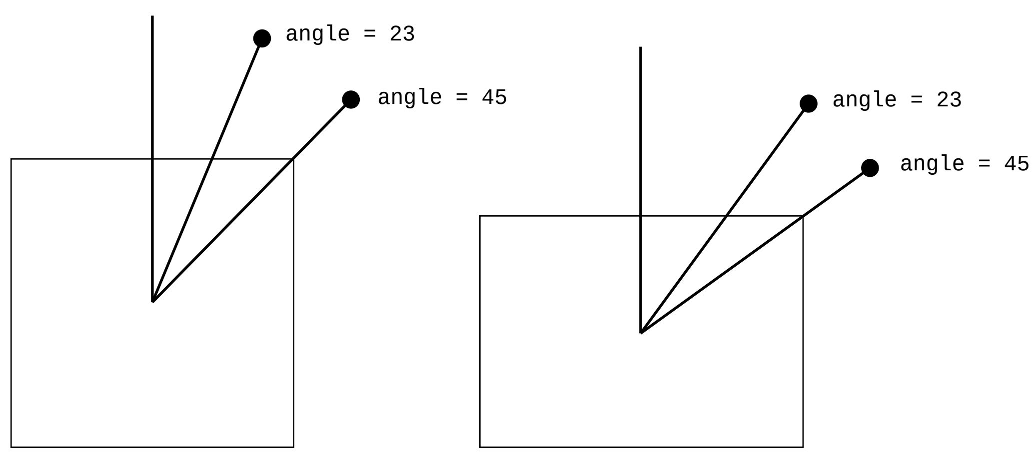
Syntax `point-to-angle` *rectangle h* `&optional` *v*

Description The `point-to-angle` function returns an angle number calculated from *rectangle* and the point specified by *h* and *v* (for details, see *Inside Macintosh*). (See Figure D-12.)

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

■ **Figure D-12** Point to angle, calculated from two rectangles



`equal-rect` [*Function*]

Syntax `equal-rect` *rect1 rect2*

Description The `equal-rect` function returns `t` if *rect1* and *rect2* are equal and `nil` otherwise.

Arguments

rect1	A rectangle.
rect2	A rectangle.

`empty-rect-p` [*Function*]

Syntax `empty-rect-p` *left* `&optional` *top right bottom*

Description The `empty-rect-p` function returns `t` if the rectangle specified by *arg* is empty (contains no points) and `nil` otherwise.

A rectangle is empty if its bottom coordinate is less than or equal to the top or if the right coordinate is less than or equal to the left.

Arguments *left, top, right, bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on rectangles

Five generic functions govern graphics operations on rectangles.

`frame-rect` [*Generic function*]

Syntax `frame-rect` (*view* `view`) *left* `&optional` *top right bottom*

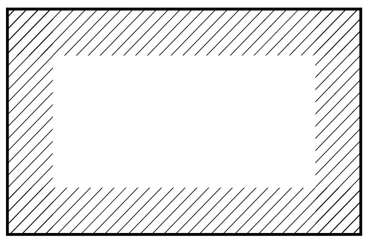
Description The `frame-rect` generic function draws a line just inside the boundaries of the rectangle specified by *arg*, using the current pen. (See Figure D-13.)

Arguments

view — A window or a view contained in a window.

left, *top*, *right*, *bottom* — These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

■ **Figure D-13** Rectangle framed in the current pen



`paint-rect` [*Generic function*]

Syntax `paint-rect` (*view* `view`) *left* `&optional` *top right bottom*

Description The `paint-rect` generic function fills the rectangle specified by *arg* with the current pen pattern and mode.

Arguments

view — A window or a view contained in a window.

left, *top*, *right*, *bottom* — These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`erase-rect` [*Generic function*]

Syntax `erase-rect` (*view* `view`) *left* `&optional` *top right bottom*

Description The `erase-rect` generic function fills the rectangle specified by *arg* with the current background pattern (in `patCopy` mode).

Arguments

view A window or a view contained in a window.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`invert-rect` [*Generic function*]

Syntax `invert-rect` (*view* `view`) *left* `&optional` *top right bottom*

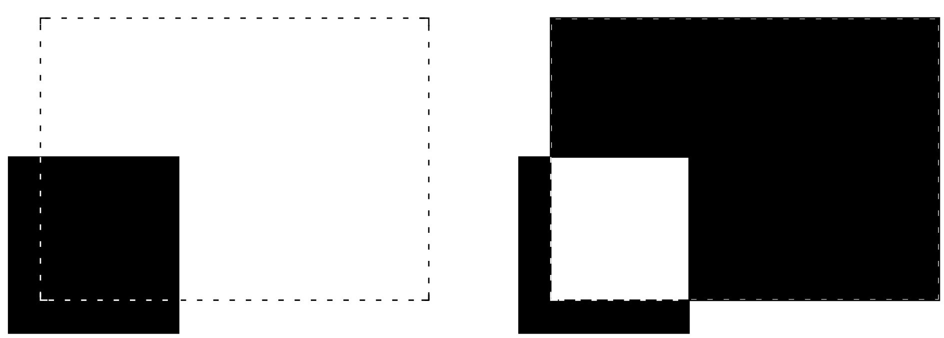
Description The `invert-rect` generic function inverts the pixels inside the rectangle specified by *arg*. (See Figure D-14.)

Arguments

view A window or a view contained in a window.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

■ **Figure D-14** Effects of `paint-rect` and `invert-rect`



`fill-rect` [*Generic function*]

Syntax `fill-rect` (*view* `view`) *pattern left* `&optional` *top right bottom*

Description The `fill-rect` generic function fills the rectangle specified by *arg* with *pattern* (in `:patCopy` mode).

Arguments

view A window or a view contained in a window.

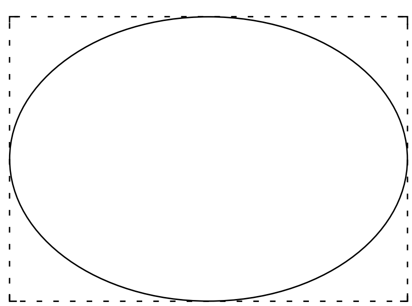
pattern A pattern record; see `pen-pattern` on page 696 for a discussion of pattern records.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on ovals

Ovals are drawn just inside rectangles. The oval is determined by the specified rectangle. (See Figure D-15.)

■ **Figure D-15** An oval within a rectangle



`frame-oval` [*Generic function*]

Syntax `frame-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `frame-oval` generic function draws a line just inside the boundaries of the oval specified by the rectangle, using the current pen pattern, mode, and size. The rectangle is specified
by *arg*.

Arguments

view A window or a view contained in a window.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`paint-oval` [*Generic function*]

Syntax `paint-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `paint-oval` generic function fills the oval specified by the rectangle specified by the arguments with the current pen pattern and mode.

Arguments

view A window or a view contained in a window.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`erase-oval` [*Generic function*]

Syntax `erase-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `erase-oval` generic function fills the oval specified by the rectangle with the current background pattern (in `:patCopy` mode). The rectangle is specified by the arguments.

Arguments

view — A window or a view contained in a window.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`invert-oval` [*Generic function*]

Syntax `invert-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `invert-oval` generic function inverts the pixels enclosed by the oval specified by the rectangle. The rectangle is specified by the arguments.

Arguments

view — A window or a view contained in a window.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`fill-oval` [*Generic function*]

Syntax `fill-oval` (*view* `view`) *pattern left* `&optional` *top right bottom*

Description The `fill-oval` generic function fills the oval specified by the rectangle with *pattern* (in `:patCopy` mode). The rectangle is specified by the arguments.

Arguments

view — A window or a view contained in a window.

pattern — A pattern record; see `pen-pattern` on page 696 for a discussion of pattern records.

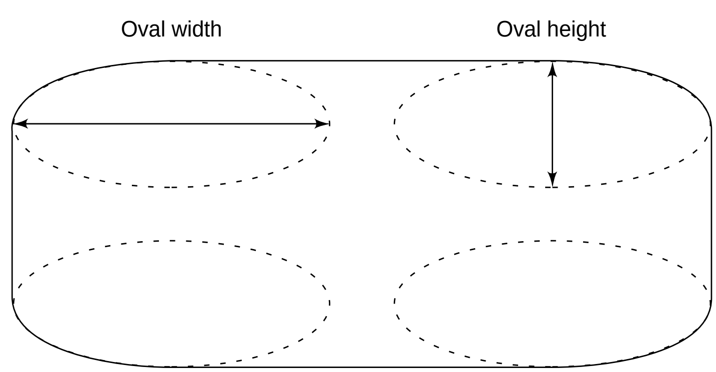
left, top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on rounded rectangles

A rounded rectangle (see Figure D-16) is a rectangle whose corners are rounded. The shapes of the corners are determined by ovals associated with the rounded rectangles. Thus, a rounded rectangle is determined by (1) the rectangle, (2) the width of the oval, and (3) the height of the oval.

■ **Figure D-16** Rounded rectangle



`frame-round-rect` [*Generic function*]

Syntax `frame-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description The `frame-round-rect` generic function draws a line just inside the boundaries of the rounded rectangle, using the current pen pattern, mode, and size. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height*.

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`paint-round-rect` [*Generic function*]

Syntax `paint-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description The `paint-round-rect` generic function fills the rounded rectangle with the current pen pattern and mode. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height*.

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`erase-round-rect` [*Generic function*]

Syntax `erase-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description The `erase-round-rect` generic function fills the rounded rectangle *t* with the current background pattern using `:patCopy` mode. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height*.

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left, top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`invert-round-rect` [*Generic function*]

Syntax `invert-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description The `invert-round-rect` generic function inverts the pixels enclosed by the rounded rectangle. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height*.

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left, top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`fill-round-rect` [*Generic function*]

Syntax `fill-round-rect` (*view* `view`) *pattern oval-width oval-height left* `&optional` *top right bottom*

Description The `fill-round-rect` generic function fills the specified rounded rectangle with the given *pattern* (in `:patCopy` mode). The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height*.

Arguments

view — A window or a view contained in a window.

pattern — A pattern record; see `pen-pattern` on page 696 for a discussion of pattern records.

oval-width — The width of the oval used to shape the rounded rectangle corner.

oval-height — The height of the oval used to shape the rounded rectangle corner.

left, *top*, *right*, *bottom* — These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on arcs

These functions perform graphics operations on arcs and wedge-shaped sections of ovals.

`frame-arc` [*Generic function*]

Syntax

`frame-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description

The `frame-arc` generic function draws a line just inside the arc specified by the rectangle, *start-angle*, and *arc-angle* using the current pen pattern, mode, and size. The rectangle is specified by the arguments. Figure D-17 shows an arc with a start angle of 45 and an arc angle of 135 inside a rectangle with the coordinates `#@(0 0)` and `#@(150 100)`.

Arguments

view — A window or a view contained in a window.

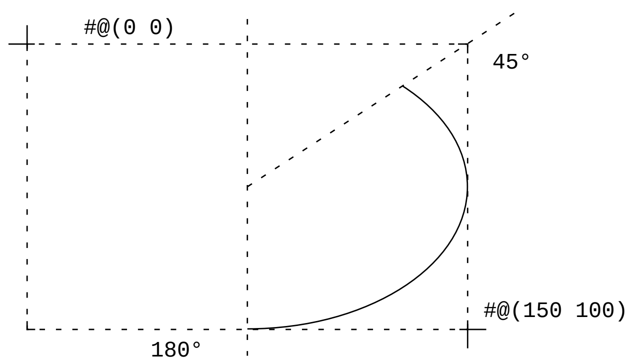
start-angle — The angle at which the arc originates, represented as an integer. An angle of 0 points straight up. (See the documentation of the `FrameArc` procedure in *Inside Macintosh*.)

arc-angle — The angle subtended by the arc.

left, *top*, *right*, *bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

■ **Figure D-17** Framing an arc




```
(frame-arc 45 135
           0 0 150 100)
```

`paint-arc` [*Generic function*]

Syntax `paint-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description The `paint-arc` generic function fills the arc specified by the rectangle, *start-angle*, and *arc-angle* with the current pen pattern and mode. The rectangle is specified by the arguments.

Arguments

view	A window or a view contained in a window.
start-angle	The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.
arc-angle	The angle subtended by the arc.

left, *top*, *right*, *bottom*
: These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`erase-arc` [*Generic function*]

Syntax `erase-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description The `erase-arc` generic function fills the specified arc with the current background pattern. The arc is specified by the rectangle, *start-angle*, and *arc-angle*. The rectangle is specified by *left, top, right, bottom.*

Arguments

view A window or a view contained in a window.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.

arc-angle The angle subtended by the arc.

left, *top*, *right*, *bottom*
: These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`invert-arc` [*Generic function*]

Syntax `invert-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description The `invert-arc` generic function inverts the pixels enclosed by the arc specified by the rectangle, *start-angle*, and *arc-angle*. The rectangle is specified by *left, top, right, bottom.*

Arguments

view A window or a view contained in a window.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.

arc-angle The angle subtended by the arc.

left, top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`fill-arc` [*Generic function*]

Syntax `fill-arc` (*view* `view`) *pattern start-angle arc-angle left* `&optional` *top right bottom*

Description The `fill-arc` generic function draws a line just inside the arc specified by the rectangle, *start-angle*, and *arc-angle* using the current pen pattern, mode, and size. The rectangle is specified by *left, top, right, bottom.*

Arguments

view A window or a view contained in a window.

pattern A pattern record; see `pen-pattern` on page 696 for a discussion of pattern records.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.

arc-angle The angle subtended by the arc.

left, top, right, bottom

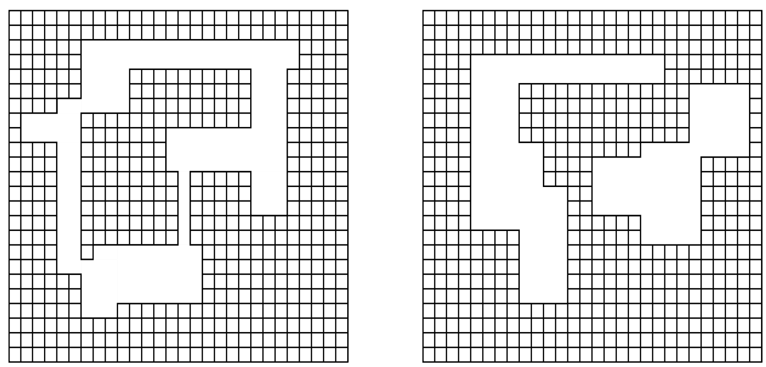
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Regions

A region divides the graphics plane of points into two sets of points: those inside the region and those outside the region. Regions can have any arbitrary shape. (See Figure D-18.)

The storage for regions is not subject to automatic garbage collection. You must reclaim region storage by calling the function `dispose-region`. With this limitation, the use of regions has been greatly simplified from the specification given in *Inside Macintosh*. Specifically, much of the initialization of regions is performed automatically.

■ **Figure D-18** Regions



`new-region` [*Function*]

Syntax `new-region`

Description The `new-region` function allocates a new empty region and returns it.

`dispose-region` [*Function*]

Syntax `dispose-region` *region*

Description The `dispose-region` function reclaims storage space used by *region* and returns `nil`.

Argument *region* A region.

`copy-region` [*Function*]

Syntax `copy-region` *region* `&optional` *dest-region*

Description The `copy-region` function either copies *region* into *dest-region*, if it is supplied, or creates a new region equivalent to *region*. It returns the new region or *dest-region*.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region A region.

dest-region Another region.

`set-empty-region` [*Function*]

Syntax `set-empty-region` *region*

Description The `set-empty-region` function destructively modifies *region* so that it is empty and returns the empty region.

Argument

region A region.

`set-rect-region` [*Function*]

Syntax `set-rect-region` *region left* `&optional` *top right bottom*

Description The `set-rect-region` function sets *region* so that it is equivalent to the rectangle specified by the arguments and returns the rectangular region.

Arguments

region A region.

left, *top*, *right*, *bottom*
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`open-region` [*Generic function*]

Syntax `open-region` (*view* `view`)

Description The `open-region` generic function hides the pen and begins recording a region. Subsequent drawing commands to the window add to the region. Recording ends when `close-region` is called. The function returns `nil`.

It is an error to call `open-region` a second time without first calling `close-region`.

Argument *view* A window or a view contained in a window.

`close-region` [*Generic function*]

Syntax `close-region` (*view* `view`) `&optional` *dest-region*

Description The `close-region` generic function shows the pen and returns a region that is the accumulation of drawing commands in the window since the last `open-region` for the window. It returns the result in *dest-region,* if supplied, or else in a newly created region.

It is an error to call `close-region` before `open–region` has been called.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

view	A window or a view contained in a window.
dest-region	A region.

Calculations with regions

The following functions do not draw; they simply perform calculations. They do not depend on a GrafPort, and so they are defined globally rather than as generic functions.

`offset-region` [*Function*]

Syntax `offset-region` *region h* `&optional` *v*

Description The `offset-region` function destructively offsets *region* by *h* to the right and *v* down and returns the offset region. If only *h* is given, it is interpreted as an encoded point, and its coordinates are used.

Arguments

region	A region.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`inset-region` [*Function*]

Syntax `inset-region` *region h* `&optional` *v*

Description The `inset-region` function destructively shrinks or expands *region* by *h* horizontally and *v* vertically and returns it. If only *h* is given, it is interpreted as an encoded point, and its coordinates are used.

Arguments

region	A region.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`intersect-region` [*Function*]

Syntax `intersect-region` *region1 region2* `&optional` *dest-region*

Description The `intersect-region` function returns a region that is the intersection of *region1* and *region2*. It returns the result in *dest-region*, if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

`union-region` [*Function*]

Syntax `union-region` *region1 region2* `&optional` *dest-region*

Description The `union-region` function returns a region that is the union of *region1* and *region2*. It returns the result in *dest-region*, if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

`difference-region` [*Function*]

Syntax `difference-region` *region1 region2* `&optional` *dest-region*

Description The `difference-region` function returns a region that is the difference of *region1* and *region2*. It returns the result in *dest-region*, if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

`xor-region` [*Function*]

Syntax `xor-region` *region1 region2* `&optional` *dest-region*

Description The `xor-region` function returns a region that consists of all the points that are in *region1* or *region2*, but not both. It returns the result in *dest-region*, if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

`point-in-region-p` [*Function*]

Syntax `point-in-region-p` *region h* `&optional` *v*

Description The `point-in-region-p` function returns `t` if the point specified by *h* and *v* is contained in *region;* otherwise, it returns `nil`. If only *h* is given, it is interpreted as an encoded point.

Arguments

region	A region.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`rect-in-region-p` [*Function*]

Syntax `rect-in-region-p` *region left* `&optional` *top right bottom*

Description The `rect-in-region-p` function returns `t` if the intersection of the rectangle specified by the arguments and *region* contains at least one point; otherwise it returns `nil`.

Arguments *region* A region.

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`equal-region-p` [*Function*]

Syntax `equal-region-p` *region1 region2*

Description The `equal-region-p` function returns `t` if *region 1* and *region 2* are identical in size, shape, and position; otherwise it returns `nil`.

Arguments *region1* A region.
region2 A region.

`empty-region-p` [*Function*]

Syntax `empty-region-p` *region*

Description The `empty-region-p` function returns `t` if *region* contains no points and `nil` otherwise.

Argument *region* A region.

Graphics operations on regions

These functions allow graphics operations on regions.

`frame-region` [*Generic function*]

Syntax `frame-region` (*view* `view`) *region*

Description The `frame-region` generic function draws a line just inside the boundaries of *region*, using the current pen.

Arguments

view	A window or a view contained in a window.
region	A region.

`paint-region` [*Generic function*]

Syntax `paint-region` (*view* `view`) *region*

Description The `paint-region` generic function fills *region* with the current pen pattern and mode.

Arguments

view	A window or a view contained in a window.
region	A region.

`erase-region` [*Generic function*]

Syntax `erase-region` (*view* `view`) *region*

Description The `erase-region` generic function fills *region* with the current background pattern, using `:patCopy` mode.

Arguments

view	A window or a view contained in a window.
region	A region.

`invert-region` [*Generic function*]

Syntax `invert-region` (*view* `view`) *region*

Description The `invert-region` generic function inverts the pixels enclosed by *region*.

Arguments

view	A window or a view contained in a window.
region	A region.

`fill-region` [*Generic function*]

Syntax `fill-region` (*view* `view`) *pattern region*

Description The `fill-region` generic function fills *region* with *pattern*, using `:patCopy` mode.

Arguments

view A window or a view contained in a window.

pattern A pattern record; see `pen-pattern` on page 696 for a discussion of pattern records.

region A region.

Bitmaps

Bitmaps are rectangular arrays of pixels that are either black or white. The following functions are useful in manipulating bitmaps.

`make-bitmap` [*Function*]

Syntax `make-bitmap` *left* `&optional` *top right bottom*

Description The `make-bitmap` function returns a new bitmap the size of the rectangle specified by the arguments. This bitmap is not displayed anywhere but can be used for calculations and storage.

Arguments

left, top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`copy-bits` [*Function*]

Syntax `copy-bits` *bitmap1 bitmap2 rect1 rect2* `&optional` *pen-mode region*

Description The `copy-bits` function copies and scales the bits inside *rect1* of *bitmap1* to the bits inside *rect2* of *bitmap2* using the transfer mode *pen-mode*.

If *region* is given, `copy-bits` clips the transferred bitmap to *region* and *pen–mode* assumes the default value `:srcCopy`.

Arguments

bitmap1	A bitmap.
bitmap2	A bitmap.
rect1	A rectangle.
rect2	A rectangle.
pen-mode	A pen mode. Should be one of the following keywords: `:patCopy`, `:patOr`, `:patXor`, `:patBic`, `:notPatCopy`, `:notPatOr`, `:notPatXor`, `:notPatBic`.
region	A region.

`scroll-rect` [*Generic function*]

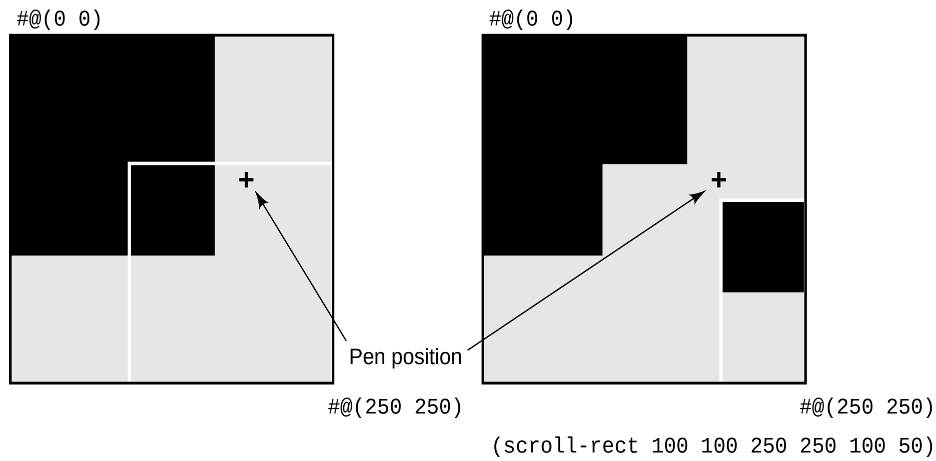
Syntax `scroll-rect` (*view* `view`) *rectangle h* `&optional` *v*

Description The `scroll-rect` generic function shifts the bits *h* pixels to the right and *v* pixels down within *rectangle*, erases the uncovered region, and adds the uncovered region to the window's update region. See Figure D-19 for an example of a scrolled rectangle.

Arguments

view	A window or a view contained in a window.
rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

■ **Figure D-19** A rectangle scrolled down and to the right



Pictures

A picture is a recording of a sequence of QuickDraw commands. Pictures may be played back at a later time, into any window. The MCL picture commands are slightly different from the QuickDraw ones, because Macintosh Common Lisp takes care of some of the memory management automatically. There are also some additional capabilities for manipulating pictures not found in QuickDraw.

`start-picture` [*Generic function*]

Syntax `start-picture` (*view* `view`) *left* `&optional` *top right bottom*

Description The `start-picture` generic function hides the pen and starts recording QuickDraw commands in a picture whose frame is the rectangle specified by the arguments, if supplied. Otherwise, the window's PortRect is the frame. The function returns `nil.`

It is an error to call `start-picture` a second time before calling `get-picture`.

You must dispose of pictures explicitly by calling `kill-picture` to reclaim their storage space

Arguments *view* A window or a view contained in a window.

left, top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`get-picture` [*Generic function*]

Syntax `get-picture` (*view* `view`)

Description The `get-picture` generic function shows the pen and returns a new picture representing the cumulative effect of all the QuickDraw commands given since the last call to `start–picture`.

It is an error to call `get-picture` before `start-picture` has been called in a window.

You must dispose of pictures explicitly by calling `kill-picture` to reclaim their storage space

Argument *view* A window or a view contained in a window.

`draw-picture` [*Generic function*]

Syntax `draw-picture` (*view* `view`) *picture* `&optional` *left top right bottom*

Description The `draw-picture` generic function draws *picture* in the window and returns *picture*.

Note that if the PortRect was used as a frame when the picture was made, and if the PortRect was arbitrarily large (the default set up by Macintosh Common Lisp), then scaling will produce no drawing (since the drawing frame is so much smaller than the creation frame).

Arguments

view A window or a view contained in a window.

picture A picture.

left, *top*, *right*, *bottom*
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`kill-picture` [*Function*]

Syntax `kill-picture` *picture*

Description The `kill-picture` function reclaims the storage space used by *picture* and returns `nil`.

Argument *picture* A picture.

Polygons

The MCL polygon commands are different from QuickDraw ones because Macintosh Common Lisp handles some of the memory management automatically.

`start-polygon` [*Generic function*]

Syntax `start-polygon` (*view* `view`)

Description The `start-polygon` generic function hides the pen and starts making a polygon. Subsequent `line` and `line-to` commands are added to a new polygon.

Within a single window, it is an error to call `start-polygon` twice before calling `get-polygon`.

You must dispose of polygons explicitly, using `kill-polygon` to reclaim their storage space.

Argument *view* A window or a view contained in a window.

`get-polygon` [*Generic function*]

Syntax `get-polygon` (*view* `view`)

Description The `get-polygon` generic function shows the pen and returns a polygon representing the cumulative effect of all the `line` and `line-to` commands since the last call to `start–polygon`.

Within a single window, it is an error to call `get-polygon` before a `start-polygon` has been called.

You must dispose of polygons explicitly, using `kill-polygon` to reclaim their storage space.

Argument *view* A window or a view contained in a window.

`kill-polygon` [*Function*]

Syntax `kill-polygon` *polygon*

Description The `kill-polygon` function reclaims storage space used by *polygon* and returns `nil`.

Argument *polygon* A polygon.

`offset-polygon` [*Function*]

Syntax `offset-polygon` *polygon h* `&optional` *v*

Description The `offset-polygon` function offsets *polygon* by *h* to the right and *v* down. This function can be performed outside of windows because it does not involve drawing. If only *h* is given,
it is interpreted as an encoded point.

Arguments

polygon	A polygon.
h	Horizontal position.
v	Vertical position. If *v* is nil (the default), *h* is assumed to specify both values.

`frame-polygon` [*Generic function*]

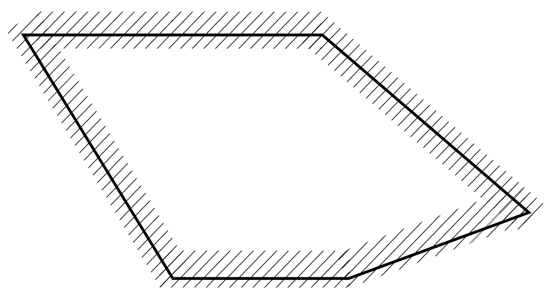
Syntax `frame-polygon` (*view* `view`) *polygon*

Description The `frame-polygon` generic function draws a line just inside the boundaries of *polygon* using the current pen. A framed polygon is shown in Figure D-20.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

■ **Figure D-20** A framed polygon



`paint-polygon` [*Generic function*]

Syntax `paint-polygon` (*view* `view`) *polygon*

Description The `paint-polygon` generic function fills *polygon* with the current pen pattern and mode.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

`erase-polygon` [*Generic function*]

Syntax `erase-polygon` (*view* `view`) *polygon*

Description The `erase-polygon` generic function fills *polygon* with the current background pattern, using `:patCopy` mode.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

`invert-polygon` [*Generic function*]

Syntax `invert-polygon` (*view* `view`) *polygon*

Description The `invert-polygon` generic function inverts the pixels enclosed by *polygon*.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

`fill-polygon` [*Generic function*]

Syntax `fill-polygon` (*view* `view`) *pattern polygon*

Description The `fill-polygon` generic function fills *polygon* with *pattern* using `:patCopy` mode.

Arguments

view	A window or a view contained in a window.
pattern	A pattern record; see `pen-pattern` on page 696 for a discussion of pattern records.
polygon	A polygon.

Miscellaneous procedures

This section contains functions to perform miscellaneous graphics procedures.

`local-to-global` [*Generic function*]

Syntax `local-to-global` (*view* `view`) *h* `&optional` *v*

Description The `local-to-global` generic function returns a global point that corresponds to the window's local point specified by *h* and *v*. If only *h* is given, it is taken to be an encoded point.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.

v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`global-to-local` [*Generic function*]

Syntax `global-to-local` (*view* `view`) *h* `&optional` *v*

Description The `global-to-local` generic function returns a point in the window's coordinate system that corresponds to the global point specified by *h* and *v*. If only *h* is given, it is interpreted as an encoded point.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`get-pixel` [*Generic function*]

Syntax `get-pixel` (*view* `view`) *h* `&optional` *v*

Description The `get-pixel` generic function returns `t` if the pixel specified by *h* and *v* is black and within the window's `VisRgn`; otherwise, it returns `nil`. If only *h* is given, it is interpreted as an encoded point.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`scale-point` [*Function*]

Syntax `scale-point` *rect1 rect2 h* `&optional` *v*

Description The `scale-point` function returns a point whose horizontal and vertical values are the scaled horizontal and vertical values of the point specified by *h* and *v*. If only *h* is given, it is interpreted as an encoded point. The scaling corresponds to the ratios of the width and height of *rect1* to the width and height of *rect2*.

Arguments

rect1	A rectangle.
rect2	A rectangle.
h	Horizontal position.

v Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`map-point` [*Function*]

Syntax `map-point` *source-rect dest-rect h* `&optional` *v*

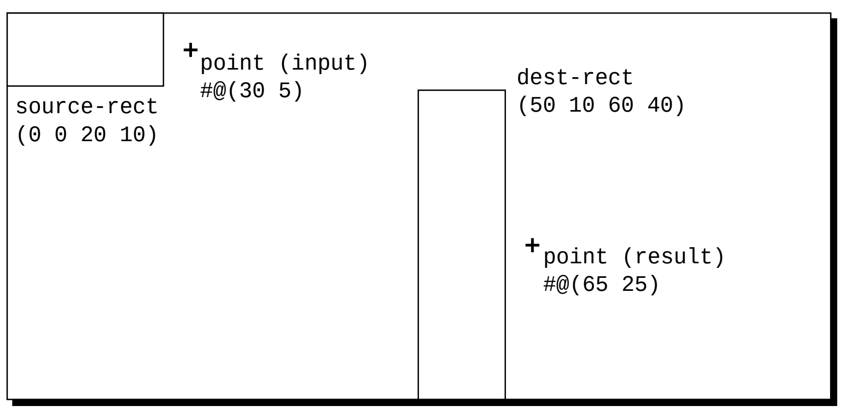
Description The `map-point` function returns a point that corresponds to *dest-rect* as the point specified by *h* and *v* corresponds to *source-rect*. If only *h* is given, it is interpreted as an encoded point.

The effect of `map-point` is shown in Figure D-21, where the point (30, 5) corresponds to *source-rect* as the point (65,25) does to *dest-rect.* The point `#@(30 5)` is half the width of *source-rect* to the right of *source-rect* and is located at the vertical midpoint of *source-rect*. The point `#@(65 25)` bears the same relation to *dest-rect*.

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

■ **Figure D-21** Effect of `map-point`



`map-rect` [*Function*]

Syntax `map-rect` *source-rect dest-rect mapped-rect*

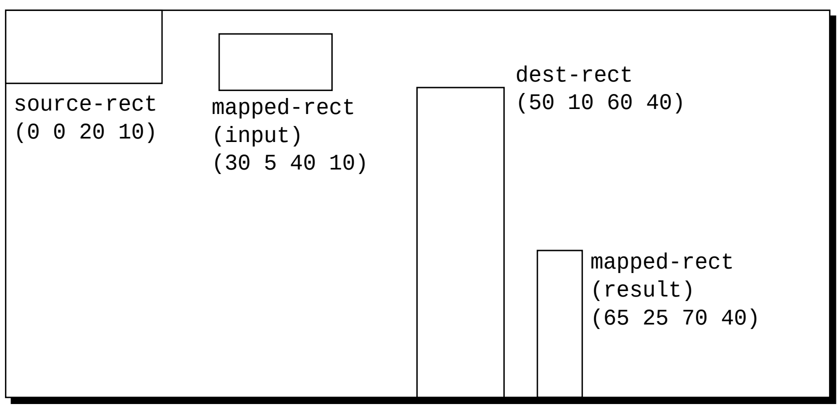
Description The `map-rect` function returns a rectangle that corresponds to *dest-rect* as *mapped-rect* corresponds to *source-rect*. The function destructively modifies *mapped-rect* to hold the returned value. The effect of `map-rect` is shown in Figure D-22, where the returned rectangle corresponds to *dest-rect* as *mapped-rect* (input) corresponds to *source-rect.*

This function is performed by applying `map-point` to the corner points of *mapped-rect*.

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
mapped-rect	A rectangle.

■ **Figure D-22** Effect of `map-rect`



`map-region` [*Function*]

Syntax `map-region` *source-rect dest-rect region*

Description The `map-region` function returns a region that corresponds to *dest-rect* as *region* corresponds to *source-rect*. The function destructively modifies *region* to hold the return value.

This function is effectively performed by applying `map-point` to all the points in the region.

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
region	A region.

`map-polygon` [*Function*]

Syntax map-polygon *source-rect dest-rect polygon*

Description The `map-polygon` function returns a polygon that corresponds to *dest-rect* as *polygon* corresponds to *source-rect*. The function destructively modifies *polygon* to hold the returned value.

This function is effectively performed by applying `map-point` to all the points that define the polygon.

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
polygon	A polygon.

Appendix E:

MCL 4.0 CD Contents

Contents

In this appendix you will find a summary of the contents of the MCL 4.0 CD and instructions for installing MCL 4.0 and MCL 3.1 from floppy disks.

What is on the MCL 4.0 CD-ROM

Highlights

This folder includes aliases to some of the more important, interesting and useful contents of the MCL 4.0 CD.

MCL 4.0

This folder contains a complete installed copy of Macintosh Common Lisp 4.0 for PowerPC Macs, ready for use on a hard disk. To install MCL 4.0 on your hard disk, simply drag this folder onto your hard disk. Note that you may not need to place the contents of the ThreadsLib folder in your Extensions Folder; see the Release Notes and the *Getting Started Guide* for more information.

MCL 3.1

This folder contains a complete installed copy of Macintosh Common Lisp 3.1 for 68K Macs, ready for use on a hard disk. To install MCL 3.1 on your hard disk, simply drag this folder onto your hard disk. If you wish to use the PTable extension for enabling EGC, place it in the extensions folder of your System Folder and restart your Macintosh. See the Release Notes and the *Getting Started Guide* for more information.

MCL 4.0 “Demo Version”

This folder contains a free demo version of MCL. This is a duration-limited version of MCL 4.0 that runs for 15 minutes per launch or until a specified expiration date via a password key supplied by Digitool. This MCL demo version and applications generated with it may be freely distributed.

MCL 4.0/3.1 Documentation

This folder contains softcopy of the combined documentation for MCL 4.0 and MCL 3.1 with complete indices, which can be read using Adobe Acrobat Exchange. If you do not have Adobe Acrobat Reader you will find an installer program for it in “Developer Essentials:Utilities:.”

MCL Floppy Disks

This folder contains two sets of folders with the contents of the MCL 4.0 and MCL 3.1 in segmented archives suitable for copying to floppy disks. Instructions for installing from these archives are given in the *Getting Started Guide*.

Additional MCL Source Code

This folder contains the sources to some of the built-in MCL objects, such as views, dialogs, menus, and the FRED editor. It also includes a file you can load to make edit-definition (meta-.) work for the definitions contained in the code files. (For MCL 4.0, this information is included in the application.)

Source code in this folder is provided as is, and is completely unsupported. If you attempt to program MCL using information gleaned from this source code, we will not be able to provide technical support, nor can we guarantee that your code will work in future versions of MCL.

Goodies from Digitool

This folder contains additional software from the MCL development team that is not part of the standard MCL product. The contents of this folder are from Digitool but are unsupported and may be untested.

This folder also includes earlier versions of MCL, such as MCL 3.0 and MCL 3.9 (the first PowerPC-native release of MCL), along with their respective documentation.

Goodies from MCL Friends

This folder contains a number of standalone applications written in MCL and contributed by their authors.

User Contributed Code

This large folder contains sample Lisp source code and other items that may be of interest to Common Lisp users. The contents of this folder have been contributed by users and are unsupported and may be untested.

Developer Essentials

This folder contains selections of the Developer Essentials folder included on several Apple CD-ROMs for programmers. The folder includes tools and information that may be of interest to Macintosh programmers using any development system. It includes utilities, online versions of Macintosh programming documentation, Macintosh interfaces for several programming languages, and versions of Macintosh system software.

Mail Archives & Other Docs

This folder includes the info-mcl and comp.lang.lisp archives. It also contains the Lisp FAQ and standards, a snapshot of the Digitool WWW site, and miscellaneous information.

Contents/Index

This folder contains aliases to all the first level of subfolders on the MCL 4.0 CD shown by name and open for a quick view of their contents.

On Location Indexes

This folder contains an On Location v2.0 index of the information on the MCL 4.0 CD. This index used with On Location v2.0 will allow searches of all filenames on the CD-ROM as well as the contents of all text files and files of several other types.

- Note: Some text files on the CD-ROM are larger than the 32K size limit of SimpleText utility (in "Developer Essentials:Utilitites:"). Use the MCL editor FRED or another word processor to open such files.

What is in the MCL 4.0 folder

MCL 4.0

This is the MCL 4.0 application.

MCL Help and MCL Help Map.pfsl

You can modify the "MCL Help" Library file to customize the documentation strings it contains. You must then create a new "map" using "make-help-map.lisp". This file contains instructions.

Examples Folder

The Examples folder contains various MCL utilities. Each of the files contains comments that serve as brief documentation.

- **animated-cursor.lisp**
  Provides ability to cycle through cursor resources to get effects such as the spinning beach ball.
- **appleevent-toolkit.lisp**
  Provides useful functions for sending and processing AppleEvents.

- **array-dialog-item.lisp**
  A Table-Dialog-Item subclass for displaying arrays.
- **assorted-fred-commands.lisp**
  FRED, MCL 4.0's editor, is fully programmable. This file contains examples of additional FRED commands.
- **auto-fill.lisp**
  A simple autofill mode for FRED.
- **balloon-help-menu.lisp**
  This file lets you add your own menu items to the Balloon Help menu, which appears on the System 7 menubar.
- **Binhex** folder
  The two files in this folder contain an example of a "stand-alone" application: follow the instruction in Binhex.lisp to create an application in which the user need not know that he or she is using a Lisp-based system. The application encodes and decodes "BinHex" files.
- **cfm-mover.lisp**
  This file defines utility similar to the Font/DA Mover for moving CFM libraries between files.
- **check-and-change.lisp**
  This file contains code used in an example in Chapter 5: Dialog Items and Dialogs. The code illustrates the use of enter-key-handler and exit-key-handler.
- **config.lisp**
  This file generates a report of your current Macintosh hardware configuration and Macintosh Common Lisp environment.
- **define-interrupt-handler.lisp**
  Provides the ability to write interrupt handlers in Lisp rather than C or Pascal.
- **defobfun-to-defmethod.lisp**
  This file automates conversion of simple Object Lisp programs (Object Lisp was MACL 1.x's object system) to CLOS (the Common Lisp Object System used by MCL since version 2.0). In a FRED window, it changes instances of (defobfun (function type) args body) to (defmethod function ((type type) args) body). Queries before each change.
- **driver.lisp**
  An example of how to access the Device Manager's drivers from Macintosh Common Lisp. Used by "serial-streams.lisp".
- **escape-key.lisp**
  Makes the Escape key cause the next character to behave as if the Meta key were pressed.
- **eval-server.lisp**
  Handles eval, dosc, and scpt AppleEvents.

- **fasl-concatenate.lisp**
  Defines the function fasl-concatenate, which can be used to concatenate multiple `fasl` or multiple `pfsl` files into a single file. This will speed up loading and ease distribution.
- **fast-slot-value.lisp**
  Optimization for slot value when the class is known at compile time.
- **FF Examples** folder
  Three of the files in this folder ("ff-example.c", "ff-example.lisp", and "ff-example.test") contain code to demonstrate the use of MCL 4.0's Foreign Function interface. The fourth file ("ff-example.c.o") is the object file, compiled by the MPW C compiler, for "ff-example.c".

  The "ff-example.lisp" file (it's in "examples:ff examples:") contains the following:

  ```
  (ff-load "ff;ff-example.c.o"
           :ffenv-name 'test
           :libraries '("clib;StdCLib.o"
                        "mpwlib;interface.o"))
  ```

  The libraries referred to are as they exist in MPW 3.2 and 3.2.x. If you are using an earlier version of MPW, you will need to reference "clib;Cinterface.o" before "mpwlib;interface.o".
- **fred-word-completion.lisp**
  Provides word completion for symbols in a FRED window.
- **grapher.lisp**
  Implements the base functionality for node and grapher windows.
- **load-all-patches.lisp**
  A simple alternative patch loading mechanism.
- **mac-file-io.lisp**
  This file implements something similar to the high-level file I/O primites in *Inside Macintosh*.
- **mark-menu.lisp**
  Adds a menu of editor-window marks, much like MPW's. Marks are not saved with the file.
- **mouse-copy.lisp**
  When this file is loaded, command-click copies the expression nearest the cursor location to the location of the insertion point.
- **NotInROM** folder
  This folder contains Michael Engber's package implementing most of the "traps" that are "Not in ROM".
- **old-dialog-hooks.lisp**
  Hooks to make the new dialog package more compatible with that of MACL 1.x. This code will run, but is intended principally as documentation of what has changed since that version.
- **old-file-search.lisp**
  Macintosh Common Lisp 2.0 version of Search Files dialog.

- **pict-scrap.lisp**
  This file defines a scrap-handler for scraps of type PICT. Once it is loaded, windows that copy and paste PICTs are able to share their work with other applications.
- **picture-files.lisp**
  Examples of reading and writing picture files, adapted from the code on page V-88 of Inside Macintosh. Shows how to draw a PICT file in a window.
- **print-class-tree.lisp**
  Code to print a simple class-tree.
- **processes.lisp**
  Code to launch or bring forward a numbr of applications from within MCL
- **progress-indicator.lisp**
  Creates a window for monitoring the progress of any task.
- **query-replace.lisp**
  Implements a query-replace function in FRED, using Control-Meta-R.
- **scrolling-windows.lisp**
  Implements a new class of windows that contain scroll bars and a scrollable area.
- **serial-streams.lisp**
  Implements a class of serial streams, which inherit from drivers and provide a stream interface to the serial drivers on the Macintosh.
- **shapes-code.lisp**
  The Lisp version of a classic Mac programming exercise. Creates a window containing a drawing area and two buttons: “circles” and “squares.” Clicking in the drawing area draws a circle or square; clicking either button redraws all the shapes.
- **text-edit-dialog-item.lisp**
  Implements text-edit-dialog-items. If FRED is too big for your application, you may wish to replace editable-text-dialog-items with text-edit-dialog-items.
- **thermometer.lisp**
  A simple thermometer that displays one or more values in a rectangular area. GC-THERMOMETER and FILE-THERMOMETER are two examples provided in the file.
- **timers.lisp**
  This code implements Genera style timers.

- **toolserver.lisp**
  AppleEvents interface to ToolServer (to use, launch ToolServer first). ToolServer is a stand-alone, tool-execution environment for Macintosh Programmer Workshop (MPW) tools such as the C and Pascal compilers and linker. ToolServer makes use of the AppleEvent feature of System 7. A copy of ToolServer is found in the "Developer Essentials" folder on the MCL 4.0 CD. The latest version is always available on E.T.O. (Essentials - Tools - Objects), a CD-ROM subscription series distributed by Apple through APDA.
- **turtles.Lisp**
  A simple object-oriented turtle graphics package.
- **uk-keyboard.lisp**
  A FRED extension to make the sharp sign character ("`#`") easier to type on keyboards like those used in the United Kingdom.
- **View-Example.lisp**
  A simple example of views code.
- **windoid-key-events.lisp**
  How to make a windoid handle key events and null events.

Interface Tools folder

Contains a system that helps you design dialogs interactively. Instructions for its use are in the file About Interface Tools.

Library folder

Most of the files in the Library folder contain code that provides functionality that is not included in the MCL 4.0 image. If you need to use the interface to QuickDraw, for example, this form will load it:

```
(require :quickdraw)
```

Some of the files in the Library folder contain code that is autoloaded by MCL 4.0 when you try to use its functionality (the first time you type a form that uses LOOP, for example, "loop.lisp" is automatically loaded). These are files that will be autoloaded (if there is a compiled version of the file, it will be used instead): "help-manager.lisp", "lisp-package.lisp", "loop.lisp", and "resources.lisp".

A few of the Library files contain code that is already a part of the MCL 4.0 image (that is, you never need to load them). These files, provided for people who want to write extensions or who are just curious, are: the files in the Inspector folder, and the files “scroll-bar-dialog-items.lisp,”“pop-up-menu.lisp,” and “save-application-dialog.lisp.”

ThreadsLib

This folder contains the “ThreadsLib” extension. You do not need this extension, and must not use this file, if your system already has the Thread Manager. Check its accompanying notes for details.

pmcl-kernel, pmcl-library, and pmcl-compiler

These are shared libraries used by MCL 4.0. They should remain in the same folder as MCL 4.0 or be aliased from your Extensions Folder.

Index

Symbols

A

B

C

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y

Z

The DIGITOOL Publishing System

This Digitool manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and Adobe FrameMaker software. Proof and final pages were created on the Apple LaserWriter printers. Line art was created using Adobe Illustrator. PostScript®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Palatino. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writer for 2.0 release: Sarah Smith

Writers for 3.0 release: Ellen Golden and Becky Spitz

Writer for 4.0 release: Andrew Shalit

Illustrator: Sandee Karr

Production: Julie Gilbert

Special thanks to Gary Byers, Steve Hain, Alice Hartley, Steven Mitchell, William St. Clair, Steve Strassmann, and to our skilled and helpful alpha and beta testers.