Macintosh®

Allegro Common LISP

Table of Contents

Introduction
- Documentation
- Running Allegro CL
- Installing Allegro CL
- Launching Allegro CL

1. Getting Started
   - Overview 1-1
   - The Allegro CL Environment 1-1
   - The Menubar 1-3

2. Fred the Editor
   - Overview 2-1
   - Fred and Packages 2-1
   - Editing Macintosh Style 2-1
   - Editing Emacs Style 2-3

3. Object Lisp
   - Overview 3-1
   - Tutorial 3-1
   - Object Lisp Functions 3-13

4. Macintosh Basics
   - Overview 4-1
   - Points 4-1
   - Font Specs 4-2
   - Turnkey Dialogs 4-3
   - Miscellaneous 4-4

5. Menus
   - Overview 5-1
   - Menubars 5-2
   - Menus 5-3
   - Menu-items 5-5

6. Windows
   - Overview 6-1
   - Window Functions and Variables 6-1
   - Supporting Undo 6-5
   - Supporting Save and Save As... 6-6

7. Dialogs
- Overview 7-1
- Dialog Functions 7-3
- Dialog Items 7-5
- Specialized Dialog-items 7-8
- Table-dialog-items 7-10
- Specialized Table-dialog-items 7-14

8. Events
- Overview 8-1
- Event Handlers 8-1
- Event Information Functions 8-3
- The Event Management System 8-4
- Cursor Handling 8-6

9. Programming Fred
- Overview 9-1
- Windows, Buffers, and Marks 9-1
- Parameter Conventions 9-2
- The Kill-Ring 9-2
- Buffer Functions 9-2
- Fred Window Functions 9-6
- Fred Command Tables 9-10

10. File System Interface
- Overview 10-1
- Pathname Specification 10-1
- Common Lisp Pathnames 10-1
- Parsing Pathname Strings 10-1
- Pathname Escape Character 10-3
- Using Lisp Pathnames 10-4
- Macintosh Pathnames 10-5
- Default Directories 10-6
- Search Path 10-7
- Wildcards 10-7
- Logical Pathnames 10-8
- File System Manipulation 10-9
- User Interface Functions 10-13

11. Debugging
Overview 11-1
Fred Commands 11-1
Inspect 11-2
Step 11-2
Backtrace 11-2
Trace 11-3

12. Low-level System Interface
Overview 12-1
Sharing Data Between Allegro CL and the Macintosh Operating System 12-1
Lisp Data Representation 12-1
Calling Macintosh Traps from Allegro CL 12-2
Memory Management 12-6
Pascal *var* arguments 12-10
DefPascal 12-10

13. Pascal Records
Overview 13-1
The Structure of Records 13-1
Record Functions 13-2

Appendix A: Implementation Notes

Appendix B: System Parameters

Appendix C: Quickdraw Graphics

Appendix D: Selected Bibliography

Glossary

Index

Introduction

Documentation
Running Allegro CL
    Hardware Requirements
    System Software Configuration
Installing Allegro CL
Launching Allegro CL

Introduction

Allegro CL is a complete implementation of the Common Lisp standard, with additional programming and Macintosh interface tools. It is based on the specification described in *Common Lisp the Language* by Guy Steele (Digital Press 1984), taking into account clarifications which have occurred since the book's publication.

For details on our implementation, and our approach to situations not clearly defined by Common Lisp, see the appendix Implementation Notes.

This manual assumes some knowledge of Common Lisp the Macintosh user interface. If you don't know anything about the Macintosh, read through the *Macintosh User's Guide* and try out some simple Macintosh applications, such as MacWrite™ and MacPaint™. For information on Lisp, consider some of the books in the bibliography at the end of the manual.

Documentation

Common Lisp: the Language should be used as the primary reference for the Common Lisp features of Allegro CL. The information in *Common Lisp: the Language* is not duplicated in the Allegro CL manual. Tutorials and other reference works on Common Lisp are listed in the bibliography.

Information regarding installation and use of Macintosh computers can be found in the *Macintosh User's Guide*. Technical information on the Macintosh can be found in *Inside Macintosh* volumes 1-5 (Addison Wesley, 1985-87). Because Allegro CL adds a high-level programming interface to many Macintosh system routines, you may not need to use *Inside Macintosh* at all. However, it does provide good background material, and is probably indispensable for programming Macintosh features which Allegro CL does not provide at a high level.

The information specific to Allegro CL is described in this manual. The documentation covers the Allegro CL environment (editing, evaluating, etc), language extensions (file system extensions and Object Lisp), Macintosh tools, and implementation notes (information on the compiler, and Allegro CL's approach to ambiguities in Common Lisp).

Running Allegro CL

Hardware Requirements

To run Allegro CL you will need a Macintosh Plus, Macintosh SE, or Macintosh II. Allegro CL requires 1 megabyte of RAM and 1.6 megabytes of disk storage (though 2 megabytes of RAM and a hard disk are recommended). It will currently take advantage of up to 8 megabytes of RAM, the limit imposed by the Macintosh operating system.

Allegro CL has been tested and works with the Levco Prodigy-4, General Computer's HyperCharger, and E-Machines' Big-Picture monitor. It has not been tested with other hardware add-ons, but should run on these without problems. Allegro CL runs approximately 5-times faster on a Macintosh II than it does on a 68000-based Macintosh.

System Software Configuration

Allegro CL has been designed to run the Macintosh System version 4.1 and Finder 5.5. These are the versions provided on the Allegro CL disks. It should also run on newer versions of the system software as they are released by Apple.

Installing Allegro CL

Allegro CL comes on 2 double sided disks.

- Disk A contains three directories. The first is a system folder containing Macintosh System 4.1 and Finder 5.5. The second is the ccl-docs directory, which contains files used by the Allegro CL help system and other useful text documents. The third is the Library folder, containing example and utility files.

- Disk B contains Allegro CL and a file named init.lisp. The init.lisp file is automatically loaded when you launch Allegro CL. The contents of this file may be changed by the user.

You should not run Allegro CL from the original disk. Instead, copy it to a hard disk (the preferable solution) or make a working copy on another floppy. If possible, the Allegro CL application, the init.lisp file, and the ccl-doc and Library folders should all reside in the same folder. The original disks should be kept as back-ups. Allegro CL is not copy-protected, so it can be copied easily.

Launching Allegro CL

To launch Allegro CL, double-click on the Allegro CL icon. This will load the kernel, initialize the Lisp system, and load the file init.lisp or init.fasl (if one of these is present). When this is done, you will be presented with the Allegro CL menubar, a Listener window, and a welcome message.

You can also launch Allegro CL by double-clicking an Allegro CL document. When this is done Allegro CL is started and the document is loaded. In this case, no init file is loaded.

You may wish to turn off the RAM cache and remove memory-hungry inits before starting Allegro CL. When running under MultiFinder, Allegro CL should be given a partition of at least 1 megabyte.

Getting Started 1

Overview
The Allegro CL Environment
- The Lisp Listener
- Evaluating Lisp Expressions
- The Environment
- Note on the Clover Key
- Note on the Clipboard and Kill-Ring

The Menubar
- File Menu
- Edit Menu
- Eval Menu
- Tools Menu
- Windows Menu

Getting Started

Overview

This chapter describes the basic information you'll need to get started programming in Allegro CL. It describes the Allegro CL environment and the functions found in the menubar.

The Allegro CL Environment

Allegro CL provides an integrated programming environment built around a Lisp listener, Emacs-style editor, and window- and menu-based programming tools.

If you are familiar with the Macintosh user interface, you can probably start working in Allegro CL with little instruction. However, there are a few things to keep in mind:

The Lisp Listener

The Lisp listener is a special window designed for interactive Lisp programming.

- Text entered into the listener by the user is in boldface. Text printed by Allegro CL is in a plain typeface.
- Interaction occurs at the bottom of the listener. The upper portions of the listener provide a transcript history.
- Typing RETURN drops the caret (i.e. the insertion point) to the bottom of the listener. If there is a selection, the selection is copied down to the bottom of the listener. If there is no selection, the text surrounding the caret is copied down only if it is boldface (i.e. if it was typed by the user), not if it is plain (i.e. printed by Allegro CL).
- Typing ENTER performs both copy-down and evaluation.
- When at the bottom of the listener, typing RETURN from a complete Lisp expression evaluates the expression. If the expression is not complete (e.g. there are unmatched parentheses or quotes), a carriage return is inserted and no evaluation takes place.
- CONTROL-RETURN inserts line-breaks in the listener without causing evaluation or drop-down. This is useful when expressions do not fit comfortably on a single line. (However, such expressions might be more conveniently edited in a non-listener editor window).
- All Fred editor commands work from the listener.
- The listener may be closed, like any other window. If there is no listener, any attempt to output to the listener will create a new listener. Closing the listener purges its buffer, which can be useful if a large buffer begins to retard listener performance.

Evaluating Lisp Expressions

- Expressions can be evaluated from any editing window. They do not have to be copied to the listener for evaluation. Simply select the expression or place the caret at the end of an expression, and type ENTER or choose the **Eval** menu-item from the **Eval** menu.
- Allegro CL uses an incremental compiler. In the default mode, evaluation of function definitions actually causes compilation (provided there are no lexically apparent bindings); expressions other than definitions are evaluated. To turn off auto-compilation of definitions set `*compile-definitions*` to `nil`.

The Environment

- When the user double-clicks the Allegro CL icon, Allegro CL attempts to load a file called "init.lisp" or "init.fasl" from its home folder (i.e. the folder containing Allegro CL). If both files are found, the more recent one is loaded. The init file can be used to create a default initial environment. For example, it can set the state of various system parameters, load utility files, etc.
- The user may also double click an Allegro CL file. In this case, Allegro CL is launched, and the file is loaded.

• There are several parameters used to control the environment. These can be set in the normal Lisp fashion, in an init file, or from the **Print Options...** and **Environment...** menu-items, available through the **Tools** menu. These parameters are described in the System Parameters appendix.
• Functions are loaded into main memory as they are needed. There may be slight pauses as functions are first loaded in from disk. Garbage collection purges any Allegro CL functions which are not currently on the stack. This "load on call" mechanism allows Allegro CL to operate with extremely modest memory requirements.
• On systems with sufficient memory, function purging can be disabled by calling `(purge-functions nil)`. Purging can be re-enabled by calling `(purge-functions t)`. All functions can be pre-loaded into memory by calling `(preload-all-functions)`. See the appendix Implementation Notes for details.
• During garbage collection, Lisp operation pauses and the cursor turns into the letters "GC". Garbage collection can be invoked manually by calling the procedure `gc`.
• Allegro CL has a pseudo multi-tasking system. This allows editing and other operations to occur while Lisp code is running (e.g. during evaluation and compilation). Allegro CL never puts up the Macintosh watch cursor. When Allegro CL is in the eval phase of the read/print/eval loop (i.e. when it is executing rather than reading) the **About Allegro CL...** menu-item will be preceded by a diamond.
• Certain Allegro CL tasks are non-interruptible, including garbage collection and event processing (such as menu-item-action processing). During these operations no other functions can be performed.
• Lisp operations—besides those which are non-interruptible—can be stopped by typing clover-. (clover period). This will return control to the top level listener loop, or will enter a break loop if `*break-on-errors*` is non-`nil`. The user can define a synonym for clover-. by setting `*abort-character*` to the desired character.
• Allegro CL has an elegant method for specifying directories with logical pathnames. These are described in a section of the File System chapter.

Note on the Clover Key

The Allegro CL environment attempts to conform to both Macintosh and Emacs standards. The largest problem is in the use of keyboard modifier keys. (Shift and control are examples of modifier keys. They do not register as keystrokes, but are only used to modify the values of other keystrokes.) Older Macintosh computers do not provide a control key. The following method is used to work around this limitation. (Note: in the discussion below, "clover-key" refers to the physical key on the keyboard, "command" refers to the logical modifier key used to invoke menu-items, and "control" refers to the logical modifier key used to invoke Fred commands.)

Allegro CL can operate in two modes: Macintosh mode and Emacs mode. In Emacs mode, the clover key is used as control, and shift-clover is used as command. In Macintosh mode, the clover key is used as command, and shift-clover is used as control. If there is no menu-item for a given character, clover and shift-clover will both be interpreted as control.

The global variable `*emacs-mode*` determines whether Allegro CL is in Macintosh or Emacs mode. Allegro CL originally comes up in Macintosh mode.

The physical control key is supported on those Macintoshes which provide one.

Note on the Clipboard and Kill-Ring

Allegro CL integrates the Macintosh clipboard with the Emacs kill-ring. The clipboard is taken to be the top item in the kill-ring. Commands which move text to the clipboard also move it to the top

of the kill-ring, and vice-versa. This mechanism functions properly with all Macintosh requirements, such as the ability to copy and paste between applications. For Macintosh users, the kill-ring can be thought of as a multi-level clipboard.

The Menubar

This section describes the options available through the Allegro CL menubar. Note that the menubar can be customized by the user (for details see the Menus chapter). This section describes the menubar that appears when an unmodified version of Allegro CL is booted.

Command-key equivalents for menu-items are noted after the menu-item name.

Apple Menu

About Allegro CL... [*Menu Item*]
displays a dialog box showing the version number of Allegro CL. Also gives the version numbers of the Macintosh ROM and system file.

The remainder of the **Apple** menu contains desk accessories. These are described in the *Macintosh User's Manual.*

File Menu

New (command-n) [*Menu Item*]
creates an editor window for a new file.

Open... (command-o) [*Menu Item*]
allows the user to select a text file, and creates a new editor window for the file.

Open Selected File [*Menu Item*]
is only enabled when there is a selection in the top editor buffer. It attempts to parse this selection as a pathname; if successful, it creates an editor window for the pathname's file. The title of this menu-item will change to reflect the contents of the current selection.

Close [*Menu Item*]
closes the active window. If text in the window has been edited since the last time it was saved, a dialog box will appear asking if you want to save or throw away the changes.

Save (command-s) [*Menu Item*]
saves the contents of the active window into the file named in the window's title bar. If the window is not yet associated with a disk file, Save As... is invoked.

Save As... [*Menu Item*]
allows the user to specify a directory and file name, and saves the contents of the active window to that directory/file name.

Revert [*Menu Item*]
reverts the window to the last version saved to disk. Before the reversion occurs, you will be asked to verify that you *really* want to revert to the last version saved.

Page Setup... [*Menu Item*]
allows the user to set printing options for the current hardcopy device.

Print (command-p) [*Menu Item*]
prints the contents of the active window to the currently selected hardcopy device.

Quit [*Menu Item*]
closes all open windows and exits the Lisp environment. If there are any modified and unsaved editing windows, you are given a chance to save them.

Edit Menu

Undo (command-z) [*Menu Item*]
undoes the last editor command. This does not work for all commands (it will be disabled at times when it will not work). In general, Undo works for insertions and deletions, but not for caret movement. The name of this menu-item may change depending on context.

Cut (command-x) [*Menu Item*]
deletes the selected region and pushes it onto the kill-ring/clipboard.

Copy (command-c) [*Menu Item*]
copies the selected region to the top of the kill-ring/clipboard.

Paste (command-v) [*Menu Item*]
replaces the current selection with the text on the top of the kill-ring (i.e. with the contents of the clipboard). If there is no current selection, the text is simply inserted.

Clear [*Menu Item*]
deletes the current selection. The deleted text is not saved anywhere, and the kill-ring/clipboard is not altered.

Select All (command-a) [*Menu Item*]
selects the entire contents of the active window.

Insert Killed String... [*Menu Item*]
shows a dialog box of the strings in the kill-ring. The user may select one for insertion in the top editor buffer.

Search (command-f) [*Menu Item*]
brings up a search/replace dialog box.

Change Font [*Menu Item*]
allows the user to change the font, font-size, and font-style of the top editor buffer. This menu-item will be disabled if the top window is not a Fred window.

Eval Menu

Eval Selection (command-e) [*Menu Item*]
evaluates the current selection in the top editing buffer. If there is no selection and the caret is next

to a close parenthesis, the expression bounded by the parenthesis is evaluated. If `*compile-definitions*` is non-`nil` (and there are no lexically apparent bindings), function definitions will be compiled rather than evaluated. If `*fast-eval*` is non-`nil` the expression may be compiled and executed.

Eval Buffer [*Menu Item*]
evaluates the entire contents of the top editing buffer. If `*compile-definitions*` is non-`nil` (and there are no lexically apparent bindings), definitions will be compiled rather than evaluated. This is equivalent to **Select All** followed by **Eval Selection**.

Load... [*Menu Item*]
allows the user to select a file for loading into Allegro CL. Both text and fasl files may be loaded.

Compile File... [*Menu Item*]
allows the user to select a file for compilation. The user will be asked to specify both the source and destination files.

Tools Menu

List Definitions [*Menu Item*]
brings up a modeless dialog box containing a table of all the definitions in the top editor buffer. The user can select a definition, and the buffer will scroll to that definition. When the dialog is the top window, typing the name of a definition causes the table to scroll to the entry for that definition.

Edit Definition... [*Menu Item*]
allows the user to enter a symbol name, and attempts to find the source code for the definition of the symbol.

Apropos... [*Menu Item*]
performs `apropos` on a string entered by the user. The user can specify several options for filtering the symbols returned.

Inspect [*Menu Item*]
brings up the inspector control dialog box. (See the Debugging chapter for details).

Backtrace [*Menu Item*]
this item is only enabled when Allegro CL is in a break loop. Selecting it brings up a stack backtrace. (See the Debugging chapter for details.)

Documents... [*Menu Item*]
opens up a standard file dialog to the ccl-docs folder. This folder contains text files with information on Common Lisp, and text files used by the Allegro CL help system.

Fred Commands [*Menu Item*]
brings up a window of Fred keyboard commands. The commands shown are calculated at run time from `*comtab*` (see the chapter Programming Fred for a description of comtabs).

Print Options... [*Menu Item*]
brings up a dialog box which allows the user to set the values of several Lisp printer parameters.

Environment... [*Menu Item*]
brings up a dialog box which allows the user to set the values of several global variables which effect the Allegro CL environment.

Windows Menu

The titles of all visible windows appear as items in this menu. The menu is ordered by window layer (i.e. the front window is the top menu-item and the back window is the bottom menu-item). To activate a window, select its menu-item. Editor buffers which have been changed and not saved have a cross next to their names. The menu-item of the front window is dimmed and cannot be selected. The listener menu-item has the command-key equivalent command-l.

Fred the Editor 2

Overview
Editing Macintosh Style
- Menus
- Windows
- The Caret

Editing Emacs Style
- Control, Command, Option, and Meta
- Documentation Conventions
- Help Functions
- Movement
- Insertion
- Deletion
- Lisp Operations
- Windows
- Miscellaneous

Fred the Editor

Overview

In the Allegro CL programming environment, Fred (Fred Resembles Emacs Deliberately) plays the star role of editor. Fred is a combination of the standard Macintosh multiple window text-editor and Emacs. If you are familiar with other editors on the Macintosh (such as MacWrite or QUED), you can skip the section on Macintosh-style editing. Following the Macintosh section is a section describing Fred's Emacs-style capabilities. Documentation for programming Fred is provided in the chapter Programming Fred.

Fred and Packages

Every Fred window has an associated package. Expressions evaluated from the window are evaluated in the window's package. If the window's package is `nil`, then `*package*` is used. (Note that the declaration `in-package` effects `*package*`. If a window has an associated package, `in-package` declarations will not effect evaluation of expressions from the window.)

A window's package may be set in one of two ways:

- Through a mode line at the start of the buffer. This method only works if the mode line is present when the buffer is opened. Only the package declaration in the mode line is parsed. The other declarations are ignored.

  The mode line, if present, must be the first non-empty line in the buffer. It begins with a comment character (semi-colon), followed by `-*-`, followed by the package declaration. For example, the following mode line would cause evaluations in a buffer to take place in the `QUUX` package.

```
;;; -*- package: QUUX  -*-
```

- Through the function `set-window-package`. This is an object function defined for Fred windows. It takes a single argument, which should be a package or a symbol.

You can find the package associated with a window through the window object function `window-package`.

Editing Macintosh Style

At first glance, Fred looks just like a normal Macintosh editor.

Menus

Many Fred commands are invoked through menus. These commands are described in the chapter Getting Started.

Windows

All Fred editing takes place inside windows. Window movement and scrolling commands conform to Macintosh standards.

- Select a window
  by moving the mouse into it and clicking. This brings it to the front and makes it the active window.

- Scroll a window horizontally and vertically
  by using the scroll bars at the bottom and right edge of the window respectively, or by dragging the thumb.

- Move the window
  by moving the cursor to the window's title bar, holding down the mouse button, and moving the mouse to a new location on the screen.

- Resize a window
  by moving the mouse to the grow-box in the lower right corner of the window, holding down the mouse button, and dragging to a new location.

- Toggle between full-screen and a smaller size
  by clicking in the zoom-box in the upper-right corner of the window. This is called 'zooming' the window. The full-screen size can be customized. See the Windows chapter for details.

- Close a window
  by clicking the mouse in the close-box in the upper-left corner of the window, or by choosing Close from the File menu.

- A small cross appears on the left side of the file name in the window's title bar to indicate that the window has been altered since the last time the buffer was saved. The cross also appears next to the window's title in the Windows menu. A quick look at the Windows menu will let you know if *any* windows have been modified and not saved.

- You can cut, copy, and paste text between different windows (including the listener).

The Caret

The caret (also known as the *insertion-point* and sometimes known as the *cursor*) is shown by a flashing vertical bar between characters. As distinguished from text editors on other computers, the caret is not *on* a character—it is between two adjacent characters. BACKSPACE deletes the character to the left of the caret. Text insertion moves the caret to the right.

- Move the caret to a location
  by moving the mouse to a new location and clicking.

- Select a region
  by dragging the mouse through the region with the mouse-button down.

  A selection can be extended by holding the shift key and clicking.

  Regions larger than the window can be selected by holding down the mouse button and moving outside the edge of the window. The window will auto-scroll.

  You can select a Lisp expression by positioning the mouse at a close-parenthesis and double-clicking.

  You can select a word by positioning the mouse over the word and double-clicking. After selecting a word with a double-click, dragging the mouse extends the selection along word boundaries.

  Inserting text when there is a selection causes the selection to be replaced by the new text.

Editing Emacs Style

Fred commands have been defined with care to conform to Emacs conventions. The exceptions are primarily due to the Macintosh standards and keyboard limitations.

Control, Command, Option and Meta

Combining Macintosh capabilities and Emacs capabilities requires some creative use of the Macintosh keyboard. The Macintosh does not have a meta key, and only the most recent Macintoshes have a control key. To get around this, Allegro CL uses the following system:

- The option key is used as the meta key.
  To access the Macintosh's optional character set, type control-q, and then type the desired option-character. For the first character following control-q, option will not be treated as meta.

- The clover key and shift clover key are used for control and command.
  In Macintosh-mode, the clover key is used for command, and shift-clover is used for control. In Emacs-mode, the mapping is reversed. For details, see the note on the clover key in the chapter Getting Started.

The function names given are accessible as object functions defined for windows.

Documentation Conventions

C stands for Control (either clover or shift-clover. See the note on commands keys in the menubar section of this chapter.)

M stands for Meta (the option key)

current expression stands for the region selected by the user, if any. If no region is selected and the caret is next to a parenthesis, the current expression is between that parenthesis and the matching parenthesis. If no region is selected and the cursor is inside a symbol, the symbol is taken as the current expression. In other cases, the current expression is considered to be `nil`.

Help Functions

C- `(ed-help)` brings up the Fred help window. This window contains a list of all Fred keyboard commands available in the current comtab. It may be searched and printed.

C-= `(ed-what-cursor-position)` prints information about the current buffer.

M-. `(ed-edit-definition)` attempts to bring up the source code (and surrounding file) for the symbol surrounding the caret. If the symbol was defined from more than one source code the user is given a choice of definitions.
This function works for most forms defined with a call to `defxxx` with `*record-source-files*` non-`nil`.

C-x C-a `(ed-arglist)` attempts to print the argument list of the function bound to the symbol surrounding the caret.
This works for Common Lisp functions, and for most forms defined with a call to `defxxx` with `*record-source-files*` turned on.

C-x C-d (ed-get-documentation) attempts to print the documentation string of the function bound to the symbol surrounding the caret. This works for most forms defined with a call to defxxx with *save-doc-strings* turned on.

C-x C-i (ed-inspect-current-sexp) inspects the current expression.

Movement

LEFTARROW (ed-backward-char) moves the caret back one character.

RIGHTARROW (ed-forward-char) moves the caret forward one character.

UPARROW (ed-previous-line) moves the caret up one line.

DOWNARROW (ed-next-line) moves the caret down one line.

C-b (ed-backward-char) moves the caret back one character.

C-f (ed-forward-char) moves the caret forward one character.

M-b (ed-backward-word) moves the caret back one word.

M-f (ed-forward-word) moves the caret forward one word.

C-M-b (ed-backward-sexp) moves the caret back one expression.

C-M-f (ed-forward-sexp) moves the caret forward one expression.

C-a (ed-beginning-of-line) moves the caret to the beginning of the line.

C-e (ed-end-of-line) moves the caret to the end of the line.

C-p (ed-previous-line) moves the caret up one line.

C-n (ed-next-line) moves the caret down one line.

M-v (ed-previous-screen) scrolls towards the top of the text by a window-full and moves the cursor to the upper left corner of the window. The number of lines to be retained after scrolling is determined by *next-screen-context-lines*.

C-v (ed-next-screen) scrolls towards the bottom of the text by a window-full and moves the cursor to the upper left corner of the window. The number of lines be retained after scrolling is determined by *next-screen-context-lines*.

M-< (ed-beginning-of-buffer) moves the caret to the beginning of the buffer.

M-> (ed-end-of-buffer) moves the caret to the end of the buffer.

M-) (ed-move-over-close-and-reindent) moves the caret over the next close parenthesis and into position for typing the next Lisp expression.

C-x h (select-all) selects the entire buffer.

Insertion

RETURN (ed-self-insert) inserts a new line and puts the caret at the beginning of that line.

C-o (ed-open-line) inserts a new line without moving the caret.

TAB (ed-indent-for-lisp) re-indents the current line.

C-M-q (ed-indent-sexp) re-indents the current expression.

C-RETURN (ed-newline-and-indent) is equivalent to RETURN followed by TAB.

C-y (ed-yank) inserts (yanks) the current kill-ring string into the buffer at the caret. If there is a selected region, it is replaced with the inserted text.

M-y (ed-insert-killed-string-from-menu) brings up a menu of the strings in the kill-ring, allows the user to pick a string from it, and inserts it into the buffer at the caret. If there is a selected region, it is replaced with the inserted text.

C-q (ed-insert-quoted) allows access to the Macintosh optional character set. For a single keystroke following C-q, the Option key will not be translated into meta. For example, inserting the bullet sign (normally the Option-8 keystroke) is accomplished by the keystrokes C-q and M-8. By itself, M-8 would cause Fred to look for a command. C-q can also be used for inserting control characters—such as tabs—into buffers.

M-" (ed-insert-double-quotes) inserts a set of double quotes and puts the caret in between them.

M-# (ed-insert-sharp-comment) inserts #| |# and puts the caret in between them.

M-((ed-insert-parentheses) inserts a set of parentheses and puts the caret in between them.

M-U (ed-upcase-word) upper-cases the current word or selection. (Because of the design of the Macintosh, this should be typed with the shift-key held down.)

M-l (ed-downcase-word) lower-cases the current word or selection

M-c (ed-capitalize-word) capitalizes the current word or selection

C-t (ed-transpose-chars) transposes the two characters surrounding the caret. If there is a selection, the first character in the selection is transposed with the character before the selection.

Deletion

BACKSPACE (ed-rubout-char) deletes the character to the left of the caret.

M-BACKSPACE (ed-rubout-word) deletes the word to the left of the caret. If the caret is inside a word, only the portion of the word to the left of the caret is deleted.

C-d (ed-delete-char) deletes the character to the right of the caret.

M-d (ed-delete-word) deletes the word to the right of the caret. If the caret is inside a word, only the portion of the word to the right of the caret is deleted.

C-M-BACKSPACE (ed-delete-bwd-delimiters) deletes backward delimiters. This works when caret is behind (...) or "..." or #|...|# .

C-M-d (ed-delete-fwd-delimiters) deletes forward delimiters. Works when the caret is in front of (...) or "..." or #|...|# .

C-w (cut) deletes the current selection, adding it to the kill-ring.

M-w (ed-copy-region-as-kill) pushes current expression onto the kill-ring. The expression is not deleted from the buffer.

C-k (ed-kill-line) kills (deletes) the selected region or the remainder of the line containing the caret, adding it to the kill-ring. If the caret is at the end of a line, the following RETURN is deleted.

C-M-k (ed-kill-sexp) kills the current expression, adding it to the kill-ring.

Lisp Operations

ENTER (ed-eval-or-compile-current-sexp) evaluates the current expression. If *compile-definitions* is t (and there are no lexically apparent bindings) then definitions will be compiled rather than evaluated.

C-x C-c (ed-compile-top-level-sexp) compiles the current selection or the current top-level Lisp expression. The current top-level Lisp-expression is determined heuristically by searching backwards for an open parenthesis at the start of a line.

C-x C-e (ed-eval-current-sexp) evaluates the current expression.

C-m (ed-macroexpand-1-current-sexp) macroexpands the current expression with macroexpand. The result of each call to macroexpand-1 is printed in the listener.

C-x C-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 macroexpand.

C-x C-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.

C-M-SPACE (ed-select-current-sexp) selects the current expression.

Windows

C-s (wfind) brings up the search dialog box.

C-x C-s (window-save) saves the contents of the top Fred window to its associated disk file. If there is no file associated with the window, the user will be requested to supply a file name.

C-x C-w (window-save-as) saves the contents of the top Fred window to file specified by the user.

C-x C-v (edit-select-file) allows the user to select a text file and opens up a Fred window for editing that file.

Miscellaneous

The following keys are used by the Macintosh to implement a function key feature. These are invoked by typing clover-shift-*n*, where *n* is a number between 0 and 9. Four standard function keys are provided with the Macintosh. Additional function keys are available through commercial and other sources.

clover-shift-1 ejects the internal floppy disk.

clover-shift-2 ejects the external floppy disk.

clover-shift-3 save current screen as MacPaint® file.

clover-shift-4 if the current hardcopy device is a non-networked ImageWriter®, prints current window (including window-frame). If the caps-lock key is depressed, the entire screen is printed.

clover-. clover-period is used as the abort key. It can be typed to interrupt program execution.

In addition to the keyboard commands listed above, many menu items can be invoked through the keyboard. See the chapter Getting Started for details.

Object Lisp 3

Overview
Tutorial
- Objects as Containers
- Objects Inherit From Other Objects
- Object Environments are Composed of Frames
- Objects with Internal Procedures
- Multiple Inheritance
- Classes and Instances
- Advanced Topic: Objects and Scoping
- Object Variables and Lexical Variables

Object Lisp Functions
- Creating and Initializing Objects
- Communicating with Objects
- Managing Bindings and Definitions Inside Objects
- Managing Objects

Object Lisp

Overview

In traditional programming methods, a series of procedures is used to operate on a set of data. The procedures and the data are stored separately. Object-oriented programming combines procedures and data into discrete units called *objects*. An object contains data as well as procedures which know how to operate on the data. When you want to perform some operation on the data, you ask the object to do it. The object knows how, so you don't need to specify the operation in more detail. You can create multiple *instances* of an object; each one has its own data, but they all have the same procedures (i.e. they all know how to do the same things).

Object-oriented programming is a powerful metaphor for simplifying many programming tasks. Object-oriented programming increases *modularity*, so it helps keep program units from interfering with each other. It aids in hiding complexity, because once an object is implemented it can be used without extensive knowledge of its internal workings. It facilitates the re-use of code, because new objects can easily inherit the capabilities of old objects.

Allegro CL currently supports an object-oriented programming system (oops) called *Object Lisp*. Object Lisp is elegant, and very easy to use. It supports multiple-inheritance, a feature missing from many oops'.

The Common Lisp community is standardizing on an oops called CLOS (Common Lisp Object System). CLOS is largely derived from Common Loops, with additions from Flavors and Object Lisp. Future releases of Allegro CL will support CLOS. At that point native support of Object Lisp may be phased out, to be replaced by an implementation built on top of CLOS. The current implementation of Object Lisp has not been optimized for speed. It is suitable for implementing user-interface tools (its main function in Allegro CL), but in other areas performance may become a problem. We don't recommend using Object Lisp in large, time-critical portions of systems.

Tutorial

Objects as Containers

Objects act as containers for variables and procedures. The variables and procedures inside an object are local to the object, i.e. they do not interfere with global versions, and they do not interfere with versions inside other objects. For example, the variable `foo` could have the value `"I am Foo"` in the global environment, it could have the value `15` in one object, and it could have the value `(a b (c d foo) (x))` in another object. In a window object the procedure `show` could cause its argument to appear on the screen, while in a speaker object, `show` could send its argument to a speech synthesizer. Each object can have its own value for any variable, and its own definition for any procedure. Objects can access their own variables and procedures directly. They can also access the variables and procedures of other objects indirectly. The global environment can be considered the global object (also known as the root object). In most situations where an object should be specified, passing `nil` indicates the root object.

The following sample interaction shows how objects can be used to create private versions of variables

```
? (setq fourth-grader (kindof nil))
#<Object #322, a generic object>
```

Here we create a new object, using the function `kindof`. `kindof` creates and returns a new object based on its argument, which should be an object. If no the new object is based on the root object. We store the new object in the global variable `fourth-grader`. The `setq` returns the object, and we see its print representation.

To do something inside an object, you use the special form `ask`. `ask` takes at least two arguments: the first argument should be an object and the rest are Lisp expressions. `ask` evaluates the expressions in the environment of the object. Think of it as asking the object to evaluate the expressions.

```
? (ask fourth-grader (have 'teacher))
NIL
```

In this case, we are asking the object to have a variable. The function `have` creates a distinct binding for a variable within the object. It tells the object to have its own version of the variable, rather than using a global or otherwise inherited version of the variable. (Details on inheritance are given below).
We now ask `fourth-grader` to set the value of the variable `teacher`. The binding between the variable and value is internal to `fourth-grader`. It does not affect other bindings of the variable.

```
? (ask fourth-grader (setq teacher "Mrs. Marple"))
"Mrs. Marple"
```

We can ask `fourth-grader` to evaluate `teacher`, and we get back the correct value, the string `"Mrs. Marple"`.

```
? (ask fourth-grader teacher)
"Mrs. Marple"
```

If we try to evaluate `teacher` in the global environment, we get an error, because `teacher` does not have a global binding. It only has a binding inside the object `fourth-grader`.

```
? teacher
> Error: Unbound variable: TEACHER
> While executing: SYMBOL-VALUE
```

We can give `teacher` a global binding by assigning it a value at the global level. Once this is done, we have established two independent bindings for the variable `teacher`: one is global and one is local to the `fourth-grader` object. These two bindings do not interfere with each other. Changing the value of one does not change the value of another.

```
? (setq teacher "Hal")
"Hal"
? teacher
"Hal"
? (ask fourth-grader teacher)
"Mrs. Marple"
```

When we evaluate `teacher` in the global environment, we get back `"Hal"`. When we evaluate it in the environment of the `fourth-grader` object, we get back the value `"Mrs. Marple"`.

Objects Inherit from Other Objects

Objects do not only have access to their own variables and procedures. They also have access to *inherited* variables and procedures. When you create a new object, the new object is always based on one or more previously existing objects. We will initially discuss the simpler case, where an object inherits from a single parent.

```
? (setq school "Bronx Science")
"Bronx Science"
? school
"Bronx Science"
? (ask fourth-grader school)
"Bronx Science"
```

Here we create a global binding for the variable `school`. We see that `fourth-grader` has access to this binding. As stated above, `fourth-grader` inherits from the root object (i.e. the global environment). Because `fourth-grader` did not have its own binding for the variable `school`, it used the inherited binding.

```
? (ask fourth-grader (setq school "PS 110"))
"PS 110"
? (ask fourth-grader school)
"PS 110"
? school
"PS 110"
```

When we ask an object to set the value of a variable, it sets whichever binding it can access. `fourth-grader` does not have its own binding of `school`, so the `setq` affects the inherited binding. If we want `fourth-grader` to have its own binding of `school`, we must to ask it to `(have 'school)`.

```
? (ask fourth-grader (have 'school))
NIL
? (ask fourth-grader (setq school "MIT"))
"MIT"
? (ask fourth-grader school)
"MIT"
? school
"PS 110"
```

When we ask `fourth-grader` to have `school`, it creates its own binding for `school`. The new binding is internal to the object, and is said to *shadow* any inherited binding. All references to `school` in `fourth-grader`'s environment will now refer to this local binding.
A single object can use some inherited values, and some values of its own.

```
? (setq planet "Earth")
"Earth"
? (ask fourth-grader planet)
"Earth"
? (ask fourth-grader school)
"MIT"
? (ask fourth-grader teacher)
"Mrs. Marple"
```

In the second line, `fourth-grader` uses the inherited binding for `planet`, because it does not have its own binding. In the last two lines, `fourth-grader` uses its own bindings.

Besides creating new objects which inherit from the root object, we can create objects which inherit from other objects. This is where the power and utility of object-oriented programming becomes more evident.
We can create another object which inherits from `fourth-grader`. This new object will have access to all of `fourth-grader`'s environment; it will have access to all of `fourth-grader`'s bindings and all the bindings which `fourth-grader` inherits.

```
? (setq Amanda (kindof fourth-grader))
<object #367, a #322>
```

Here the argument to `kindof` is `fourth-grader`. We are creating a new object based on `fourth-grader` and storing this new object in the global variable `Amanda`. A short interaction shows us that because `Amanda` inherits from `fourth-grader` she has access to all the things which fourth-grader can access.

```
? (ask Amanda teacher)
"Mrs. Marple"                    ;fourth-grader's binding
? (ask Amanda school)
"MIT"                            ;fourth-grader's binding
? (ask Amanda planet)
"Earth"                          ;global binding
```

Just as fourth-grader's bindings are not accessible to its parent, Amanda's bindings are not accessible to her parent.

```
? (ask Amanda
    (have 'calculator "hp")) ;have can take another argument
"hp"
? calculator
> Error: Unbound variable: CALCULATOR
> While executing: SYMBOL-VALUE   ;no global binding for calculator
? (ask fourth-grader calculator)
> Error: Unbound variable: CALCULATOR
> While executing: SYMBOL-VALUE   ;no fourth-grader binding
? (ask Amanda calculator)
"hp"                             ;Amanda has a binding
```

This points out that inheritance flows in only one direction. `Amanda` inherits from `fourth-grader` (and, through `fourth-grader`, from the root object), but `fourth-grader` does not inherit from `Amanda`. `fourth-grader` therefore does not have access to `Amanda`'s bindings.

Object Environments are Composed of Frames

The above examples illustrate an object hierarchy; Amanda inherits from fourth-grader which in turn inherits from the global environment. In such a hierarchy, there is a simple rule for determining which binding is accessed by an object: the object will use the innermost binding. If the object has its own binding, that will be used. If not, the look-up will proceed to the parent, then to the parent's parent, and so on, all the way to the root object. As soon as one binding is found, it is used and the search stops. All further bindings are shadowed by the innermost binding, and are therefore inaccessible.

Technically, an object's environment is composed of a series of frames. The innermost frame contains the object's own bindings. The next frame out contains the bindings local to the object's parent. The next frame contains the bindings of the parent's parent, and so on. The outermost frame is the global environment. Whenever a look-up occurs, the frames are searched from innermost to outermost.

Objects with Internal Procedures

In all of the examples above, we have given objects internal variables, but not any internal functions. Objects can have private versions of both functions and variables. Function inheritance follows the same rules as variable inheritance.

The macro defobfun is used to define functions within objects.

```
? (defobfun (say fourth-grader) (sentence)
     (princ "*** ")
     (princ sentence)
     (princ " ***")
     (terpri))
SAY
? (say "hello")
> Error: Undefined function: SAY .
> While executing: EVAL        ;no global definition of say

? (ask fourth-grader (say "hello"))
*** hello ***
NIL
?
```

Here we define the function say inside of fourth-grader. Just as with object variables, this function definition is not accessible globally. It is only accessible from fourth-grader, and from objects which inherit from fourth-grader.

Because Amanda inherits from fourth-grader, she has access to fourth-grader's function definition of say, (just as she has access to fourth-grader's variable binding for teacher).

```
? (ask Amanda (say teacher))
*** Mrs. Marple ***
NIL
? (ask Amanda (say calculator))
*** Hp ***
NIL
?
```

(Note: Amanda has access to the definition of say, even though say was defined in fourth-grader *after* Amanda was created from fourth-grader. This is because the look-ups, for both values and functions, occur at run time.)

So far so good. Amanda can do the things that any fourth-grader can do. But she can actually do even more. One of the most powerful features of object oriented programming is that objects can modify inherited behavior. Of course, we could define say inside Amanda, thereby shadowing the inherited version of say and giving Amanda her own, completely new version of say. But a more efficient technique allows Amanda to use the inherited version, *and* add her own idiosyncrasies.

```
? (defobfun (say Amanda) (sentence)
      (usual-say sentence)
      (princ "I have an hp calculator")
      (terpri))
SAY
? (ask Amanda (say "hi there."))
*** hi there. ***
I have an hp calculator
NIL
?
```

The magic line in the above definition is (usual-say sentence). The construct usual-xxx is used to call the shadowed version of xxx. That is, it calls the version of the function which would have been called if the current version did not exist. In this case, say calls the shadowed version of say with the same argument that it was passed, and then performs some additional behavior. There are many other uses of usual, however. For example, a function can do some bounds checking, or other type of checking, before invoking its usual function. Or it can modify an argument or pass a completely different argument to the usual function. In addition, it can add behavior before and/or after the usual behavior, as in this example.

We can create another object which inherits from fourth-grader, and give it yet another definition of say:

```
? (setq doubly (kindof fourth-grader))
<object #400, a #322>
? (ask doubly (say teacher))
*** Mrs. Marple ***
NIL
? (defobfun (say doubly) (sentence)
      (usual-say sentence)
      (usual-say sentence))
SAY
? (ask doubly (say "hello"))
*** hello ***
*** hello ***
NIL
?
```

Here we have given `doubly` a slightly more complex version `say`. It calls the `usual-say` twice. Of course, `Amanda`'s version of `say`, and `fourth-grader`'s version of `say` are unaffected by this redefinition inside `doubly`.

```
? (ask Amanda (say "hi there"))
*** hi there. ***
I have an hp calculator
NIL
? (ask fourth-grader (say "I'm fourth-grader")
*** I'm fourth-grader ***
NIL
?
```

Multiple Inheritance

Multiple inheritance occurs when one object inherits from more than one other object—when an object has more than one parent.

In Object Lisp, an object can have a virtually unlimited number of parents. There are two general cases of multiple inheritance: the simple case, in which the characteristics of the parents do not have any overlap; and the more complex (but often more useful) case, in which there is overlap in the characteristics of the parents.

In the simple case there is no overlap between the functionality of the parents, so there is no possible ambiguity in variable and function reference. Some functions and variables are inherited from one parent, some from another.

It is often useful to create an object with multiple parents, where the parents *do* have some overlap in their functionality. In this case, the same function may have one definition in one parent, and a different definition in another. There is a simple rule for determining which binding is used when a look-up occurs in the child object. The frames of the parents are combined in a linear sequence, with duplicate frames removed. An example will illustrate this.

```
? (setq Bob1 (kindof Amanda Doubly))
<object #464, a #367, #400>
? (ask Bob1 (say "I am Bob1"))
*** I am Bob1 ***
*** I am Bob1 ***
I have an hp calculator
NIL
```

Here we create a new object based on both `Amanda` and `Doubly`. The frames inherited from `Amanda` are:

`Amanda`'s, `fourth-grader`'s, and the global frame.

The frames inherited from `Doubly` are:

- `Doubly`'s, `fourth-grader`'s, and again the global frame.

First we combine these two sequences of frames. This gives us the series of frames:

`Amanda`, `fourth-grader`, `root`, `Doubly`, `fourth-grader`, `root`.

The next step involves removing duplicate frames. Only the outer-most occurrence of each frame is retained. After the removal we have the frames:

`Amanda`, `Doubly`, `fourth-grader`, and `root`.

It is as if `Amanda` inherited from `Doubly` which inherited from `fourth-grader`, etc. When `Amanda`'s `say` calls `usual-say`, the `say` inside `Doubly` will be called. When `Doubly` calls `usual-say`, `fourth-grader`'s `say` will be invoked. The outermost frame is retained because inner definitions are often specializations which depend on being able call their shadowed counterparts.

The order in which parents are specified can have a significant effect on the behavior of the child object. In the example below we see that in an object that inherits from `Doubly` and `Amanda` in the inverse order, the `say` procedure behaves quite differently. Both cases can be understood by looking at the order of frames and tracing through the calls to `say` and `usual-say`.

```
? (setq Bob2 (kindof Doubly Amanda))
<object #4699, a #367, #400>
? (ask Bob2 (say "I'm Bob2"))
*** I'm Bob2 ***
I have an hp calculator
*** I'm Bob2 ***
I have an hp calculator
NIL
```

A final word about these examples. They all work because the specialized functions make calls to their `usual` counterparts. Given the example of `Bob1`, `Doubly`'s behavior would not have been exhibited if `Amanda`'s `say` had not called `usual-say`. If we look at `Bob1`'s set of frames—`Amanda`, `Doubly`, `fourth-grader`, and `root`—it becomes clear why. If `Amanda`'s `say` did not call `usual-say`, the call to `say` would just stop inside `Amanda`. Any function which does not call its `usual` counterpart cuts off the inheritance path. This can be useful in some situations (for example, in implementing filters), but it can cause unexpected behavior in others. (Don't worry about calling `usual-`*foo* if there is no inherited version. Object Lisp simply ignores such calls.)

Classes and Instances

In the examples above, the evolution of objects from `fourth-grader` through `Amanda` and `Doubly`, to the `Bobs` was exploratory and somewhat haphazard, and does not show how a typical object hierarchy would be set up in a program.

When using objects there is usually a distinction made between *classes* and *instances*. Object Lisp does not enforce this distinction; you are free to create and modify objects incrementally, as we did above. This is very useful when learning, and when prototyping a system. However, in production work the class/instance paradigm often leads to more organized programs.

A *class* describes a type of object. For example, you could have a class of spreadsheet windows. From this class, multiple *instances* can be created; you might want to have many spreadsheet windows on the screen at once. It is normal for the class to contain procedures (i.e. behavior), while instances contain *instance variables* (i.e. data) for maintaining the individual state of the instance. In the spreadsheet example, the class would contain the code for entering information and performing calculations, while each instance would have unique instance variables holding the contents of the cells, screen position, etc.

All objects, both classes and instances, are normally created from classes. You would not typically create a new object based on an instance. In general, classes are created at the start of a program, and instances are created for use during the course of a program.

Initializing Instances and the `exist` Procedure

In creating object instances, it is usually necessary to perform some initialization. The initialization may give the new instance its own bindings for instance variables and set these variables to initial values; it also may create more complex data structures which need to be owned by the instance.

For example, each instance of the spreadsheet window class will need to be associated with its own spreadsheet data, and with its own window on the screen. In this case, the initialization of an instance would involve creating a window and some instance variable bindings and filling the instance variables with initial data. These variables cannot be stored in the class, because then there would be no difference between one spreadsheet and another. Each instance must have its own bindings for the instance variables, and must have its own window. These bindings are created by asking each instance to `have` the variables. This process should occur when the instance is initialized.

Object Lisp provides a standard protocol for initializing instances. This is the `exist` procedure. Any initialization common to all instances of a class should be performed by `exist`. As stated above, this initialization usually involves giving the instance its own bindings of variables, and perhaps setting these bindings to initial values.

The function `oneof` is supplied to facilitate the creation of instances. `oneof` does two things. First it calls `kindof`, creating a new object, and then it asks the new object to `exist`. It then returns the newly created and initialized object. In general, most `exist` procedures will call `usual-exist`; this guarantees that no inherited steps in the initialization process are left out.

By convention, `exist` takes a single argument, called an *init-list*. This argument, again by convention, is a list of alternating keys and values. The keys are generally keywords, and the values may be any Lisp data. The values are extracted from the init-list with the Common Lisp function `getf` (for this reason, init-lists should always have an even number of elements). When

calling `usual-exist`, the init-list or a modified version of the init-list is generally passed as an argument.

`exist` is simply an object-function that can be defined for a class. When instances are created from a class, the `exist` procedure will be run. Because the object is new, and doesn't have its own `exist` procedure, it will naturally run the one it inherits from its class.

```
? (defobfun (exist fourth-grader) (init-list)
    (have 'nick-name (getf init-list :nick-name "Anonymous"))
    (have 'age  (getf init-list :age 10)))
EXIST
?
```

When we create new instances of `fourth-grader` with `oneof`, they will be initialized to have a `name` and `age` instance variable (note that `oneof` takes a rest argument, which it passes to `exist` as a list.)

```
? (setq franklin (oneof fourth-grader
                        :nick-name "frankie" :age 11))
<object #476, a  #322>
? (ask franklin (print nick-name) (print age) (terpri))

"frankie"
11
nil
? (setq bettina (oneof fourth-grader :nick-name "betsy"))
<object #481, a #322>
? (ask bettina (print nick-name) (print age) (terpri))
"betsy"
10                                  ;betsy gets the default age
nil
?
```

Advanced Topic: Objects and Scoping

This section discusses scoping in objects, especially cases where objects need to communicate the values of their instance variables to other object. You may want to skip it until you have some experience with programming in Object Lisp.

When you ask an object to evaluate an expression, the expression is evaluated in the environment of the object being asked, not in the environment of the asking object. This means that variable references inside the expression will generally refer to bindings within the object being asked. The following example shows a common scoping difficulty which occurs when one object asks another object to do something.

```
? (setq ob-1 (kindof))
#<Object #500, a generic object>
? (ask ob-1 (have 'x-var 10))
10
? (ask ob-1 x-var)
10
? (setq ob-2 (kindof))
```

```
#<Object #501, a generic object>
? (ask ob-2 (have 'y-var 20))
20
? (ask ob-1 (ask ob-2 (+ x-var y-var)))
> Error: Unbound variable: X-VAR .
> While executing: SYMBOL-VALUE
```

The last line caused an error, because the reference to `x-var` was evaluated inside `ob-2`, and `ob-2` has no binding—its own or inherited—for `x-var`. To get around this problem, the last line could be rewritten:

```
? (ask ob-1 (+ x-var (ask ob-2 y-var)))
30
```

A more insidious problem occurs when `ob-2` *does* have a binding for `x-var` (or perhaps inherits such a binding), but we want the reference to refer to `ob-1`'s binding of `x-var`.

```
? (ask ob-2 (have 'x-var 20))
20
? (ask ob-1 (ask ob-2 (+ x-var y-var)))
40
?
```

This situation does not signal an error, but gives incorrect results. The thing to keep in mind is that variable look-ups always occur in the environment of the innermost `ask`. Again, the proper functionality could be achieved by rewriting the last line

```
? (ask ob-1 (+ x-var (ask ob-2 y-var)))
30
?
```

Object Variables and Lexical Variables

The solutions given above are usable because the function + is defined in both both `ob-1` and `ob-2`. A more difficult situation arises when ob-1 wants to pass a value to be acted upon by a function defined in object-2. Of course, it is possible to write

```
(ask ob-1
    . . .                          ;miscellaneous code here
    (ask ob-2 (frob (ask ob-1 x-var) y-var))
            ;frob is only defined in ob-2
    . . .)
```

But this is somewhat ungainly. There is a better way of communicating the values of instances variables between objects. The method relies upon the fact that lexical bindings take precedence over object bindings, and they are always visible (unless, of course, they are shadowed by other lexical bindings). Using lexical bindings, the above code could be rewritten

```
(ask ob-1
 (let ((my-x-var x-var))
   . . .                                ;miscellaneous code
   (ask ob-2 (frob my-x-var y-var)) ;lexical binding is accessible
   . . .))
```

Object bindings are only accessed for free references. If there is a lexical binding for a symbol, that will be used.

```
? (ask ob-2 (have 'y-var 100))
Y-VAR
? (ask ob-2
     (print (+ y-var 1))
     (let ((y-var 200))
      (+ y-var 1)))
101
201
?
```

Because of the lexical binding of `y-var` (from the let statement), `ob-2`'s object binding was not used. This technique makes it possible for an object to pass the values of its instance variables to another object. The giving object first binds the values to lexical variables, and then uses these in the call to the receiving object. The giving object doesn't need to worry about the possibility that the receiving object has object variables with the same name. As an added benefit, lexical bindings are accessed more quickly than object bindings.

Lexical bindings are established for arguments to functions, as well as by let statements.

```
? (defun lex-show (x1)
     (ask ob-1 (princ x1)) (terpri)
     (ask ob-2 (princ x1)) (terpri))
LEX-SHOW
? (lex-show "x1 is now referenced lexically")
x1 is now referenced lexically
x1 is now referenced lexically
NIL
?
```

Object Variables and Special Variables

Object variables have dynamic scope. For example:

```
? (setq foo (kindof))
#<Object #507, a generic object>
? (ask foo (have 'bar 10))
10
? (defun baz (n) (+ n bar)
BAZ
? (baz 5)
> Error: Unbound variable: BAR
> While executing: BAZ
? (ask foo (baz 5))
15
```

The first call to `baz` fails because of the free reference to `bar`. However, the second call succeeds, because the call was made inside `foo`. Inside `foo`, `bar` is bound, and the free reference in `baz` uses this binding.

The dynamic nature of object variables introduces possible conflicts with special variables. For this reason, an error will be signalled if a program attempts to use a variable as both a special variable and an object variable.

Object Lisp Functions

In the functions below, which require objects as arguments, `nil` should be passed to indicate the root object.

Creating and Initializing Objects

`kindof` `&rest` *parents* [*Function*]
creates and returns a new object which inherits from *parents*. *parents* should all be objects. If no *parents* are given, the new object will inherit from the root object.

`oneof` `&optional` *parents* `&rest` *key-val-pairs* [*Function*]
creates and returns a new object using `kindof`, and then asks the object to run `exist` with *key-val-pairs* as the argument. *parents* should be either an object or a list of objects from which the new object will inherit. If no arguments are passed to `oneof`, the new object will inherit from the root object.

`defobject` *name* {*parents*}* [*Macro*]
creates a new object based on *parents* using `kindof` or `remake-object` (see below), binds the new object to the global symbol *name*, and gives the new object an instance variable `object-name`, whose value is the symbol *name*. The new object is consed onto the instance variable `object-children` in any of the parents that were created with `defobject`. If no *parents* are specified, then the new object will inherit from the root object.

If *name* is not currently globally bound to an object, the new object is created with `kindof`. If *name* is currently globally bound to an object, then the new object is created with `remake-object`. *name* is returned.

`remake-object` *object* `&rest` *parents* [*Function*]
remakes *object*, leaving its own frame intact, but making it inherit from *parents*, rather than the objects from which it previously inherited. If no *parents* are given, the object will be made to inherit directly from the root object. The new inheritance path is propagated to any objects which inherit from *object*. `exist` is not called. The variables and functions owned by the object are not affected.

`exist` *init-list* [*Function*]
`exist` is the function conventionally used to initialize new instance objects. `exist` is used to give instance variables to a new instance and to perform other required initializations. `exists` should generally call `usual-exist` (unless they have a specific reason for not doing so). `exist` is called by `oneof`.

init-list should be a list of alternating keys and values. The keys should be symbols or keywords. *init-list* is used to specify the initial state of the new object. `getf` can be used to conveniently extract values from an init-list (for this reason, init-lists should always have an even number of elements). `init-list-default` (described below) is also useful for manipulating init-lists. Note that nothing is automatically done with an init-list. Each object must have a definition of `exist` that performs actions appropriate to the contents of the init-list it receives.

`init-list-default` *init-list* {*key expression*}$^+$ [*Macro*]
is used to add default values to an init-list.

init-list should be a list of alternating keys and values. For each *key* and *expression*, if *key* is a member of *init-list* (using `eq` for comparison), then *expression* is not evaluated and *init-list* is not altered; if *key* is not a member of *init-list*, then *expression* is evaluated, and result and *key* are appended to *init-list*. The modification is not destructive; the resulting list is returned.

Communicating with Objects

`ask` *object* {*form*}$^+$ [*Macro*]
evaluates *forms* inside *object*. It returns the value returned by the last *form*. Multiple value returns are supported.

`talkto` `&optional` (*object* `nil`) [*Function*]
changes the current object to *object*. If *object* is not specified, the root object is used. Future evaluation will take place in *object*'s environment (until the execution of another `talkto`). `talkto` should only be used from the top-level read-eval-print loop; inside programs, `ask` should be used instead. `talkto` returns the *previously* current object.

Whenever the current object is not the root object, the listener question-mark prompt is preceded by the current object's number (from its `object-license`). This is done to remind you that forms typed in are no longer evaluated in the global Lisp environment.

Managing Bindings and Definitions Inside Objects

`defobfun` (*name* [*object*]) *lambda-list* {*declaration* | *doc-string*}* {*form*}* [*Macro*]
`defobfun` *name lambda-list* {*declaration* | *doc-string*}* {*form*}*
is used to define and name a function within an object. It causes *name* to be bound in *object* to the function `(lambda` *lambda-list* {*declaration* | *doc-string*}* {*form*}*`)`. Definitions of *name* in other objects are not affected. Within the *forms* `usual-`*name* may be called. Such calls will invoke the inherited version of *name*, i.e. the definition of *name* which would have been called had there been no binding for *name* within *object*.

If *object* is not specified the parentheses surrounding *name* may be omitted; the definition will be created in the root object

It is an error to `defobfun` a function which has previously been defined with `defun`, and it is an error to `defun` a function which has previously been defined with `defobfun`. In this way the name-spaces of global and object-bound functions are kept separate.

symbol-value *symbol* [*Function*]
symbol-function *symbol* [*Function*]
These Common Lisp functions are extended to work with the object system. Both functions work properly with `setf`.

`symbol-value` returns the first accessible object binding or special binding of *symbol*. The search will proceed up the inheritance path of the current object. If there is no object binding of *symbol*, then the value of any special binding will be used.

`symbol-function` returns the first accessible object or global function binding of symbol. The search will proceed up the inheritance path of the current object.

set *symbol new-value* [*Function*]
setq {*symbol new-value*}* [*Special Form*]
fset *symbol function* [*Function*]
fset-globally *symbol function* [*Function*]
`set` and `setq` are extended to work with the object system. `set` makes an assignment to the first accessible object or special binding of *symbol*, using the same look-up algorithm as `symbol-value`. If *symbol* is unbound, a global binding is created.

`setq` performs the same action as `set`, except that it takes any even number of arguments, does not evaluate its *symbol*s, and recognizes lexical bindings of its *symbol*s (whereas `set` does not).

`fset` is analogous to `set`, except it affects the functional binding of *symbol*.

`fset-globally` always sets the global definition of *symbol* and makes it illegal to ever create an object binding for it. It is analogous to proclaiming a variable `special`. `defun` uses `fset-globally`.

The *new-value* or *function* is returned.

nfunction *name lambda-expression* [*Special Form*]
is equivalent to `(function` *lambda-expression*`)`, except that it lets the compiler and evaluator associate *name* with the function. The name lets otherwise functions work with usual mechanism. It is also used in the error system.

name must be a symbol; *lambda-expression* must be a lambda expression.

In the following example, we cannot call `usual-double` from an anonymous compiled function. We have to call it from a compiled function with the name `double`.

```
? (defobject foo)
foo
? (defobfun (double foo) (x) (+ x x))
double
? (defobject bar foo)
bar
? (defobfun (double bar) (x) (usual-double (+ x x)))
double
? (ask bar (double 1))
```

```
4
? (ask bar (fset 'double #'(lambda (x) (usual-double x))))
#<An Anonymous Compiled Function>
? (ask bar (double 1))
> Error: Undefined function: usual-double .
> While executing: ask
? (ask bar (fset 'double
                 (nfunction double
                            (lambda (x) (usual-double x)))))
#<Compiled Function DOUBLE>
? (ask bar (double 1))
2
```

have *symbol* `&optional` *value* [*Function*]
fhave *symbol function* [*Function*]
`have` creates a value binding for *symbol* in the current object and sets the binding to *value* (or `nil` if *value* is not supplied). `fhave` creates a function binding for *symbol* in the current object and sets it to *function*. *value* or *function* is returned. `defobfun` uses `fhave`.

makunbound *symbol* [*Function*]
fmakunbound *symbol* [*Function*]
These Common Lisp functions are extended to work in the object system. They will delete the current object's own value or function binding of *symbol*. Inherited bindings are not affected. If the current object does not have its own value or function binding of *symbol*, no action is taken. *symbol* is returned.

makunbound-all *symbol* [*Function*]
fmakunbound-all *symbol* [*Function*]
`makunbound-all` deletes all the value bindings of *symbol*, in any object including the root object. `fmakunbound-all` performs the same operation on function bindings. These are very dangerous operations. If used without care, they can delete necessary system bindings and crash the system.

ownp *symbol* [*Function*]
fownp *symbol* [*Function*]
`ownp` returns `t` if *symbol* has a value binding in the current object (inherited bindings are excluded); otherwise, it returns `nil`. `fownp` performs the same operation on function bindings.

boundp *symbol* [*Function*]
fboundp *symbol* [*Function*]
These functions are extended to work with the object system. `boundp` is true if *symbol* has a value binding which is accessible from the current object. `fboundp` is true if *symbol* has a function-binding which is accessible from the current object, or if it is a macro or a special form. Inherited bindings are included. (Lexical bindings are, of course, not included).

bound-anywhere-p *symbol* [*Function*]
fbound-anywhere-p *symbol* [*Function*]
`bound-anywhere-p` is true if *symbol* is bound anywhere in the system, otherwise it returns `nil`. `fbound-anywhere-p` is true if symbol has a function binding anywhere in the system or was ever defined inside an object, otherwise it returns `nil`. Note that unlike the Common Lisp function `fboundp`, `fbound-anywhere-p` does not check for macro and special form definitions. (These functions do not recognize lexical bindings; they only recognize object and special bindings).

where *symbol* [*Function*]
fwhere *symbol* [*Function*]
`where` returns the object containing the first accessible value binding of *symbol*, or `nil` if *symbol* has no currently accessible binding. `fwhere` returns the object containing the first accessible function binding of *symbol*, or `nil` if *symbol* has no currently accessible function binding. The look-up algorithm is the same as that used by `symbol-value` and `symbol-function`, respectively.

Managing Objects

self [*Function*]
returns the current object. The returned value can, for example, be lexically bound, passed to functions, or passed to objects.

typep *thing specifier* [*Function*]
extended to work with the object system if specifier is an object. `typep` returns `t` if *thing* inherits from *specifier*, otherwise returns `nil`. (Note: this means that *specifier* must be an ancestor of *thing*, not necessarily an immediate parent of *thing*).

`typep` is the one function which does not accept `nil` as a synonym for the root object. To find out if *thing* is an object, *specifier* can be the symbol `object` (alternatively, use the function `objectp` instead of `typep`).

If either *thing* or *specifier* is not an object, the normal rules for evaluating `typep` take place.

Examples:

```
? (typep *window* 'object)
T
? (typep *window* 'number)
NIL
? (typep *window* nil)
NIL
? (typep (car (windows)) *window*)
T
? (typep (front-window) *menu*)
NIL
? (typep (car (windows)) *stream*)
T
```

objectp *thing* [*Function*]
returns non-nil if thing is an object, otherwise returns nil.

object-name [*Instance Variable*]
Every object created with defobject is given an instance variable object-name. The value of object-name will be the symbol to which the new object is being bound. Objects created with oneof and kindof may be given a binding of object-name explicitly. The system uses object-name in the print representation of an object and its children. If a program binds or alters object-name, it should ensure that the new value can be easily displayed in the object's print representation.

object-children [*Instance Variable*]
This instance variable is maintained for objects created by defobject. It contains a list of objects which have been created from the object with defobject. Objects created from the object with kindof and oneof are not included on this list. (Note: because such children are pointed to by their parents, the children will not be garbage-collected until the parents are garbage-collected.)

object-parents *object* [*Function*]
returns a list of *object*'s immediate parents. These are the objects which were passed to kindof, oneof, defobject, or remake-object when *object* was created.

object-ancestors *object* [*Function*]
returns a list of *object*'s ancestors, i.e. its entire inheritance path. Duplicate references are removed, so that the list returned is the one used when function and variable look-up takes place in *object*.

print-self &optional *stream* [*Function*]
is used to print objects. The print representation of an object will vary, depending on whether the object has an object-name, and whether its parent(s) have object-names. The print representation is not readable. If *stream* is not specified, the object will be printed to *standard-output*. print-self may be shadowed.

In the unshadowed form of print-self, the format for print representations is:

#<object #*n*, [*name*,] a {*parent-name-or-number*,}+>

Examples:

```
? (defobject fourth-grader)
FOURTH-GRADER
? fourth-grader
#<object #45, FOURTH-GRADER, a generic-object>
? (setq foo (kindof))
#<object #47, a generic-object>
? (defobject Amanda fourth-grader)
AMANDA
? amanda
#<object #50, AMANDA, a fourth-grader>
? (defobject Herbert foo)
HERBERT
```

```
? herbert
#<object #51, HERBERT, a #47>
? (setq bar (kindof foo))
#<object #53, a #47>
? (defobject baz Herbert foo)
BAZ
? baz
#<object #54, baz, a Herbert, #47>
? (setq bim (kindof foo Herbert))
#<object #55, a #49, HERBERT>
? (ask foo (have 'object-name "foobird"))
"foobird"
? foo
#<object #49, "foobird", a generic-object>
? baz
#<object #54, BAZ, a "foobird", HERBERT>
```

object-license *object* [*Function*]
returns *object*'s license number. Whenever an object is created, it is given a `object-license`. `object-licenses` are integers, generated sequentially by the Lisp operating system (though there may appear to be gaps in the numbers due to objects generated by the run time system). `object-licenses` are used in print representations, and are sometimes useful for interactive debugging. Programs should not depend on them, because they will almost certainly change from one programming session to another.

The `object-license` of the root object is 0.

license-to-object *integer* [*Function*]
returns the object whose `object-license` is *integer*, or `nil` if no such object exists. This function should be used for single-session debugging only.

next-license-to-object *integer* [*Function*]
returns the object with the smallest `object-license` greater than *integer*, or `nil` if no such object exists.

highest-license-number [*Function*]
returns the highest `object-license` used by the system so far.

do-all-objects (*var result-form*) {*form*}* [*Macro*]
goes through all existing objects, evaluating *forms* with *var* bound to each object in turn. When all of the objects have been gone through, *result-form* is evaluated and its result is returned. While *result-form* is being evaluated, *var* is bound to `nil`. Objects created during the evaluation of *forms* may or may not be included in the iteration.

Example:

```
;;who-fowns returns a list of all the objects which have their
;; own definition of a given function
```

```
(defun who-fowns (sym &aux return-list)
   (do-all-objects (ob)
      (when (ask ob (fownp sym))
         (push ob return-list)))
   return-list)

? (who-fowns 'window-close)
(#<Object #164, ccl::*step-fred-window*, a *fred-window*>
 #<Object #84, a *dialog*>
 #<Object #82, a *dialog*>
 #<Object #68, *dialog*, a *window*>
 #<Object #10, ccl::*da-window*, a *window*>
 #<Object #9, ccl::*listener*, a #2,
                               ccl::*selection-stream*,
                               *fred-window*>
 #<Object #8, *fred-window*, a *window*>
 #<Object #7, *window*, a *stream*>)
```

do-object-variables (*var object result-form*) {*form*}+ [*Macro*]
goes through all the variables owned by *object*, evaluating *forms* with *var* bound to the name of each variable in turn. When all the variables have been gone through, *result-form* is evaluated and its result is returned. While *result-form* is being evaluated, *var* is bound to `nil`. The effect of creating or removing bindings during the evaluation of *forms* is undefined.

do-object-functions (*var object result-form*) {*form*}+ [*Macro*]
goes through all the functions owned by *object*, evaluating *forms* with *var* bound to the name of each function in turn. When all the variables have been gone through, *result-form* is evaluated and its result is returned. While *result-form* is being evaluated, *var* is bound to `nil`. The effect of creating or removing bindings during the evaluation of *forms* is undefined.

Macintosh Basics 4

Overview
Points
Font Specs
Turnkey Dialogs
Miscellaneous

Macintosh Basics

Overview

This chapter describes some data formats and functions used in Allegro CL's implementation of Macintosh features, points and font-specs. It also describes some turnkey dialogs built into the Allegro CL system.

Points

Points are used throughout Allegro CL for representing two-dimensional data. The most common use of points is in graphics operations which require a width and height (e.g. the size of a window), or a horizontal and vertical coordinate (e.g. the position of a dialog-item within a dialog).

2-dimensional points are often stored in 31-bit fixnums. This is known as the "encoded form." The low-order 16 bits correspond to the horizontal dimension, and the high-order 15 bits correspond to the vertical dimension. Both dimensions are signed This representation makes point manipulation very efficient; creating points does not cons, and `eq` can be used to compare points (in Allegro CL, `eq` can be used to compare fixnums).

Many functions which take a point as an argument can take it as two coordinates (`h` and `v`) or a single fixnum 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 fixnum.

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

`point-string` *point* [*Function*]
returns a string representation of *point.*

```
? (point-string 4194336)
"#@(64 32)"
```

`point-h` *point* [*Function*]
returns the h-coordinate of *point.*

```
? (point-h 2097184)
32
```

`point-v` *point* [*Function*]
returns the v-coordinate of *point.*

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

make-point *h v* [*Function*]
returns a point constructed from horizontal and vertical coordinates *h* and *v*. If *v* is `nil`, *h* is assumed to be a point, and is returned unaltered. `make-point` signals an error if *h* and *v* are not fixnums (or `nil` for *v*).

add-points *point-1 point-2* [*Function*]
returns a point that is the result of adding *point-1* and *point-2*. (Points cannot be added with the generic addition function because of possible overflow between the low and high words).

subtract-points *point-1 point-2* [*Function*]
returns a point that is the result of subtracting *point-2* from *point-1*. (Points cannot be subtracted with the generic subtraction function because of possible overflow between the low and high words).

Font Specs

Font information is given in the form of font-specs. A font spec is an atom or list of atoms giving the font-name and/or font-size and/or font-styles and/or transfer-mode.

The font-name is a string. It should correspond to a font available in the system file. You can find out which fonts are available by examining the variable `*font-list*`. Font names are not case sensitive.

The font-size is a fixnum in the range 1-127.

The font-style is one or more of the following style keywords: `:plain`, `:bold`, `:italic`, `:underline`, `:outline`, `:shadow`, `:condense`, `:extend`. These keywords are the keys in an association list in the global variable `*style-alist*`.

The transfer-mode should be one of the following transfer-mode keywords: `:srcCopy`, `:srcOr`, `:srcXor`, `:srcBic`, `:srcPatCopy`, `:srcPatOr`, `:srcPatXor`, `:srcPatBic`. (See the pen-mode section in the addendum on Quickdraw for a description of the transfer modes). These keywords appear in the global variable `*pen-modes*`.

An error will be signalled if more than one name, more than one size, or more than one transfer mode are given in a single font-spec. Multiple font-styles are allowed.

The following are examples of legal font-specs:

```
"New York"
"nEw YOrk"
(9 "Monaco")
("Monaco" :extend :shadow 57 :srcpatcopy)
:srccopy
:outline
(12 :srccopy)
```

font-list [*Variable*]
a list of all the fonts installed in the current Macintosh operating system.

real-font &optional *font-spec* [*Function*]
returns t if *font-spec* corresponds to a font/font-size that actually exists in the system (i.e. is not a calculated font). The font-styles and transfer mode are not significant. If *font-spec* is not supplied, the font-spec of the current window object (if any) is used, otherwise the font-spec of the current grafport is used.

string-width *string* &optional *font-spec* [*Function*]
returns the width in pixels of *string*, as if it were displayed using the font, size and style of *font-spec*. If *font-spec* is not supplied, the font-spec of the current window object (if any) is used, otherwise the font-spec of the current grafport is used.

font-info &optional *font-spec* [*Function*]
returns four values that represent the ascent, descent, widmax, and leading in pixels of *font-spec*. The ascent is distance from the baseline to the top of the font, descent is the distance from the baseline to the bottom of the font, widmax is the maximum width for characters in the font, and the leading is the suggested spacing between lines. Only the font and font-size are used in the calculation. The font-styles and transfer mode are not significant. If *font-spec* is not supplied, the font-spec of the current window object (if any) is used, otherwise the font-spec of the current grafport is used.

Turnkey Dialogs

Allegro CL provides four pre-designed dialogs for use by applications. For instructions on how to customize or make more complex dialogs, see the chapter Dialogs.

For all of the following dialogs, choosing Cancel causes a throw to :cancel. This throw may be caught by user code, otherwise it will cause a return to top-level, or if it occurs within event-processing, the execution of the interrupted program will continue.

message-dialog *message* &optional *position size* [*Function*]
displays a dialog box holding the message, with a single button containing the text OK. The function will return when the user clicks OK. An optional *position* and *size* may be specified.

y-or-n-dialog &rest *format-args* [*Function*]
puts up the standard Macintosh Yes, No, Cancel dialog. The message displayed in the dialog will be a string resulting from applying format to *format-args*. If the user clicks Yes, t is returned; if No, nil is returned; if Cancel, the dialog throws to :cancel.

get-string-from-user &optional *message initial-string ok-string cancel-string position size* [*Function*]
prompts the user for a string, which it returns. The dialog will display the string *message* as the message in the dialog. *initial-string* can specify the default string in the editable-text item of the dialog. *ok-string* and *cancel-string* specify the text to be placed in the ok and cancel buttons. *position* and *size* may be used to set the position and size of the dialog.

If the user clicks OK, the contents of the editable-text are returned. If the user clicks Cancel, a throw to :cancel is performed.

choose-font-dialog `&optional` *font-spec* [*Function*]
displays a dialog showing available fonts, sizes and styles. If the user clicks OK, the chosen font information is returned as a font-spec. If the user clicks Cancel, a throw to `:cancel` is performed.

font-spec may be given as an argument to set the initial values displayed when the dialog appears. For example, when setting the font of a window, the choose-font dialog should reflect the font, size, and style currently being used by the window.

Miscellaneous

ed-beep [*Function*]
makes a short beeping sound, often used by the Macintosh to issue warnings. `ed-beep` can also be useful when running diagnostics (i.e., if it beeps, you know that part of the got executed).

Menus 5

Overview
Menubars
Menus
Menu-items

Menus

Overview

The standard Macintosh user interface displays a menubar at the top of the screen. The menubar displays the titles of several menus. When the user clicks on a menu title, the menu pulls down, displaying the titles of its menu-items. The user can then move the mouse to select one of the menu-items.

In Allegro CL menus and menu-items are objects. A menubar is simply a list of menus. You can maintain several such lists, but only one will be the current menubar. Only the current menubar is displayed and useable.

A menu object is created from the class `*menu*`. A menu will not be displayed (and will therefore be unusable) until it is added to the current menubar with the function `menu-install`. Once installed, a menu will appear in the current menubar until it is de-installed. The menu-title may be changed, and menu–items may be added or removed regardless of whether the menu is installed. Menus may also be disabled; an installed disabled menu appears grayed-out and its items cannot be selected.

A menu-item has five characteristics. These are the title (used for displaying the menu-item when its menu is pulled down), the keyboard equivalent for the menu-item (if any), the font-style to be used in displaying the menu-item title, whether the menu-item has a check mark (or other character) next to it when it is displayed, and whether the menu-item is enabled or disabled (disabled menu-items are displayed grayed-out and cannot be selected).

Most menu-items have a definition for the object function `menu-item-action`. This object function is called whenever the user selects the menu-item. Menu-items may also have definition for the object function `menu-item-update`. `menu-item-update` is called for every menu-item when the user clicks in the menubar or types a keyboard equivalent. `menu-item-update` functions let menu-items adjust to program context (for example, the **Save** menu-item is only enabled when the top window is an editor buffer which has been modified). The usual `menu-item-update` does nothing.

If a menu-item has the keyboard equivalent <x>, then the menu-item's action may be invoked by typing command-<x>. In Emacs mode, command-<x> is typed as clover-shift-<x>. In Macintosh mode, command-<x> is typed as clover-<x>. For a complete description of Emacs and Macintosh modes, see the chapter on Getting Started.

It is often desirable to separate menu-items within a menu into groups. This is done by placing a dotted line between the groups of items. A menu-item whose title is the string `"-"` will appear as a dotted line which cannot be selected.

Whenever a menu-item is selected by the user (either by mousing a menu or using a keyboard equivalent) the current program is interrupted and the menu-item's definition of `menu-item-action` is run. The result of the function is not used. When the function returns, normal program operation resumes.

During the execution of `menu-item-action`, event interaction is disabled. If a menu-item is used to initiate a program, the program should be inserted into the normal read/eval/print loop with `eval-enqueue`. This will allow event processing during program execution.

Many menu-items perform their actions on the front window. Such actions can find the front window with the `front-window` function (see chapter Windows for details).

Menubars

menubar [*Function*]
set-menubar *new-menubar* [*Function*]
`menubar` returns the current menubar (i.e. a list of all the installed menus).
`set-menubar` installs a new menubar. *new-menubar* should be a list of menus. All the currently installed menus are asked to `de-install`, and each of the menus in *new-menubar* is asked to `install`. *new-menubar* may be `nil`, in which case the menubar is simply cleared.

Example:

```
? (setq foo (menubar))
(#<Object #15, "", a *menu*>
 #<Object #18, "File", a *menu*>
 #<Object #29, "Edit", a *menu*>
 #<Object #42, "Eval", a *menu*>
 #<Object #48, "Tools", a *menu*>
 #<Object #60, "Windows", a *menu*>)
? (set-menubar (list (car foo) calc-menu))
(#<Object #15, "", a *menu*>
 #<Object #180, "Calculate", a *menu*>)
? (menubar)
(#<Object #15, "", a *menu*>
 #<Object #180, "Calculate", a *menu*>)
```

default-menubar [*Variable*]
the default menubar. This is the menubar which appears when you begin a Allegro CL session. It may be installed using `set-menubar` if you want to get a fresh start after hacking things up.

Example:

```
? (set-menubar *default-menubar*)
(#<Object #15, "", a *menu*>
 #<Object #18, "File", a *menu*>
 #<Object #29, "Edit", a *menu*>
 #<Object #42, "Eval", a *menu*>
 #<Object #48, "Tools", a *menu*>
 #<Object #60, "Windows", a *menu*>)
```

find-menu *string* [*Function*]
returns the first installed menu that has *string* as its title.

Example:

```
? (find-menu "File")
#<Object #18, "File", a *menu*>
```

Menus

menus [*Function*]
returns a list of all existing menu objects, including those which are not currently installed.

menu [*Variable*]
the menu class, used for creating new menus.

exist *init-list* [*Menu Function*]
initializes the menu so that menu-items can be added to it and so that it can be installed. `exist` does not add the menu to the menu-bar. This must be done by asking the menu to `menu-install`.

keyword	type	default	meaning
`:menu-title`	*string*	`"Untitled"`	the title of the menu.
`:menu-items`	*list*	`()`	list of items to be added to the newly created menu.

Example:
```
? (setq foo-menu (oneof *menu* :menu-title "Foo"))
#<Object #187, "Foo", a *menu*>
? (ask foo-menu (menu-title))
"Foo"
? (ask foo-menu (menu-installed-p))  ;Hasn't yet been installed
NIL                                  ;  in the menubar.
```

menu-install [*Menu Function*]
menu-deinstall [*Menu Function*]
menu-installed-p [*Menu Function*]
`menu-install` adds the menu to the menubar, at the rightmost position. From this point (until it is deinstalled), it will be useable.
`menu-deinstall` removes a menu from the menubar. It still exists, so it retains its state and may be re-installed.
`menu-installed-p` returns non-nil if the menu is installed, nil if the menu is not installed.

menu-dispose [*Menu Function*]
retires the menu and reclaims its storage. If it is installed, it will first be asked to `deinstall`. `menu-dispose` returns `t`. Once a menu is `disposed` of, its state is lost. To do anything more with it, you must first ask it to `exist` again.

menu-title [*Menu Function*]
set-menu-title *new-title* [*Menu Function*]
`menu-title` returns the menu's title as a string.
`set-menu-title` sets the menu's title to *new-title*, which should be a string. If the menu is installed , the change in title will be immediately reflected in the menubar. *new-title* is returned.

Example:
```
? (ask foo-menu (menu-title))
"Foo"
? (ask foo-menu (set-menu-title "Bar"))
"Bar"
```

```
? (ask my-menu (menu-title))
"Bar"
```

menu-disable [*Menu Function*]
menu-enable [*Menu Function*]
menu-enabled-p [*Menu Function*]
`menu-disable` dims out a menu. Its items may still be viewed, but they cannot be selected. The menu and its items will appear in grayed out text. `menu-disable` has no effect if the menu was already disabled.
`menu-enable` undims a menu, making it possible to use its items (provided it is installed). `menu-enable` has no effect if the menu was already enabled.
`menu-enabled-p` returns non-nil if the menu is enabled, or `nil` if the menu is disabled.

add-menu-items `&rest` *menu-items* [*Menu Function*]
appends *menu-items* to the menu. They will be added to the bottom of the menu in the order they are specified. `nil` is returned.

Example:

```
? (ask foo-menu (add-menu-items
                  (oneof *menu-item*
                         :menu-item-title "Beep"
                         :menu-item-action '(ed-beep))
                  (oneof *menu-item*
                         :menu-item-title "Say Hello"
                         :menu-item-action '(print "Hello"))))
NIL
```

remove-menu-items `&rest` *menu-items* [*Menu Function*]
removes *menu-items* from the menu. The *menu-items* may later be re-installed, or may be installed in other menus. `nil` is returned. It is not an error to attempt to remove an item that is not in the menu.
menu-items `&optional (`*menu-item-class* `*menu-item*)` [*Menu Function*]
`menu-items` returns a list of the menu-items currently installed in the menu. Only those menu-items which inherit from *menu-item-class* are included in the list that is returned. The menu-items are in the same order in which they appear in the menu.

Example:

```
? (ask foo-menu (menu-items))
(#<Object #190, "Beep", a *menu-item*>
 #<Object #191, "Say Hello", a *menu-item*>)
```

find-menu-item *string* [*Menu Function*]
Returns the first menu-item in the menu whose title is *string*. If no menu-items in the menu have *string* for a title, `nil` is returned.

menu-update [*Menu Function*]
This procedure is called whenever the user clicks in the menubar. The usual `menu-update` simply asks all the of menu's menu-items to `menu-item-update`. This facility is provided so that menus and menu-items can adjust to the current program context before they are displayed (for example, by checking, unchecking, enabling or disabling, or adding or removing items).

`menu-update` may be shadowed, but should not normally be called by the user. (It is called by the Lisp run-time system).

Menu-items

`*menu-item*` [*Variable*]
the menu-item class. This is used for creating menu-items.

`exist` *init-list* [*Menu-item Function*]
initializes the menu-item so that it may be installed in a menu.

keyword	type	default	meaning
`:menu-item-title`	*string*	`"Untitled"`	the title of the menu-item.
`:command-key`	*char/nil*	`nil`	if `nil`, then the menu-item will have no command-key equivalent. If a character, then that character will be the menu-item's command-key equivalent.
`:menu-item-action`	*thing*	`()`	used to define `menu-item-action`, i.e. the action run when the menu-item is selected. *thing* may be a function, a symbol with a function binding, or a form. If it is a form, it will be used as the body of a defobfun.
`:disabled`	*boolean*	`nil`	if non-`nil`, the menu-item will be disabled.

Example:

```
? (setq Quux (oneof *menu-item*
                    :menu-item-title "Quux"
                    :menu-item-action #'(lambda ()
                                          (ed-beep)
                                          (print "I beeped!"))))
#<Object #203, "Quux", a *menu-item*>
?
```

`menu-item-action` [*Menu-item Function*]
this function is called whenever the user selects the menu-item with the mouse or through the command-key equivalent. The usual version does nothing. This function should generally be shadowed by each menu-item. It is responsible for actually doing whatever the menu-item is supposed to do.

`menu-item-title` [*Menu-item Function*]
`set-menu-item-title` *new-title* [*Menu-item Function*]
`menu-item-title` returns the menu-item's title (a string).
`set-menu-item-title` sets the menu-item's title to *string* and returns *string*.
If a menu-item's title is `"-"`, then the menu-item will be an unselectable dotted line. Such items are useful for separating sets of items in a menu.

Example:

```
? (ask Quux (menu-item-title))
"Quux"
? (ask Quux (set-menu-item-title "Baz"))
"Baz"
? (ask foo (menu-item-title))
"Baz"
```

command-key [*Menu-item Function*]
set-command-key *char* [*Menu-item Function*]
`command-key` returns the menu-item's keyboard equivalent. If the menu-item does not have a command-key equivalent, `nil` is returned.
`set-command-key` sets the menu-item's keyboard equivalent to *char*, which should be a character.

menu-item-disable [*Menu-item Function*]
menu-item-enable [*Menu-item Function*]
menu-item-enabled-p [*Menu-item Function*]
`menu-item-disable` disables a menu-item. It will be grayed-out, and the user will not be able to select it (with the mouse or its command-key equivalent). `menu-item-disable` has no effect if the menu-item is already disabled.
`menu-item-enable` enables a menu-item. The user will be able to select it.
`menu-item-enable` has no effect if the menu-item is already enabled.
`menu-item-enabled-p` returns non-`nil` if the menu-item is enabled, or `nil` if the menu-item is disabled.

menu-item-check-mark [*Menu-item Function*]
set-menu-item-check-mark *new-mark* [*Menu-item Function*]
`menu-item-check-mark` returns the character which is currently used to check the menu-item, or `nil` if the item is not currently checked.

`set-menu-item-check-mark` sets the check-mark of the menu-item. If *new-mark* is `nil`, then the item will be unchecked. If it is `t`, then the item will be checked with a standard check-mark symbol (√). If it is a character or ascii value, then the appropriate character will be used as a check-mark next to the menu-item. *new-mark* is returned. (Note, the check-mark character has the reader macro #\checkmark.)

(See example below).

menu-item-style [*Menu-item Function*]
set-menu-item-style *new-styles* [*Menu-item Function*]
`menu-item-style` returns a single keyword or list of keywords representing the menu-item's style.

new-styles should be a keyword or list of keywords. Allowable keywords are `:plain`, `:bold`, `:italic`, `:shadow`, `:outline`, `:underline`, `:condense`, and `:extend`. `:plain` indicates the absence of other styles.

menu-item-update [*Menu-item Function*]
`menu-item-update` is called by `menu-update` whenever the user clicks in the menubar. The pre-defined `menu-item-update` does nothing, but may be shadowed to let menu-items adjust

to the current program context. (for example, by checking, unchecking, disabling, or enabling themselves.) It is not normally called by the user. The value returned by `menu-item-update` is not normally used.

Example:

```
(defobfun (menu-item-update Quux) ()
          (if (= *time-zone* 5)
              (set-menu-item-check-mark t)
              (set-menu-item-check-mark nil)))
```

`Quux` will be checked if Allegro CL is running in the same time zone as Cambridge, Massachusetts.

Windows 6

Overview
Window Functions and Variables
Supporting Undo
Supporting Save and Save As...

Windows

Overview

Windows are the primary method for doing screen-related I/O. Information is displayed in windows, and the user edits, alters, and enters information by mousing windows, typing, etc. Events (such as keystrokes) are generally handled by the top window (see the Events chapter for details). The most commonly used windows are subclasses of `*window*`, Fred windows and dialog windows, which add important behavior. This chapter describes only the primitive superclass from which other windows inherit. All the commands described in this chapter can, however, be applied to the subclasses.

For information on drawing into windows, see the Quickdraw appendix.

For information on the size, resolution, and other physical characteristics of the display, see the System Parameters appendix.

Window Functions and Variables

`*window*` [*Variable*]
the window class. Windows inherit from `*stream*`.

`windows` `&optional` (*class* `*window*`) (*only-visible-p* `t`) [*Function*]
returns a list of existing windows which inherit from *class*. They are listed in order from front to back. If *only-visible-p* is `nil`, then invisible windows are included in the list; otherwise only visible windows are included.

`front-window` `&optional` (*window-class* `*window*`) (*only-visible-p* `t`) [*Function*]
returns the front-most window satisfying the arguments. The window must inherit from *window-class*, and if *only-visible-p* is `t`, then only visible windows qualify. If no windows satisfy the tests, `nil` is returned.

`find-window` *string* [*Function*]
returns the front-most window whose title is *string*. If no window has *string* as its title, `nil` is returned. (Note: the cross which appears in the title bar of modified Fred windows is ignored).

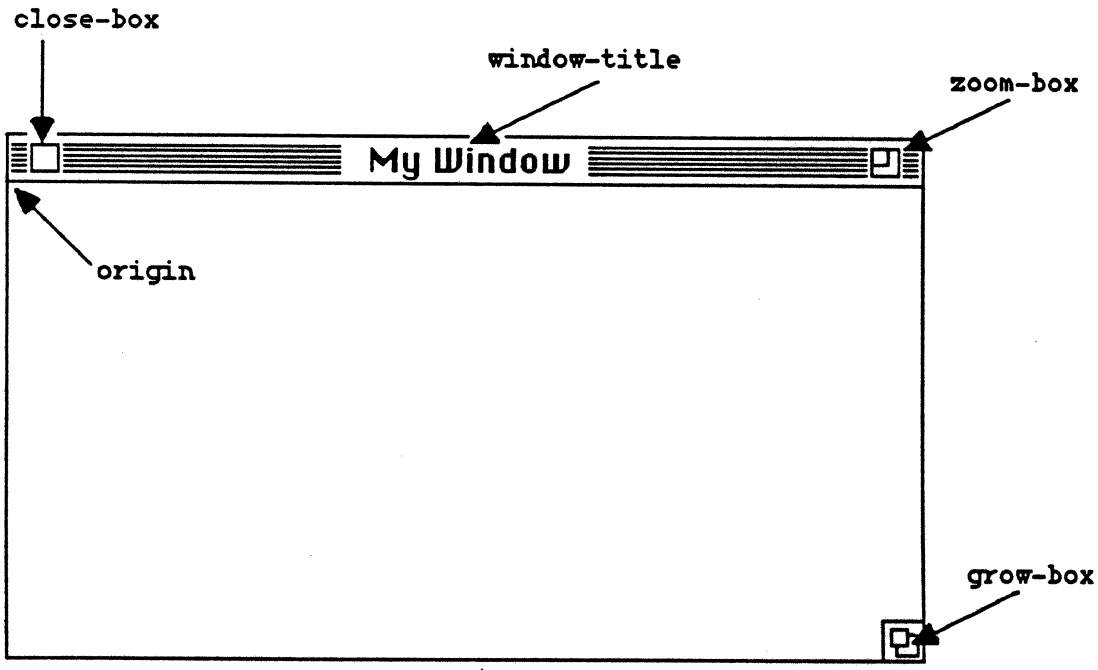
`exist` *init-list* [*Window Function*]
initializes a window instance.

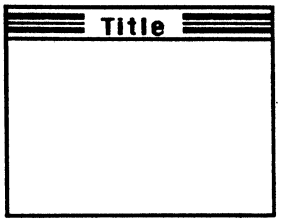
keyword	type	default	meaning
`:window-title`	*string*	`"Untitled"`	initial title of window is *string*. Only some types of windows display their titles.
`:window-position`	*point*	`#@(6 44)`	initial position of window is *point*.
`:window-size`	*point*	`#@(502 150)`	initial size of the window is x and y dimensions of *point*.
`:window-show`	*boolean*	`t`	determines whether the window is visible when created.
`:window-font`	*font-spec*	`("monaco" 9)`	the font-spec used by the window.

`:window-layer` ..... *integer* ..... 0 ..... determines the plane in which the new window will be created (see `set-window-layer` for details).

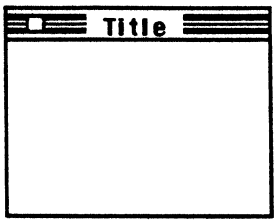
`:window-type` ..... *type-key* .... `:document-with-zoom` sets the window type according to *type-key*. *type-key* should be one of `:document`, `:document-with-grow`, `:document-with-zoom`, `:double-edge-box`, `:single-edge-box`, `:shadow-edge-box`, or `:tool` (see diagram below). The window type cannot be modified once the window is created.

`:close-box-p` ..... *boolean* ..... `t` ..... determines whether the window has a close box. This feature cannot be modified once the window is created. It is only available on `:document`, `:document-with-grow`, `:document-with-zoom`, and `:tool` windows.

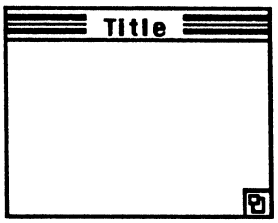




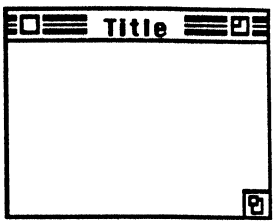
:document



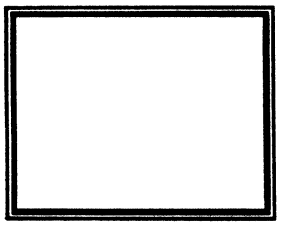
:document with close-box



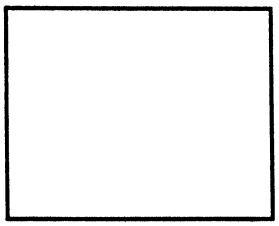
:document-with-grow without close-box



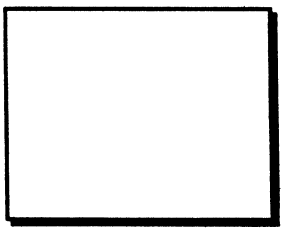
:document-with-zoom with close-box



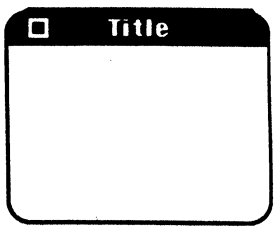
:double-edge-box



:single-edge-box



:shadow-edge-box



:tool with close-box

Example

```
? (setq baz (oneof *window*    :window-title "Bazwin"
                               :window-position #@(200 300)
                               :window-type :tool))
#<Object #206, "Bazwin", a *window*>
```

window-close [*Window Function*]

removes the window from the screen and reclaims the memory it uses. This operation is the inverse of `exist`. The window may be asked to `exist` again but its state will be lost. If the window is active when it is closed, `window-deactivate` is called first. It is an error to call `window-close` a second time, without first calling `exist`. `window-close` is called by the Allegro CL event system when the user clicks in a window's close-box.

You can tell if a window has been closed (or has not yet been asked to `exist`) by seeing if the window has a binding for the variable `wptr`.

Example:

```
? (ask baz (window-title))
"Bazwin"
? (ask baz (ownp 'wptr))
t                              ;the window has a binding of wptr
? (ask baz (window-close))
NIL                            ;the window disappears from screen
? (ask baz (ownp 'wptr))
NIL                            ;now closed, so no binding of wptr
? (ask baz (exist))
NIL                            ;the window reappears,
? (ask baz (window-title))
"Generic Window"               ;but with default state
```

window-position [*Window Function*]
set-window-position *h* `&optional (v nil)` [*Window Function*]
`window-position` returns the position of the window's origin (i.e. the upper-left corner of the window's content region), in global screen coordinates. The position is returned as a point.

`set-window-position` moves the window. If both *h* and *v* are give, they should be the new horizontal and vertical coordinates of the window. If *v* is `nil`, *h* is taken to be an encoded point holding both dimensions. The new position is returned as a point.

Example:

```
? (setq bim (oneof *window* :window-position #@(50 50)))
#<Object #208, "Untitled", a *window*>
? (ask bim (point-string (window-position)))
#@(50 50)
```

window-size [*Window Function*]
set-window-size *h* `&optional (v nil)` [*Window Function*]
`window-size` returns the window's size as a point.
`set-window-size` sets the size of the window. The upper-left corner of the window will be anchored, and the lower-right corner will move 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 *v* is `nil`, *h* is taken to be an encoded point holding both dimensions. The new size is returned as a point.

window-zoom-position [*Window Variable*]
window-zoom-size [*Window Variable*]
These variables determine the size and position of a window when it is zoomed out. Initially, windows use inherited bindings for these values. The inherited bindings may be changed, or particular classes or instances may be given their own bindings. `window-zoom-position` defaults to the upper-left corner of the screen, and `window-zoom-size` defaults to the full size of the screen.

window-title [*Window Function*]
set-window-title *new-title* [*Window Function*]
`window-title` returns the window's title as a string.
`set-window-title` sets the window's title to *new-title*.
Both functions ignore the crosses in the title-bars of modified Fred windows.

window-font [*Window Function*]
set-window-font *font-spec* [*Window Function*]
`window-font` returns the font-spec used for drawing text into the window.
`set-window-font` sets the window's font-spec to *font-spec*.

Example
```
? (ask bim (set-window-font '("helvetica" 10)))
#<A MAC Pointer 10371E>
```

window-show [*Window Function*]
window-hide [*Window Function*]
window-shown-p [*Window Function*]
`window-show` makes a window visible on the screen (assuming it is not at an off-screen position). `window-hide` makes the window invisible. `window-shown-p` returns nil if the window is currently hidden, and otherwise returns non-`nil`.

window-layer `&optional (`*class* `*window*)` [*Window Function*]
`(`*only-visible-p* `t)`
set-window-layer *number* `&optional (`*class* `*window*)` [*Window Function*]
`(`*only-visible-p* `t)`
`window-layer` returns the number of windows in front of the current window. Only windows of the specified *class* are counted. If *only-visible-p* is non-`nil`, then only visible windows are counted.
`set-window-layer` changes a window's layer to *number*, which should be a non-negative integer. *number* indicates how many windows should be in front of the window. If *number* is greater than the number of windows, the window will be moved all the way to the back. *class* and *only-visible-p* have the same meaning as for `window-layer`.

window-select [*Window Function*]
makes the window the front window, draws its controls, and makes it the recipient of future non-window-specific events (See the Events chapter for details), and shows it if it is hidden. The window which was previously the front window is deactivated.

wptr [*Window Variable*]
holds the pointer to the window record on the Macintosh heap. It can be used for directly examining the window record or for passing a window-pointer to Macintosh traps.

You can test to see if a window has been `asked` to `exist`, and has not been asked to `window-close`, by seeing if it has a binding for `wptr`.

with-port *grafport* {*form*}* [*Macro*]
executes the *forms* with *grafport* as the current grafport. The *forms* usually include low-level toolbox traps. Upon exit, the previously current grafport is restored. *grafport* is usually the `wptr` of a window.
This macro is not needed when using the high-level Quickdraw calls described in the Quickdraw chapter. It is only used when calling Quickdraw traps directly for maximum efficiency.

Supporting Undo

Any window can support an undo feature in conjunction with the **Undo** menu-item. Fred windows have a special mechanism for supporting undo, which is described in the chapter Programming Fred. The following general mechanism can be used by any window.

undo [*Window Function*]
If a window has a function binding for the symbol `undo`—but doesn't have a function binding for `window-can-undo-p`—then when the window is on top, the **Undo** menu-item will be enabled. If the window also has a function binding for `window-can-undo-p`, then the **Undo** menu-item will only be enabled if `window-can-undo-p` returns non-`nil`.

The `undo` function will be called when the **Undo** menu-item is selected.

window-can-undo-p [*Window Function*]
If a window has an undo function, it may also have `window-can-undo-p` function. If it does, then the **Undo** menu-item will only be enabled when calls to `window-can-undo-p` return non-`nil`.

`window-can-undo-p` can also have the sneaky side-effect of changing the title of the **Undo** menu-item (this menu-item is bound to the global variable `*undo-menu-item*`). The title will automatically be changed back when another window is selected.

Supporting Save and Save As...

Any window can make use of the **Save** and **Save As...** menu-items on the file menu. To support **Save**, the window should have a definition for the function `window-save`. To support **Save As...**, the window should have a definition for the function `window-save-as`. In addition, a window may have a definition for the function `window-needs-saving-p`, which determines whether the **Save** menu-item is enabled when the window is on top.

window-save [*Window Function*]
called when the window is on top and the user selects the **Save** menu-item from the **File** menu. If a window does not have (or inherit) a definition for this function, the **Save** menu-item will be disabled when the window is on top.

window-save-as [*Window Function*]
called when the window is on top and the user selects the **Save As...** menu-item from the **File** menu. If a window does not have (or inherit) a definition for this function, the **Save As...** menu-item will be disabled when the window is on top.

window-needs-saving-p [*Window Function*]
if defined for a window that also has a definition for `window-save`, `window-needs-saving-p` is used to determine whether the **Save** menu-item from the **File** menu should be enabled. A window does not need to have a definition for this function; if it doesn't (but *does* have a definition for `window-save`) then the **Save** menu-item will always be enabled. Otherwise, it will only be enabled if `window-needs-saving-p` returns non-`nil`.

Dialogs 7

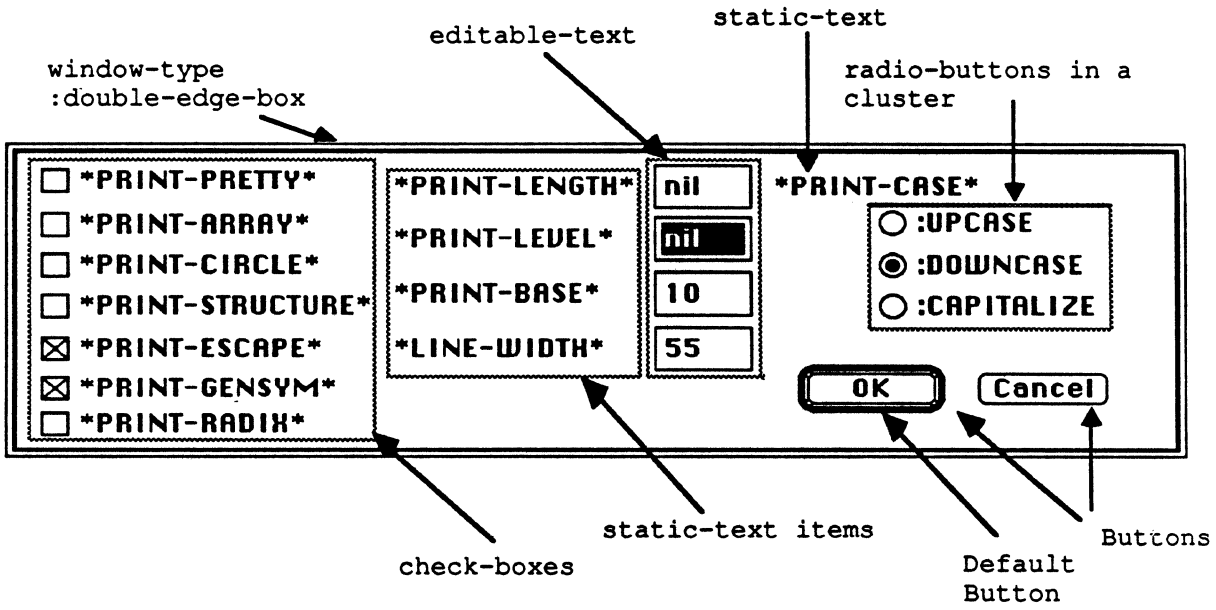
Overview
Dialog Functions
Dialog Items
Specialized Dialog-items
Table-dialog-items
Specialized Table-dialog-items

Dialogs

Overview

A *dialog* is a specialized kind of window that contains *dialog-items* (which perform actions when clicked by the user), text, and graphics. Dialog-items are implemented as objects; the available classes are buttons, check-boxes, radio-buttons, static text, editable text, and tables. Each dialog-item class inherits from the generic `*dialog-item*` class, so they share common capabilities. They all have a position, size, text and action. They are also either enabled or disabled. Disabled dialog-items usually appear grayed-out.

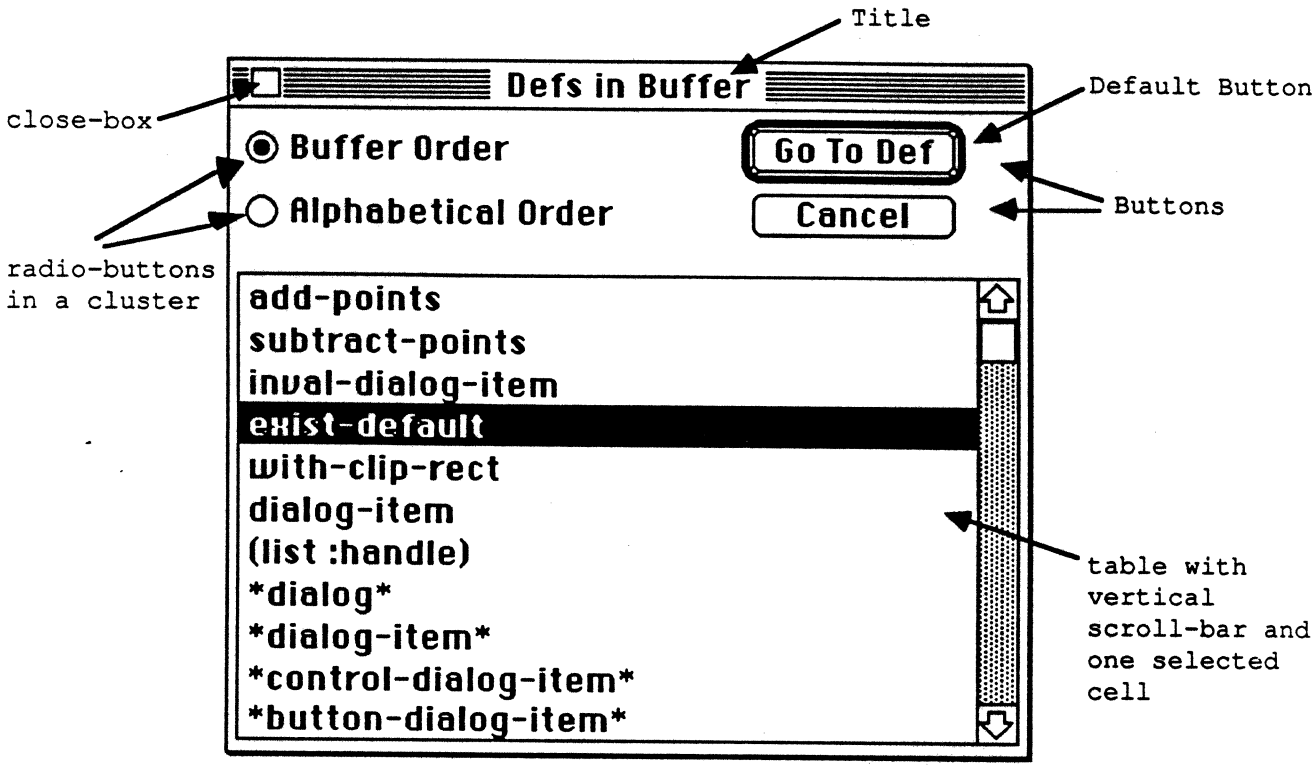
Because dialogs inherit from `*window*` they can perform all window operations. Some window operations (such as `window-show` and `set-window-position`) are needed to use dialogs effectively . See the chapter **Windows** for details of these operations.



A Modal Dialog Box

A dialog may be either *modal* or *modeless* . A modal dialog requires the user to exit the dialog before any other user actions can be performed A message box with an OK button is an example of a modal dialog. A modeless dialog allows active menus and the activation of other windows without explicitly terminating the dialog interaction. The "search" dialog is an example of a modeless dialog.

A dialog's modality is not inherent in the dialog object, but is determined by how the dialog is used. Any dialog object may be used as a modal or modeless dialog (although dialogs will normally be set up to work as one or the other). In order not to confuse users, a particular dialog should only be used in one way, as modal or modeless. Modal dialogs are usually created from window-type `:double-edge-box` (and are therefore not movable and have no close-box, size-box, or title). Modeless dialogs are usually created from window-type `:document` (and are therefore movable, have a title, and may have a close-box). Modal dialogs should only be made from `:document` windows when they must be movable.



A Modeless Dialog Box

A modal dialog is activated by calling the function `modal-dialog` with a dialog object as the argument. The dialog will be shown (an error will be signalled if the dialog is already shown), and made the front window. All subsequent user events will be processed by the dialog, illegal events will produce a beep and legal events will trigger the action of the appropriate dialog-item. The dialog will continue to intercept all events until `return-from-modal-dialog` is called. This causes the dialog to be hidden and supplies the value to be returned from the call to `modal-dialog`. Modal dialogs may be nested. An abort (command-period) always gets you out of a modal dialog.

Modeless dialogs are available for use if they are shown. Like any window, if a modeless dialog is clicked when it is not the front window, it comes to the front and becomes the active window. If a modeless dialog is the active window, then appropriate user events trigger the actions of its dialog-items.

Unless otherwise specified, all the text in a dialog (i.e. the text of all the items) appears in the window's current font. The desired font should be set before the dialog is created (using `set-window-font` or the `window-font` init option). Individual items may be given their own font, which will be used instead of the window's font. Changing the font while a dialog is shown may produce some undesirable cosmetic effects.

`*dialog*` [*Variable*]
Contains the dialog class object.

exist *init-list* [*Dialog Function*]
Initializes the dialog. The dialog will take up space on the Macintosh heap until it is closed. A window is closed by clicking in the close-box or explicitly asking it to `window-close`.

keyword	type	default	meaning
`:window-title`	*string*	`"Untitled Dialog"`	the window title of the dialog.
`:window-position`	*point*	`#@(200 50)`	initial position
`:window-size`	*point*	`#@(200 100)`	size of dialog
`:window-show`	*boolean*	`t`	whether dialog window is initially shown
`:window-font`	*font-spec*	`("chicago" 12)`	font to be used in the dialog.
`:window-type`	*keyword*	`:document`	type of window used. For available window-types, see the **Windows** chapter.
`:dialog-items`	*list*	`nil`	a list of dialog-items to place in the dialog
`:default-button`	*thing*	`t`	may be `t`, `nil`, or a button. If `nil`, the dialog will have no default button. If t, the first button in the dialog will be the default button. If a button, it will be made the default button.

modal-dialog *dialog-object* `&optional` (*close-on-return* `t`) [*Function*]
makes *dialog-object* the front window, shows it, then intercepts subsequent user events until a `return-from-modal-dialog` is executed. User events are directed to the appropriate dialog-items triggering their actions; events outside the frame of the dialog window cause a beep. The return value is supplied by the call to `return-from-modal-dialog` that exits the modal dialog. An error is signalled if *dialog-object* does not inherit from `*dialog*`. Modal dialogs may be nested.
The optional argument *close-on-return* specifies whether the dialog window should be closed just before the call to `modal-dialog` returns. Having the windows closed automatically prevents the accumulation of numerous hidden windows during development. If *close-on-return* is `nil`, then the window is hidden, but not closed. (Note: *close-on-return* is unwind-protected, so that any throw past `modal-dialog` will close or hide the window, as appropriate).

return-from-modal-dialog *values* [*Macro*]
Causes *values* to be returned from the most recent call to `modal-dialog`. The dialog is hidden or closed, depending on the value of *close-on-return* that was passed to the call to `modal-dialog`. (Any throw past the `modal-dialog` call also causes the dialog to be hidden or closed). If the dialog is only hidden, it will continue to take up memory until it is explicitly asked to `window-close`.

`return-from-modal-dialog` supports multiple-values. If the first (or only) value is the keyword `:cancel`, `return-from-modal-dialog` will throw to `:cancel`. Unless caught by the user, this throw will return to the top level listener loop. It is customary for the cancel button to return the value `:cancel`.

If the modal dialog has a close box (they usually shouldn't), and the user clicks in it, `return-from-modal-dialog` will be called with the argument `:closed`. The window will be closed, regardless of the value of *close-on-return*.

window-close [*Dialog Function*]
Reclaims space used by the dialog. All of the items in the dialog are removed.

add-dialog-items `&rest` *dialog-items* [*Dialog Function*]
each of the *dialog-items* is added to the dialog. If any of the items are already owned by a dialog, an error is signalled. `nil` is returned.

Example:

```
(ask my-dialog
     (add-dialog-items (oneof *button-dialog-item*
                              :dialog-item-text "Do It"
                              :dialog-item-position #@(20 80)))
```

remove-dialog-items `&rest` *dialog-items* [*Dialog Function*]
Removes each of the *dialog-items* from the dialog. `nil` is returned.

dialog-items `&optional` *dialog-item-class* [*Dialog Function*]
returns a list of the dialog's dialog-items that inherit from *dialog-item-class*. *dialog-item-class* defaults to `*dialog-item*` so that all of the dialog's dialog-items will be returned. The items are returned in the reverse order that they were added.

find-dialog-item *string* [*Dialog Function*]
returns the first item in the dialog whose `dialog-item-text` is `equalp` to *string*. The items are searched in the order they were added to the dialog (which is the reverse of the order that they are returned by `dialog-items`).

current-editable-text [*Dialog Function*]
set-current-editable-text *dialog-item* [*Dialog Function*]
The current-editable-text-item is the editable text item that contains the blinking insertion point. This is the item where user typing appears.
`current-editable-text` returns the dialog's current editable text dialog-item. If the dialog contains no enabled editable text items, `nil` is returned.
`set-current-editable-text` changes the dialog's current editable text item to be *dialog-item*. *dialog-item* must inherit from `*editable-text-dialog-item*` and be in the dialog's list of dialog-items or an error will be signalled. *dialog-item* is returned.

default-button [*Dialog Function*]
set-default-button *thing* [*Dialog Function*]
The default button is the button whose action is run when the user types `#\return` or `#\enter` while a dialog is active. The default button is automatically outlined with a black border to indicate to the user which button it is. If the current editable text allows returns, return characters will go into the editable text instead of running the default button's action. If this is true, the black border around the default button will be removed to indicate this condition to the user (but calls to `default-button` will still return the button).

`default-button` returns the current default button. If the dialog has no default button, `nil` is returned.

`set-default-button` changes the default button in the dialog according to the value of *thing*, which should be `nil`, `t`, or a button-dialog-item. If *thing* is `nil`, there will be no default button. If it is `t`, the first button in the dialog will be made the default button. If it is a button, the button will be made the default button. *thing* is returned.

`pushed-radio-button` `&optional (`*cluster* `0)` [*Dialog Function*]
returns the pushed radio-button from the specified cluster. `nil` is returned if there is no such cluster, or no pushed radio-button in the cluster (the latter should only occur when the radio-buttons in a cluster are disabled).

`cut` [*Dialog Function*]
Deletes selected text from the current editable text dialog-item and puts it into the Clipboard.

`copy` [*Dialog Function*]
Copies selected text from the current editable text dialog-item into the Clipboard.

`paste` [*Dialog Function*]
Replaces the selected text of the current editable text dialog-item with the contents of the Clipboard.

Dialog-items

`*dialog-item*` is an abstract class (it is not meant to be directly instantiated). It is the parent of more specific classes of dialog-item class.

The dialog-item subclasses are:

```
*button-dialog-item*
*check-box-dialog-item*
*radio-button-dialog-item*
*static-text-dialog-item*
*editable-text-dialog-item*
*table-dialog-item* (an abstract class)
*sequence-dialog-item*
*array-dialog-item*
```

`*dialog-item*` [*Variable*]
The dialog-item class. `*dialog-item*` provides the basic functionality that all dialog-items share.

`exist` *init-list* [*Dialog-Item Function*]
Initializes a dialog-item. If `dialog-item-size` is not specified in *init-list* it is determined heuristically from the dialog-item's class and other *init-list* data. For example, a button will be made just big enough to contain its text. If `dialog-item-position` is not specified, the dialog-item is positioned in the first vacant space when it is added to the dialog window; if there is no rectangle large enough to hold the item, an error is signalled.

keyword	type	default	meaning
`:dialog-item-position`	*point*	calculated	the position in the dialog where the item will be placed, in local window coordinates. If not

specified, the first available position large enough to hold the item will be used.

`:dialog-item-size` ..... *point* ..... calculated ..... the size of the dialog-item. If not specified, this will be calculated to fit the item. If specified and too small, the item will be clipped.

`:dialog-item-font` ..... *font-spec* ... `nil` ..... the font used to display the dialog-item's text. If `nil`, the dialog's window-font is used.

`:dialog-item-enabled-p` .... *boolean* ..... `t` ..... whether the item is enabled. disabled items are displayed grayed-out, and their actions are not run when you click on them.

`:dialog-item-text` ..... *string* ..... `t` ..... the text of the dialog-item.

`:dialog-item-action` ..... *expression* . `nil` ..... used to define `dialog-item-action`, i.e. the action run when the dialog-item is selected. *thing* may be a function, a symbol with a function binding, or a form. If it is a form, it will be used as the body of a defobfun.

`dialog-item-action` [*Dialog-Item Function*]
This function is called with no arguments whenever the user clicks in the dialog-item. Each individual instance of a dialog-item may have its own definition, but since each class of dialog-item has a usual behavior (for example check-boxes toggle their state), `usual-dialog-item-action` should normally be called. `dialog-item-action` will not be called if the item is disabled. It is normally called when the mouse button is released, not when it is pressed (this allows tracking to be performed by the system).

When a `dialog-item-action` is called, the program is interrupted and further events are disabled until the `dialog-item-action` returns. This means that other dialog-items cannot be selected until the first returns. To avoid locking out other actions, dialog-items can insert forms into the read/eval/print loop with `eval-enqueue`. For details, see the Events chapter.

If a `dialog-item-action` is fbound to an anonymous lambda expression, the action will only be able to call usual-action if the lambda was defined with `nfunction`. See the Object Lisp chapter for details.

`dialog-item-position` [*Dialog-Item Function*]
`set-dialog-item-position` *h* `&optional` (*v* `nil`) [*Dialog-Item Function*]
A dialog-item's position is the position of the item's upper-left corner in the dialog window's local coordinates.
`dialog-item-position` returns the dialog-item's position.
`set-dialog-item-position` moves the dialog-item to the position represented by *h* and *v* in the dialog window's local coordinates. If *v* is `nil`, *h* is assumed to represent a point. The new position is returned as a point.

dialog-item-size [*Dialog-Item Function*]
set-dialog-item-size *h* `&optional (v nil)` [*Dialog-Item Function*]
When a dialog-item is drawn, the drawing is clipped to the rectangle determined by the dialog-item's position and size. This is also the rectangle that is used to determine which dialog-item receives user mouse-clicks.
`dialog-item-size` returns the dialog-item's size as a point.
`set-dialog-item-size` changes the size of the dialog-item to the width and height represented by *h* and *v*. If *v* is not given, *h* is assumed to represent a point. The new size is returned.

dialog-item-text [*Dialog-Item Function*]
set-dialog-item-text *string* [*Dialog-Item Function*]
The text of a dialog-item has different meanings for each dialog-item class. It is the text of static-text and editable-text items. It is displayed inside buttons, and to the right of radio-buttons, and check-boxes. If a different location is desired, set the text to the empty string and use a separate static-text item. Tables do not display their dialog-item-text.
`set-dialog-item-text` returns the text associated with the dialog-item as a string.
`dialog-item-text` sets the text associated with the dialog-item to *string* and returns *string*.

dialog-item-font [*Dialog-Item Function*]
set-dialog-item-font *font-spec* [*Dialog-Item Function*]
`dialog-item-font` returns the font used by the item in the form of a font-spec, or `nil` if the dialog-item uses its window's font.
`set-dialog-item-font` sets the dialog-item's font to *font-spec*. If *font-spec* is `nil`, the dialog-item will use the font of its dialog window. The dialog-item will not resize, so calling this function after a dialog-item has been created may cause display problems.

dialog-item-draw [*Dialog-Item Function*]
This function is defined in each type of item. It is called by the system whenever the item needs to be drawn (for example, when the dialog is first displayed, or when it is uncovered after having been covered up). This function may be defined for any user-specialized items.

dialog-item-enable [*Dialog-Item Function*]
dialog-item-disable [*Dialog-Item Function*]
dialog-item-enabled-p [*Dialog-Item Function*]
`dialog-item-enable` enables the dialog-item. It will not be dimmed, and its action will be run when the user clicks in it. `nil` is returned.
`dialog-item-disable` disables the dialog-item. This causes the dialog-item to become dimmed; clicks in the item are ignored, and the action of the item is never run. Disabling a check-box does not uncheck it, and disabling a radio-button does not unpush it (you may want to uncheck or unpush the item explicitly). `nil` is returned.
`dialog-item-enabled-p` returns `t` if the dialog-item is enabled, `nil` if it is disabled.

my-dialog [*Dialog-Item Variable*]
the dialog object that owns the dialog-item. It is `nil` if the item is not installed in any dialog object. This variable should never be changed by the user. It is affected by the `*dialog*` functions: `add-dialog-items`, and `remove-dialog-items`.

add-self-to-dialog *dialog* [*Dialog-Item Function*]
this function is called by `add-dialog-items` when it is adding an item to a dialog. Its purpose is to perform those initialization tasks that require a window (e.g. defaulting the position or size). It should never be called directly by user code. However, it may be shadowed. *dialog* will hold the dialog that the item is being added to. Specialized versions of `add-self-to-dialog` should always call `usual-add-self-to-dialog`.

remove-self-from-dialog [*Dialog-Item Function*]
this function is called when a dialog-item is being removed from a dialog by a call to `remove-dialog-items`. It should never be called directly by user code. However, it may be shadowed. Specialized versions of `remove-self-from-dialog` should dispose of any Macintosh data (i.e. non-garbage-collectable data) used by the item, and should always call `usual-remove-self-to-dialog`.

Specialized Dialog-items

button-dialog-item [*Variable*]
the class used to make button dialog-items. Buttons are displayed as rounded rectangles that contain text. Clicking in a button usually has an immediate result (they should have their own definition of `dialog-item-action`). The first button added to a dialog automatically becomes the default button. This may be changed by asking the dialog to `set-default-button`. The default button is outlined with a black border. In general, typing return or enter is equivalent to clicking in the default button. However, if the current editable-text item allows returns, then returns and enters will not trigger the default button; in this case, the black border around the default button is removed.

static-text-dialog-item [*Variable*]
the class of static text dialog-items. Static text may be positioned anywhere in a dialog window to explain the layout of the dialog or supply additional information to the user. The text appears in the window's font, unless otherwise specified. Static text does not generally have an action, but it may. (Unlike other items, the action of a static text is run when button is pressed, not when it is released).

Depending on the amount of text, and the size of the item, the text may perform word-wrap to fit within its area. If the size is not given explicitly, it will be the size needed to hold the text without wrapping to multiple lines.

editable-text-dialog-item [*Variable*]
Contains the editable text dialog-item class. Editable text dialog-items are surrounded by a box. The text that appears in the box may be edited by the user in the standard Macintosh ways. It may be selected using the mouse, and works with cut, copy, and paste (though the command-keys for cut, copy and paste do not work from within modal dialogs).

At any given time, there is only one current editable text dialog-item. This is the one that contains a blinking cursor or a highlighted selection. User typing is directed to the current editable text dialog-item by asking it to `dialog-item-key-event-handler`. The tab key changes the current editable text to the next editable text in the dialog, cycling back to the first after the last. The current editable text item may determined by asking the dialog `current-editable-text` and changed by asking the dialog to `set-current-editable-text`.

The text may be determined by calling `dialog-item-text` or changed by calling `set-dialog-item-text`. The initial text may be specified when an editable text dialog-item is created by using the `dialog-item-text` init option.

`exist` *init-list* [*Editable-Text-Dialog-Item Function*]
Initializes the editable text dialog-item.

keyword	type	default	meaning
`:allow-returns`	*boolean*	`nil`	if `allow-returns` is non-nil, then carriage returns may be entered into the text of the editable text item. The default button (if there is one) will be disabled.

`allow-returns` [*Editable-Text-Dialog-Item Variable*]
this variable will be non-`nil` if the editable-text item allows returns, otherwise it will be `nil`. This variable should never be changed by the user.

`*check-box-dialog-item*` [*Variable*]
Contains the check-box class. Check-boxes are small squares that contain an x when they are checked. The usual dialog-item-action toggles its state between being checked and unchecked. The text of a check-box appears to the right of the box.

`exist` *init-list* [*Check-Box-Dialog-Item Function*]
Initializes the item.

keyword	type	default	meaning
`:check-box-checked-p`	*boolean*	`nil`	whether the item is initially checked.

`check-box-check` [*Check-Box-Dialog-Item Function*]
`check-box-uncheck` [*Check-Box-Dialog-Item Function*]
`check-box-checked-p` [*Check-Box-Dialog-Item Function*]
`check-box-check` checks the dialog-item's check-box. The dialog-item's action is not run.
`check-box-uncheck` unchecks the dialog-item's check-box. The dialog-item's action is not run.
`check-box-checked-p` returns `t` if the dialog-item's check-box is checked, `nil` if it is unchecked.

`*radio-button-dialog-item*` [*Variable*]
Contains the radio-button dialog-item class. Radio-buttons are small open circles that contain a black dot when pushed. Radio-buttons occur in clusters in which only one button is pushed at a time. They are like the buttons on a car radio: pushing one button causes the previously pushed button to unpush.

exist [*Radio-Button-Dialog-Item Function*]
Initializes the radio-button instance.

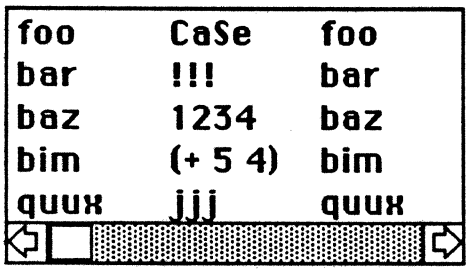
keyword	type	default	meaning
`:radio-button-cluster`	*sym*	0	the cluster to which the radio-button belongs. To test if two buttons are in the same cluster, the values for this variable are compared using eq.
`:radio-button-pushed-p`	*boolean*	`nil`	determines whether the radio-button is initially pushed.

radio-button-push [*Radio-Button-Dialog-Item Function*]
radio-button-unpush [*Radio-Button-Dialog-Item Function*]
radio-button-pushed-p [*Radio-Button-Dialog-Item Function*]
`radio-button-push` pushes the radio-button and unpushes the previously pushed button in the cluster. The dialog-item's action is not run.
`radio-button-unpush` unpushes the radio-button.
`radio-button-pushed-p` returns `t` if the radio-button is pushed, `nil` if it is not pushed.

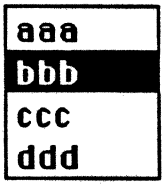
radio-button-cluster [*Radio-Button-Dialog-Item Variable*]
this variable holds the radio-button's cluster ID. Only one button from a given cluster is pushed at a time. Whenever you push one button, all the other buttons with the same value for `radio-button-cluster` (testing with `eq`) are asked to `radio-button-unpush`.

Table-dialog-items

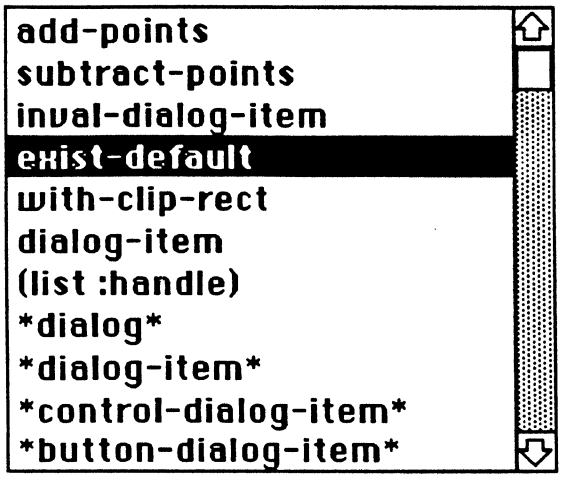
Table-dialog-items provide a method for viewing a set of items, and selecting items from the set. The items are usually members of a list, vector, or multi-dimensional array, but this is not required. table-dialog-items may be one or two dimensional. Two dimensional table-dialog-items look like spreadsheets. One dimensional table-dialog-items look like the file selection boxes when you choose **Save As....** (Note: table-dialog-items are implemented using the Macintosh List Manager. We call them tables to avoid confusion with the ubiquitous "list" of Lisp.)



2 dimensional table-dialog-item with horizontal scroll-bar.



1 dimensional table-dialog-item arranged vertically, with no scroll-bar.



1 dimensional table-dialog-item arranged vertically, with vertical scroll-bar.

1 dimensional table-dialog-item arranged horizontally, with horizontal scroll-bar

Everything that works for normal dialog-items (such as dialog-item-size, dialog-item-position, etc) works for tables, except that the text of tables is not shown.

Table-dialog-items are rectangles with a series of cells. In each given dimension (vertically and horizontally), if all the cells cannot be fit in the item's rectangle, then scroll bars will appear. When you click on a cell, it is selected, and the previously selected cell is de-selected. Selected cells usually appear in inverse video (though you can shadow the highlighting function). A contiguous group of cells can be selected by holding down the shift key and dragging or clicking. A discontiguous group of cells can be selected by holding down the command key and clicking on each desired cell. Clicking and dragging outside the table's rectangle causes auto-scrolling. Your program can access various information about a table, such as which cells are selected, the quickdraw positions of any cell, the contents of any cell, etc.

The cells in table-dialog-items are described as points. The horizontal and vertical dimensions of the point are encoded in a fixnum, with the same technique used for Quickdraw points. The horizontal and vertical components can be retrieved from the point with the functions `point-h` and `point-v`. The point can be printed readably with the function `point-string`. The following diagrams show one and two dimensional tables, and the points associated with their cells.

#@(0 0)	#@(1 0)	#@(2 0)	#@(3 0)
#@(0 1)	#@(1 1)	#@(2 1)	#@(3 1)
#@(0 2)	#@(1 2)	#@(2 2)	#@(3 2)
#@(0 3)	#@(1 3)	#@(2 3)	#@(3 3)
#@(0 4)	#@(1 4)	#@(2 4)	#@(3 4)
#@(0 5)	#@(1 5)	#@(2 5)	#@(3 5)
#@(0 6)	#@(1 6)	#@(2 6)	#@(3 6)
#@(0 7)	#@(1 7)	#@(2 7)	#@(3 7)
#@(0 8)	#@(1 8)	#@(2 8)	#@(3 8)
#@(0 9)	#@(1 9)	#@(2 9)	#@(3 9)

#@(0 0)
#@(0 1)
#@(0 2)
#@(0 3)
#@(0 4)
#@(0 5)
#@(0 6)
#@(0 7)
#@(0 8)
#@(0 9)

#@(0 0) #@(1 0) #@(2 0) #@(3 0) #@(4 0)

table-dialog-item [*Variable*]
an abstract class, from which usable classes of table-dialog-items inherit. It provides the base functionality for all types of table-dialog-items. This class should not be directly instantiated.

Exist *init-list* [*Table-Dialog-Item Function*]
initializes a new table-dialog-item.

keyword	type	default	meaning
`:table-vscrollp`	*boolean*	calculated	whether the table should have a vertical scroll-bar.
`:table-hscrollp`	*boolean*	calculated	whether the table should have a horizontal scroll-bar.
`:table-dimensions`	*point*	`#@(0 0)`	number of cells in horizontal and vertical dimension. A dimension given as zero will be calculated. The theoretical dimension limit is 32k by 32k. However, the Macintosh allocates 4 bytes per cell, so the practical limitation is much smaller than this, and depends on available memory.
`:visible-dimensions`	*point*	calculated	if given, will set the size of the table so that the number of cells shown horizontally and vertically correspond to the dimensions of *point*.
`:cell-size`	*point*	calculated	horizontal and vertical dimensions of the cells in the table.

`:selection-type` ..... *keyword* .. `:single` ..... determines whether the table allows single or multiple selections, and whether multiple selections must be contiguous. Possible keywords are `:single`, `:contiguous`, and `:disjoint`.

The standard keyword arguments for dialog-items (such as `dialog-item-position`, `dialog-item-size`) may also be used. Note that table-dialog-items do not display their text.

`cell-contents` *h* `&optional (`*v* `nil)` [*Table-Dialog-Item Function*]
This function is shadowed by specialized forms of table-dialog-items to return a printable representation of the contents of the cell specified by *h* and *v*. `cell-contents` is called by `draw-cell-contents`.

`draw-cell-contents` *cell* [*Table-Dialog-Item Function*]
draws the contents of *cell*. This function is called by CCL and should not usually be called by user code. It may be shadowed to provide specialized displays (for example, you could create a table of patterns, or icons). The usual version `princs` the cell's contents into the display; if the item is too long to fit in the cell, an ellipsis is added to specify that truncation has occurred. *cell* is a point.

Before calling this function, CCL sets the clip-rect to restrict drawing to the cell, erases the cell, and moves the pen to a position three-pixels from the bottom, and three pixels from the left edge of the cell.

`table-dimensions` [*Table-Dialog-Item Function*]
`set-table-dimensions` *h* `&optional (`*v* `nil)` [*Table-Dialog-Item Function*]
`table-dimensions` returns a point indicating the number of cells horizontally and vertically in the table.
`set-table-dimensions` sets the number of cells horizontally and vertically according to *h* and *v*. The theoretical limits on the number of cells are 32k by 32k. However, the Macintosh allocates 4 bytes per cell, so the practical limit will be much lower. The new dimensions are returned as a point.

`visible-dimensions` [*Table-Dialog-Item Function*]
`set-visible-dimensions` *h* `&optional (`*v* `nil)` [*Table-Dialog-Item Function*]
provides an alternate to `dialog-item-size` for specifying the size of a table.
`visible-dimensions` returns a point indicating the number of cells visible in the horizontal and vertical dimensions.
`set-visible-dimensions` 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.

`cell-size` [*Table-Dialog-Item Function*]
`set-cell-size` *h* `&optional (`*v* `nil)` [*Table-Dialog-Item Function*]
`cell-size` returns the cell-size of the table.
`set-cell-size` sets the cell size according to *h* and *v* and returns the new size as a point.

cell-select *h* &optional (*v* nil) [*Table-Dialog-Item Function*]
cell-deselect *h* &optional (*v* nil) [*Table-Dialog-Item Function*]
cell-selected-p *h* &optional (*v* nil) [*Table-Dialog-Item Function*]
`cell-select` selects the cell specified by *h* and *v*. Previously selected cells are not affected.
`cell-deselect` deselects the cell specified by *h* and *v*.
`cell-selected-p` returns `t` if the cell specified by *h* and *v* is selected, otherwise `nil`.

selected-cells [*Table-Dialog-Item Function*]
returns a list of all the cells selected in the table. Each cell is represented by a point. If no cells are selected, `nil` is returned.

scroll-to-cell *h* &optional (*v* nil) [*Table-Dialog-Item Function*]
scrolls the table so that the cell specified by *h* and *v* is in the upper-left corner.

scroll-position [*Table-Dialog-Item Function*]
returns the coordinates of the cell in the upper left corner of the table.

cell-position *cell* [*Table-Dialog-Item Function*]
if the cell is visible, returns the position of the *cell*'s upper left corner, otherwise returns `nil`. Positions are given in the dialog window's local window coordinates.

point-to-cell *point* [*Table-Dialog-Item Function*]
point should be given in the local coordinates of the dialog window. Returns the cell enclosing *point*, or `nil` if *point* is not within a cell.

Specialized Table-dialog-items

The following are specializations of table-dialog-items which may be used to create dialog-item instances. The first associates the table with a sequence. The second associates the table with an array.

sequence-dialog-item [*Variable*]
the sequence dialog-item class, used for displaying the elements of a sequence. Each instance is associated with a sequence. The elements of the sequence are displayed in the table, in a single row or column, or in multiple rows and columns. The table will only have multiple rows and columns if the length of the sequence is greater than the `:sequence-wrap-length`.

exist [*Sequence-Dialog-Item Function*]
initializes a sequence dialog-item.

keyword	type	default	meaning
`:table-sequence`	*sequence*	must be specified	the sequence to be associated with the table. This argument must be specified by the user.
`:sequence-order`	*keyword*	`:vertical`	whether the sequence will fill the table row by row, or column by column. Allowed keywords are `:vertical` and `:horizontal`.

`:sequence-wrap-length`.....*integer*.....`1073741823`.....The number of items to be allowed in a row or column before the tables wraps to the next row or column. This number overrides the `:table-dimensions` argument.

`table-sequence` [*Sequence-Dialog-Item Function*]
`set-table-sequence` *new-sequence* [*Sequence-Dialog-Item Function*]
`table-sequence` returns the sequence associated with the dialog-item. `set-table-sequence` sets the sequence associated with the dialog-item to *new-sequence*, resets the table's dimensions and scroll-bars, and redisplays the table.

`cell-to-index` *cell* [*Sequence-Dialog-Item Function*]
returns an index into the sequence associated with the table, corresponding to the element associated with *cell*.. This index may be used with `elt`.

`index-to-cell` *index* [*Sequence-Dialog-Item Function*]
returns a cell in the table which corresponds into the *index*'th element of the table's sequence.

Example
The following definitions create a new class which inherits from `*sequence-dialog-item*`. This class is useful for displaying association lists. Only the keys are displayed in the table. The keys, the values, or the complete association pair can be retrieved from the table.

```
;define the class
(defobject *alist-dialog-item* *sequence-dialog-item*)

;cell-contents is called by draw-cell-contents
;it returns only the car of the pair
(defobfun (cell-contents *alist-dialog-item*) (cell)
   (car (usual-cell-contents cell)))

;full-cell-contents returns the full pair
(defobfun (full-cell-contents *alist-dialog-item*) (cell)
  (elt (table-sequence) (cell-to-index cell)))

;value-cell-contents returns the value
(defobfun (value-cell-contents *alist-dialog-item*) (cell)
  (cdr (full-cell-contents cell)))
```

`*array-dialog-item*` [*Variable*]
The array-dialog-item-class. Each instance displays an associated array. The array may have any number of dimensions. At a given time, the index into all but two of the dimensions will be constant. The indexes into the non-constant dimensions are associated with the rows and columns of the table, and may be scrolled through. The non-constant dimensions are called the `h-specifier` and `v-specifier`. The indexes into the constant dimensions are taken from the `current-subscript` list.

exist [*Array-Dialog-Item Function*]

keyword	type	default	meaning
`:table-array`	*array*	must be specified	the array which will be associated with the table.
`:h-specifier`	*integer*	`0`	the array dimension to be associated with the horizontal axis of the table.
`:v-specifier`	*integer*	`1`	the array dimension to be associated with the vertical axis of the table.
`:table-subscript`	*list*	`'(0 0 0 ...)`	list of integers used for indexing into the array. The list's length should equal the array's rank.

h-specifier [*Array-Dialog-Item Function*]
set-h-specifier *new-h-specifier* [*Array-Dialog-Item Function*]
`h-specifier` holds the dimension of the array which is currently shown along the horizontal axis of the table.
`set-h-specifier` sets the table's h-specifier to *new-h-specifier*.

v-specifier [*Array-Dialog-Item Function*]
set-v-specifier *new-v-specifier* [*Array-Dialog-Item Function*]
`v-specifier` holds the dimension of the array which is currently shown along the vertical axis of the table.
`set-v-specifier` sets the table's v-specifier to *new-v-specifier*.

cell-to-subscript *cell* [*Array-Dialog-Item Function*]
given a cell (as a point), returns an index into the array (as a list of integers).

subscript-to-cell *list-of-integers* [*Array-Dialog-Item Function*]
given a list of subscripts, returns the corresponding cell or `nil` (if there is no corresponding cell).

table-array [*Array-Dialog-Item Function*]
set-table-array *new-array* [*Array-Dialog-Item Function*]
`table-array` returns the array associated with the dialog-item.
`set-table-array` sets the table's array to *new-array*, resets the table's dimensions and scroll-bars, and redisplays the table.

table-subscript [*Array-Dialog-Item Function*]
set-table-subscript *new-subscript* [*Array-Dialog-Item Function*]
`table-subscript` returns a list holding the subscripts currently used for indexing into the array.
`set-table-subscript` sets the subscripts used for indexing into the array. *new-subscript* should be a list (of integers) whose length equals the rank of the array. The h-specifier and v-specifier still determine which dimensions are displayed in the vertical and horizontal dimensions of the table. The indexes into the other dimensions are taken from the table-subscript.

Events 8

Overview
Event Handlers
Event Information Functions
The Event Management System
Cursor Handling

Events

Overview

This chapter describes Allegro CL's facilities for handling events, and for changing the appearance of the cursor.

Whenever possible, Macintosh programs should be *event driven.* Events are usually generated by the user as a way of directing program flow. Typical events are keystrokes and mouse clicks. Events interrupt a program and often require a response. This chapter explains how Allegro CL processes events and describes the language features for responding to events.

Allegro CL automatically handles events as a background task. When an event occurs, the current program is interrupted and the event is handled. Program execution is not resumed until the event handling function returns. Further event processing is also deferred until the event handling function returns. To initiate a program from within an event handler, use the function `eval-enqueue`.

Many user programs do not need to handle events explicitly. For those programs that do, there are several different event handling methods available. In order of increasing complexity these are:

- Defining a window object's response to specific types of events;
- Defining a window object's response to all events directed to the window;
- Defining a hook procedure that has first crack at processing all events;
- Disabling all background event-processing, and handling events with an event loop.

Most programming languages for the Macintosh support only the last (the most difficult) method of event handling. Programs in Allegro CL rarely need to do anything more complex than the first.

Event Handlers

The Allegro CL event system gets each event from the Macintosh operating system in turn and binds it to `*current-event*`. It then determines the type of the event and asks the appropriate window object to run an event-handling function for that event. The name of each window event-handler function starts with "`window-`" to indicate that it is a window object function and ends with "`-event-handler`" to indicate that it should only be called by the event system.

Many of the system's default event-handler functions do nothing, though they are called whenever an event of the appropriate type is processed. These handlers exist so that they may be shadowed by any window object that needs to process events of that type.

Event handler functions assume that a valid event record (see the chapter Pascal Records) is bound to `*current-event*`; they may call the current event information functions listed in the next section, which depend on `*current-event*` being bound.

`window-click-event-handler` *where* [*Window Function*]
is called by the event system whenever the user clicks in the content region of the active window. It is not called when the user clicks in an inactive window; in this case `window-activate-event-handler` is called. It is also not called when the user clicks in the title-bar, close-box, or other window control. *where* holds the mouse position of the click, in window coordinates. The

usual version does nothing (except, of course, in specialized windows provided by the system, such as Fred windows).

The following function will print the mouse coordinates whenever the user clicks in `my-window`.

```
(defobfun (window-click-event-handler my-window) (where)
    (print (point-string where)))
```

window-key-event-handler *char* [*Window Function*]
is called by the event system whenever the window is active and the user types a key. *char* is the character that was typed. This function is not called in response to command- and control-key events. Such events are handled by Fred (see the chapter Programming Fred for details).

window-null-event-handler [*Window Function*]
is called by the event system whenever it checks for an event and finds no events pending. `window-null-event-handler` should be defined for any window that needs to perform some periodic action, such as blinking the caret. The global version calls `update-cursor` (see the section Manipulating the Cursor, below). The Fred version blinks parentheses and other delimiters.

window-activate-event-handler [*Window Function*]
is called by the event system when a window is made the front window. It draws its controls, and deactivates the previously active window.

window-deactivate-event-handler [*Window Function*]
is called by the event system to deactivate a window. It is called when the window is active, and a different window is brought to the front.

window-update-event-handler [*Window Function*]
is called by the event system whenever any portion of the window needs to be redrawn. The usual version calls `_BeginUpdate` to up the `VisRgn` of the window to the portion that needs to be redrawn, draws the controls (if any), calls `window-draw-contents`, then calls `_EndUpdate` to restore the window's `VisRgn`. Since its behavior is universal for all windows, this function is seldom shadowed. Instead, `window-draw-contents` is shadowed.

window-draw-contents [*Window Function*]
is called whenever a window needs to redraw its contents. It may be shadowed so that a user-defined window can redraw when portions of it are covered and uncovered. When `window-draw-contents` is called by the event system, the window's `VisRgn` will be set so that drawing will only occur in the portions of the window that need to be redrawn. `window-draw-contents` is not strictly an event handler, since it may be called at any time (not only during event processing).

window-key-up-event-handler [*Window Function*]
Every key typed by the user actually generates two events. One event when the key is pressed down and another when the key is let up. This function is called whenever a key is let up. The usual version does nothing.

window-mouse-up-event-handler [*Window Function*]
is called whenever the user releases the mouse button. The usual version does nothing.

window-select-event-handler [*Window Function*]
is called whenever the user clicks the mouse in an inactive window. The usual version calls _SelectWindow. window-select-event-handler may be shadowed, for example, to make a window unselectable.

window-suspend-event-handler [*Window Function*]
is called by the event system whenever Lisp is suspended by the Switcher. The usual version converts the scrap from internal format to universal format (if necessary) and calls window-deactivate-event-handler.

window-resume-event-handler [*Window Function*]
is called by the event system whenever Lisp is resumed from the Switcher. The usual version converts the scrap to internal format and calls window-activate-event-handler.

window-disk-insert-event-handler [*Window Function*]
is called whenever a disk is inserted. The usual version mounts the disk; if the disk is unreadable, the usual version asks the user whether or not it should be initialized.

Event Information Functions

The following functions give event-related information. In addition, a program may examine the value of *current-event* during event-handling.

mouse-down-p [*Function*]
returns t if the mouse button is pushed, otherwise returns nil. This function may be called at any time, not only during event processing.

window-mouse-position [*Window Function*]
returns the mouse position as a point in the window's local coordinates. The point will be returned as a single fixnum (see the chapter Macintosh Basics). This function may be called at any time from within a window object, not only during event processing. The coordinates may be negative, or outside of the window's portrect, depending on the position of the mouse.

double-click-p [*Function*]
returns t if the mouse-click currently being processed was the second half of a double-click. double-click-p will return nil if called from outside event processing.

double-click-spacing-p *point1 point2* [*Function*]
This function is called by double-click-p when checking if two clicks should count as a double-click. *point1* and *point2* give the mouse positions of the two clicks. Macintosh guidelines specify that if the mouse is moved excessively between clicks, the clicks should not be counted as a double-click.

The usual version of double-click-spacing-p returns nil 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, t is returned.

This function may be shadowed by windows which filter double-click spacing with a different algorithm.

command-key-p [*Function*]
control-key-p [*Function*]
option-key-p [*Function*]
shift-key-p [*Function*]
caps-lock-key-p [*Function*]
Each of these functions has two meanings, depending on whether they are called during or outside of event processing.

If called during event processing, they will return `t` if the corresponding key was depressed during the event, otherwise `nil`.

If called outside of event processing, they will return `t` if the key is currently depressed, otherwise `nil`.

Note that only the most recent Macintosh keyboards have a control key.

The Event Management System

This section describes the system used for implementing event-handling in Allegro CL.

event-dispatch [*Function*]
is called periodically in the background. It calls `_GetNextEvent`, binds the value of `*current-event*` for the duration of the event processing, sets up the event-processing environment. It then calls `*eventhook*`, if it is bound. If `*eventhook*` returns `nil`, the event is then passed passed to the system event handlers. If `*eventhook*` returns non-`nil`, the processing of the event stops.

current-event [*Variable*]
holds the event record currently being processed. This is bound by `event-dispatch` and is only valid during event processing. The fields of `*current-event*` may be accessed with `rref` (see the chapter Pascal Records and *Inside Macintosh* for details).

The definition of the event record-type is:

```
(defrecord Event
  (what integer)
  (message longint)
  (when longint)
  (where point)
  (modifiers integer))
```

eventhook [*Variable*]
If `*eventhook*` is non-`nil`, it should contain a function to call in response to all events. After getting an event from the Macintosh operating system and binding it to `*current-event*`, the event system will funcall `*eventhook*` if it is non-`nil`. If the function returns `nil`, then normal event processing will continue, otherwise no further event processing will occur. This is the mechanism used to implement modal dialogs that beep on inappropriate events.

event-ticks [*Function*]
returns the number of ticks (60ths of a second) between calls to `event-dispatch`.

set-event-ticks *ticks* [*Function*]
sets the number of ticks between calls to `event-dispatch` to *ticks*. If *ticks* is too low, the system may get bogged down by event processing. If it is too high, the system may not respond smoothly to events.

window-event [*Window Function*]
After determining which window is the appropriate recipient for an event, the event system asks the window object to `window-event`. The usual `window-event` determines the type of the event and calls the appropriate event-handler. `window-event` should be shadowed by windows that need to do something in addition to or different from the default behavior.

eval-enqueue *form* [*Function*]
queues up *form* for evaluation in the read-eval-print loop. `eval-enqueue` returns immediately.

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*.

Example:

```
(setq my-menu (oneof *menu* :menu-title "Events"))

(ask my-menu (menu-install))

(setq inter-item
      (oneof *menu-item*
             :menu-item-title "Interruptible"
             :menu-item-action '(eval-enqueue
                                 '(loop
                                    (print "choose menus")))))
(setq non-inter-item
      (oneof *menu-item*
             :menu-item-title "Non Interruptible"
             :menu-item-action '(loop
                                 (print "choose if you can"))))

(ask my-menu (add-menu-items inter-item non-inter-item))
```

The first menu-item does not disable event handling. The second menu-item does. (Note: both can be aborted by typing command-period.)

The use of `eval-enqueue` can also be important for use in dialogs.

Example:

```
(setq my-dialog
      (oneof *dialog*
             :window-title "Stop and Go"
             :dialog-items
             `(,(oneof *button-dialog-item*
                       :dialog-item-text " Start "
                       :dialog-item-action
                       '(eval-enqueue
                         '(progn
                            (setq *stop-global* nil)
                            (loop
                              (if *stop-global*
                                  (return)
                                  (print
                                    "Click 'Stop' when bored"))))))
               ,(oneof *button-dialog-item*
                       :dialog-item-text " Stop "
                       :dialog-item-action
                       '(setq *stop-global* t)))))
```

Notice that the action of the Stop button does not use `eval-enqueue`. If it did, the queued up form would never be evaluated (because the form queued up by the Go button would never return). The stop button has to communicate with the Go button's action by side-effecting a variable.

`without-interrupts` `&body` *body* [*Special Form*]
executes *body* with all event processing disabled. This should be used sparingly since anything executing dynamically within a `without-interrupts` cannot be aborted or easily debugged. You are on your own if you execute a `break` within a `without-interrupts`.

Manipulating the Cursor

The *cursor* is the screen image corresponding to the mouse. As the mouse moves, the cursor moves on the screen. This is distinguished from the *caret* or *insertion point* that indicates the position in text where typed characters will appear.

The cursor often changes shape as it is moved over different areas of the screen. For example, when pointing at the menubar or into scroll-bars, it is shaped like an arrow; when inside a text window, the cursor is shaped like an I-beam. There are four ways a program can control the appearance of the cursor:

- a window may have a `window-cursor` variable. The event system will set the cursor according to this variable whenever the cursor is over the window.
- a window may have a `window-update-cursor` function. This function will be called by the event system whenever the cursor if over the with window.
- the `with-cursor` macro may surround a series of forms. The cursor will be set to a given shape for the duration of the macro.
- the variable `*cursorhook*` may be bound to a function or cursor-shape, giving you complete control over the shape of the cursor.

window-cursor [*Window Variable*]
Whenever the window is on top, and the cursor is over its content region, the cursor will be set to the cursor-shape contained in this variable. This is done by `*window*`'s `window-update-cursor` function. If this function is shadowed (as it is by Fred windows), then this variable may not have any effect.

window-update-cursor *where* [*Window Function*]
This function is called by `update-cursor` whenever the mouse is over the window (i.e. not only over the content region). *where* is the position of the cursor in the window's local coordinates. `*window*`'s version simply sets the cursor to the contents of the variable `window-cursor`. `*fred-window*`'s version sets the cursor to `*arrow-cursor*` if it is over a scroll bar or `*i-beam-cursor*` if it is not. `window-update-cursor` should be shadowed if a window needs to change the cursor to a different shape depending on what part of the window it is over.

with-cursor *cursor* `&body` *body* [*Macro*]
body is executed with the cursor set to *cursor*. *cursor* may be a cursor record or a CURS resource id (see *Inside Macintosh* for details).

cursorhook [*Variable*]
If this variable is non-`nil`, then no other cursor functions will be called. If `*cursorhook*` is a function, it will be called repeatedly in the background and have complete control over the state of the cursor at all times. If it is not a function then it should be a cursor. The cursor shape will be repeatedly set to its value using `set-cursor`.

update-cursor [*Function*]
does the actual work of cursor handling. It calls the `*cursorhook*` function if there is one, otherwise it calls `window-update-cursor`. It is called periodically by the global `window-null-event-handler`. It is not normally necessary to call this function directly, but it may be called to make sure that the cursor is correct at a particular time. The definition of update-cursor could be the following:

```
(defun update-cursor ()
  (if *cursorhook*
    (if (functionp *cursorhook*)
      (funcall *cursorhook*)
      (set-cursor *cursorhook*))
    (window-update-cursor)))
```

set-cursor *cursor* [*Function*]
Sets the shape of the mouse cursor to *cursor*. *cursor* may be a cursor record (either a pointer or a handle) or a CURS resource id. Note: if `set-cursor` is called from anywhere besides within a `window-update-cursor` function, a `*cursorhook*` function, or a `without-interrupts`, the cursor will be immediately set back to some other shape by the event system's background cursor handling. If *cursor* is not of an acceptable type, then no action is taken and no error is signalled.

arrow-cursor [*Variable*]
The standard north-northwest arrow cursor shape.

watch-cursor [*Variable*]
The watch cursor. This cursor should be shown during time-consuming operations when event-processing is disabled.

i-beam-cursor [*Variable*]
The I-beam cursor which is used when the cursor is over text to position the insertion point or make selections.

Programming Fred 9

Overview
Windows, Buffers, and Marks
Buffers
Marks
Fred Windows
Parameter Conventions
The Kill-Ring
Buffer Functions
Fred Window Functions
Fred Command Tables

Programming Fred

Overview

This chapter describes the functions and concepts needed to program Fred the editor.

Windows, Buffers, and Marks

Fred editing depends on two new data types and one new class: the two new data types are `buffer` and `mark`, and the new class is `*fred-window*`. Edited text appears on the screen in Fred windows. The actual text being edited is stored in a buffer. Locations or components of a buffer are associated with marks. In general, low-level operations are not object-oriented, and take a buffer as an argument. Higher-level operations are implemented as object functions for Fred windows.

Buffers

A buffer holds a sequence of characters, much like a string. However, the implementation of buffers makes insertion and deletion of characters much more efficient. In addition, a buffer has:

- a modification counter, which is incremented any time the buffer is modified;
- the set of marks in the buffer;
- a default-position mark, which provides the default position for many buffer operations such as insertion and deletion;
- a property list.

Marks

A mark is a pointer into a buffer. It contains:

- a pointer to its owning buffer;
- a position in its buffer, dynamically updated as the buffer changes;
- a direction, forward or backward.

The direction determines what happens when a character is inserted precisely at the mark's position: forward marks move forward, placing themselves after the new character, backward marks stay behind the new character. The buffer default-position mark is (initially) a forward mark. The window-position mark is (initially) a backward mark.

Fred Windows

Fred windows are used for on-screen editing. The variable `*fred-window*` holds the class of Fred windows. Fred windows inherit from `*window*`. In addition, a Fred window has

- a buffer, the buffer which is displayed in the window;
- a cursor mark, where the blinking caret appears in the window, and where typing is generally inserted into the window. The cursor is a forward mark in the window's buffer. It is *not* the default position mark of the buffer;
- a selection range, which is displayed in reverse video. Note that the selection range is distinct from the cursor, although most Fred window functions try to keep the cursor at one of the ends of the selection range;
- a window position, a backward mark in the first line of the buffer to be displayed in the buffer's window;
- a filename string.

Fred windows are output streams, and may be used in any situation which calls for a stream. Characters output to a Fred window stream are inserted at the cursor mark of the window. Stream

output to Fred windows is buffered, and will not be displayed until `window-update` or `force-output` is called.

Parameter Conventions

The descriptions below use the following conventions for argument description:

- A *place* is either a buffer or a mark. It will be coerced to a buffer or mark as needed. When a function which wants a mark is passed a buffer, it will generally use the buffer's default-position mark. When a function which wants a buffer is passed a mark, it uses the buffer associated with the mark.
- A *position* is 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. The *position* will often be an optional argument which defaults to the default-position of the given buffer.

The Kill-Ring

Fred combines the kill-ring with the Macintosh clipboard. Operations which normally operate from the clipboard (such as cut, copy, and paste), instead use the top item on the kill-ring. Clipboard interaction with other Macintosh programs works according to normal Macintosh standards.

The kill-ring is stored in the global variable `*killed-strings*`. It is a list of strings. The contents of the list may be freely manipulated.

Buffer Functions

buffer *place* [*Function*]
coerces *place* to a buffer. If *place* is already a buffer, it is returned. If *place* is a mark, the mark's buffer is returned. It is an error if *place* is neither a buffer nor a mark.

mark *place* [*Function*]
coerces *place* to a mark. If it is already a mark, it is returned. If it is a buffer, the buffer's default mark is returned. It is an error if *place* is neither a buffer nor a mark.

markp *form* [*Function*]
returns non-`nil` if and only if *form* is a mark, otherwise `nil`. The same as `(typep` *form* `'mark)`.

bufferp *form* [*Function*]
returns non-`nil` if and only if *form* is a buffer, otherwise `nil`. The same as `(typep` *form* `'buffer)`.

buffer-mark *place* [*Function*]
returns the default-position mark of `(buffer` *place*`)`.

buffer-position *place* `&optional` (*position* (`mark` *place*)) [*Function*]
Returns the position (number of characters from start of buffer) of *position* in (`buffer` *place*). If *position* is an integer, it is checked for being in the range of legal buffer positions and then

returned. If *position* is a mark in (`buffer` *place*), its position is returned. Otherwise, an error is signalled. This function is often used to check *position* arguments to other functions.

`mark-position` *place* [*Function*]
returns the current position (number of characters from the start of the buffer) of (`mark` *place*).

`make-mark` *place* `&optional` (*position* (`mark` *place*)) (*backward-p* `nil`) [*Function*]
creates and installs a new mark, with owner (`buffer` *place*), and the specified position and direction. If given, *position* must be a mark or an integer. The new mark is returned.

Note: Since each buffer maintains a list of all the marks it owns, marks don't always get garbage-collected. They stay around as long as the buffer stays around. You must keep track of the marks you create and explicitly dispose of them (with `kill-mark`*) when you are done with them. Use* `with-mark` *whenever you need a temporary mark.*

`kill-mark` *mark* [*Function*]
kills *mark*, removing it from its owner. Returns `t` unless *mark* was already dead, in which case it does nothing and returns `nil`. It is an error to pass a killed mark to any other Fred function.

If *mark* is essential to the operation of the system, for example if it is the default mark of a buffer or a window position mark of a window, then `kill-mark` will do nothing and return `nil`.

`with-mark` (*var place* [*position* (`mark` *place*)] [*backward-p* `nil`]) {*form*}* [*Macro*]
evaluates *forms* with *var* bound to a new mark created by (`make-mark` *place position backward-p*). The mark is killed when the `with-mark` form is exited.

`set-mark` *place position* [*Function*]
sets the position of (`mark` *place*) to *position*. Returns the updated mark.

`move-mark` *place* `&optional` (*distance* `1`) [*Function*]
moves (`mark` *place*) an amount specified by *distance*, which should be an integer. The same as (`set-mark` *place* (`+` (`mark-position` *place*) *distance*).

`mark-backward-p` *place* [*Function*]
returns `t` if (`mark` *place*) is a backward mark, otherwise `nil`.

`reverse-mark` *place* [*Function*]
reverses the direction of (`mark` *place*). The changed mark is returned.

`make-buffer` [*Function*]
returns a new, empty buffer.

`buffer-size` *place* [*Function*]
returns the number of characters in (`buffer` *place*).

`buffer-modcnt` *place* [*Function*]
returns the modification count of (`buffer` *place*). This 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.

Example:

```
(let ((start-count (buffer-modcnt buffer)))
   (maybe-do-something buffer)
   (unless (eql (buffer-modcnt buffer) start-count)
      (princ "Did something!")))
```

buffer-plist *place* [*Function*]
Returns the property list of (`buffer` *place*). `setf` may be used with `buffer-plist` to replace the entire property list. This is not recommended, since the system itself keeps certain information on buffer property lists.

buffer-getprop *place key* `&optional` *default* [*Function*]
looks up the *key* property on (`buffer-plist` *place*). Returns the value associated with the key, if found, otherwise *default*.

buffer-putprop *place key value* [*Function*]
gives *key* the value *value* on (`buffer-plist` *place*). *value* is returned.

buffer-line-start *place* `&optional` (*position* (`mark` *place*)) (*count* `0`) [*Function*]
returns the position of the start of the *count*th line from the line containing *position*. *count* of 0 means the start of that line, *count* of -1 means start of previous line, *count* of 1 means start of next line and so on. If there aren't enough lines in the buffer, returns the end of the range searched (start of buffer if *count* is negative, the end of buffer if *count* is positive) and a second value which is the number of lines of short-fall.

buffer-line-end *place* `&optional` (*position* (`mark` *place*)) (*count* `0`) [*Function*]
returns the position of the end of the *count*th line from the line containing *position*. *count* of 0 means the end of that line, *count* of -1 means end of previous line, *count* of 1 means end of next line and so on. If there aren't enough lines in the buffer, returns the end of the range searched (start of buffer if *count* is negative, the end of buffer if *count* is positive) and a second value which is the number of lines of short-fall.

buffer-column *place* `&optional` (*position* (`mark` *place*)) [*Function*]
returns the distance between *position* and the start of the line containing it.

buffer-line *place* `&optional` (*position* (`mark` *place*)) [*Function*]
returns the line number of *position* in the buffer. The first line is number 0, etc.

lines-in-buffer *place* [*Function*]
returns the number of lines in the buffer.

buffer-char *place* `&optional` (*position* (`mark` *place*)) [*Function*]
returns the character at the specified position in (`buffer` *place*).

buffer-char-replace *place char* `&optional` (*position* (`mark` *place*)) [*Function*]
replaces the character at the specified position in (`buffer` *place*) with *char*.

buffer-insert *place string* `&optional` (*position* (`mark` *place*)) [*Function*]
inserts *string* into (`buffer` *place*) at position *position*. *string* may actually be anything acceptable to the `string` function, that is, a string, a symbol or a character.

buffer-substring *place* &key (:start (mark *place*)) :end :length [*Function*]
returns a string of the characters in (buffer *place*) in the range described by the keyword arguments. Either :end or :length, but not both, must be specified. If :length is given, the range consists of :length characters starting at :start. If :end is given, the range consists of the characters between :start and :end. The returned string is always a simple string.

buffer-current-sexp-start-pos *place* &optional (*pos* (mark *place*)) [*Function*]
returns the starting position of the current s-expression, or nil if there is no current s-expression.

buffer-current-sexp *place* &optional (*pos* (mark *place*)) [*Function*]
returns two values. The first is the s-expression in (buffer *place*) at *pos*. This function actually reads the characters from the buffer, so you may evaluate what it returns. nil is returned if there is no s-expression at *pos*.
The second value returned is t if an s-expression was found at *pos*, or nil if no s-expression was found at *pos*.

buffer-delete *place* &key (:start (mark *place*)) :end :length [*Function*]
deletes the characters in (buffer *place*) in the range described by the keyword arguments. Either :end or :length, but not both, must be specified. If :length is given, deletes :length characters starting at :start. If :end is given, deletes the characters between :start and :end.

buffer-downcase-region *place start* &optional (*end* (mark *place*)) [*Function*]
buffer-upcase-region *place start* &optional (*end* (mark *place*)) [*Function*]
buffer-capitalize-region *place start* &optional (*end* (mark *place*)) [*Function*]
lower-cases, upper-cases or capitalizes all words between *start* and *end*.

buffer-char-pos *place chars* &key (:start (mark *place*)) [*Function*]
(:from-end nil)
(:end (if :from-end 0 t))
buffer-not-char-pos *place chars* &key (:start (mark *place*)) [*Function*]
(:from-end nil)
(:end (if :from-end 0 t))
buffer-char-pos returns the position of the first occurrence of a character which is an element of *chars* in the buffer between :start and :end. *chars* may be a string or a character. If :from-end is non-nil, the search is backwards. The search is case-sensitive, i.e., the comparison is done using char=. If the character is not found, nil is returned.
buffer-not-char-pos performs the same function, except that it returns the first character in the buffer that is *not* one of *chars*.

buffer-string-pos *place string* &key (:start (mark *place*)) [*Function*]
(:from-end nil)
(:end (if :from-end 0 t))
returns the position of the first occurrence of *string* in the buffer between :start and :end. An error is signalled if :start is not less than :end. If :from-end is non-nil, the search will proceed backwards. 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, i.e., the comparison is done using char=.

buffer-substring-p *place string* `&optional` (*position* (`mark` *place*)) [*Function*]
returns `t` if *string* appears at the specified position in (`buffer` *place*). The comparison is case insensitive. *string* may be a string or a character.

buffer-word-bounds *place* `&optional` (*position* (`mark` *place*)) [*Function*]
returns two values, the start and end of the word at *position*.

buffer-fwd-sexp *place* `&optional` (*position* (`mark` *place*)) [*Function*]
returns the position of the end of the s-expression which starts at *position*.

buffer-bwd-sexp *place* `&optional` (*position* (`mark` *place*)) [*Function*]
returns the position of the start of the s-expression which ends at *position*.

buffer-insert-file *place pathname* `&optional` (*position* (`mark` *place*)) [*Function*]
inserts the file specified by *pathname* into (`buffer` *place*) at *position*.

buffer-write-file *place pathname* `&key` (`:if-exists :error`) [*Function*]
outputs the contents of the buffer to the file specified by *pathname*. *if-exists* specifies what to do if the file already exists. If it is `:error`, an error is signalled. If it is `:supercede`, the file is deleted and a new file is written. If it is `:overwrite`, the data fork of the existing file is replaced with the contents of the buffer, and the resource fork remains unchanged.

Fred Window Functions

In addition to the functions outlined below, many Fred functions are associated with keystrokes. The names and actions of these functions are given in the chapter Using Fred. They are defined only for Fred windows, not globally. In addition, most of the keystroke functions are not exported, so you must refer to them using the `CCL::` prefix, for example `CCL::ed-forward-char`.

You can create a new Fred window with the call `(oneof *fred-window*)`.

exist *init-list* [**fred-window* Function*]
initializes a Fred window. The standard window init-list arguments are allowed, in addition to the following:

keyword	type	default	meaning
`:buffer`	*buffer*	`(make-buffer)`	The buffer to be displayed in the Fred window.
`:filename`	*pathname*	`nil`	if given, specifies a file to read into the buffer. The file type defaults to `.lisp`.
`:scratch-p`	*boolean*	`nil`	if non-`nil`, the user will not be prompted to save the contents of the buffer when it is closed, and the modified marker will not appear in the buffer's title.

`:package` ..... *package* ..... `nil` ..... if given, associates a package with the window. Commands which evaluate expressions from the buffer (e.g. `window-eval-selection`) will bind `*package*` to the specified package before reading. If `nil`, the current package is not changed when evaluating from the window.

If both `:buffer` and `:filename` are specified, the buffer is erased before the file is read in.

`window-buffer` [**fred-window* Function*]
returns the buffer displayed in the window.

`window-update` [**fred-window* Function*]
updates the window display of the window, using the current values of the cursor-mark, window-position mark, selection region and the contents of the buffer.

`window-update` 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 mouse-clicks in the window; and in response to `force-output` calls to the window (windows inherit from output streams).

If you modify the window (or its buffer) in any other context, for instance in a menu command or in a function meant to be called directly by user code, you must call `window-update` explicitly or the changes you make will not become visible on the screen.

`window-start-mark` [**fred-window* Function*]
returns the window-position mark of the window. The line containing this mark is always the first line drawn in the window. By moving this mark you affect which part of the window buffer is displayed by `window-update`.
For example:

```
(ask the-window
  (set-mark (window-start-mark) 0)
  (window-update))
```

will cause the beginning of the buffer in `the-window` to be visible.

Note that after every Fred keyboard command, Fred will try to make sure the cursor is visible on the screen, repositioning the `window-start-mark` if necessary. To disable this behavior for the duration of one Fred command, set the variable `*show-cursor-p*` to `nil`.

`window-cursor-mark` [**fred-window* Function*]
returns the cursor mark of the window. `window-update` will draw the blinking vertical bar wherever this mark is located (unless the mark is off the screen, in which case no vertical bar will be drawn). Note that the cursor mark is *not* the same as the default-position mark of the window's buffer.

window-point-position *h* `&optional` *v* [**fred-window* Function*]
returns the buffer position of the character nearest to the point specified by *h* and *v* in the local coordinates of the window. (See the chapter Macintosh Basics for a description of the point format). This function assumes that the buffer has not been modified since the last call to `window-update`.

window-hpos `&optional` (*pos* (`window-cursor-mark`)) [**fred-window* Function*]
returns the horizontal position of the character at the given *pos* in the window's buffer. The value is given in local window coordinates. The position is computed as the length (in pixels) of the line containing *pos*, minus the amount of horizontal scrolling currently in effect in the window.

window-line-vpos *line-number* [**fred-window* Function*]
returns the vertical position (in local window coordinates) of the baseline of the *line-number*th line in the window.

selection-range [**fred-window* Function*]
returns two values giving the beginning and end of the current selection. If there is no selection, it returns the cursor mark position as both values. You can test if there is a selection by checking whether the two values are `eql`.
Selections in the top window are displayed in reverse video.

set-selection-range `&optional` (*pos* (`window-cursor-mark`)) [**fred-window* Function*]
sets the current selection to the range of the buffer between *pos* and the cursor, and updates the window display. If *pos* is equal to the cursor position, the selection range is made empty.

collapse-selection *forward-p* [**fred-window* Function*]
If there is no selection, this function does nothing and returns `nil`. Otherwise, it sets the cursor mark to the end of the selection if *forward-p* is non-`nil`, or the beginning of the selection if *forward-p* is `nil`, de-selects the selection, and returns `t`. The Macintosh user interface guidelines suggest that cursor motion commands collapse the selection in this way and not perform their actions when there is a selection. The following function shows how to use `collapse-selection` in order to conform to the guidelines:

```
(def-fred-command (:meta #\3) ed-forward-3 "Move forward by 3")
(defobfun (ed-forward-3 *fred-window*) ()
  (unless (collapse-selection t)
    (move-mark (window-cursor-mark) 3)))
```

window-filename [**fred-window* Function*]
returns the filename associated with the window (as a string), or `nil` if there is none. Files become associated with windows when `set-window-filename` is called, or if the window was created with the `:filename` initlist argument.

set-window-filename *filename* [**fred-window* Function*]
sets the filename associated with the window. When the window contents are saved, they will be saved to the new filename. If a file corresponding to the filename already exists, it will be overwritten without warning when the window is next saved.

window-package [**fred-window* Function*]
returns the package associated with the window, or `nil` if there is none.

set-window-package *package* [**fred-window* Function*]
sets the package associated with the window. *package* can be a package or symbol whose name is the name of a package.

window-save [**fred-window* Function*]
saves the window to its disk file. If the window is not associated with a disk file, `window-save-as` is called.

window-save-as [**fred-window* Function*]
calls the standard save-as dialog box, allowing the user to choose a directory and input a filename, and saves the contents of the window to the filename. Note: if the user selects the Cancel item from this dialog, Allegro CL will throw to `:cancel`. User code may wish to catch cancel to prevent a return to top-level.

window-revert [**fred-window* Function*]
reverts the window to the last version saved. The user will be asked to confirm the reversion before it is performed.

window-hardcopy [**fred-window* Function*]
prints the contents of the window to the current hardcopy device. Before printing takes place, the user will be prompted for various printer options.

setup-undo *undo-function* `&optional` *title* [**fred-window* Function*]
allows Fred commands to support the **Undo** menu-item. Any undo-able Fred action should call `setup-undo`. *undo-function* should be a function to call when the **Undo** menu-item is chosen. If given, *title* should be a short string which will be used as the title of the menu-item.

The **Undo** menu-item will be enabled when the effected Fred window is the top window, and as long as the `buffer-modcnt` of the window's buffer does not change.

Example

```
;this function inserts the string "hello" at the cursor position.
;it supports undo and redo.

(defobfun (insert-hello *fred-window*) (&aux buf start-pos)
  (setq buf (window-buffer)
        start-pos (mark-position (window-cursor-mark)))
  (buffer-insert buf "hello" start-pos)
  (setup-undo #'(lambda ()
                  (buffer-delete buf
                                 :start start-pos
                                 :end (window-cursor-mark))
                  (window-update)
                  (setup-undo #'(lambda ()
                                  (insert-hello)
                                  (window-update))
                              "Redo Hello"))
                "Undo Hello"))
```

cut [**fred-window* Function*]
deletes the current selection from the buffer and adds it to the kill-ring/clipboard.

copy [**fred-window* Function*]
adds the current selection to the kill-ring/clipboard. The selection is not removed from the buffer.

paste [**fred-window* Function*]
replaces the current selection with the text from the top of the kill-ring/clipboard. The kill-ring is not affected. If the kill-ring is empty, no action is taken. If there is no selection, the text from the kill-ring is inserted at the caret.

select-all [**fred-window* Function*]
sets the current selection to the entire contents of the buffer.

Fred Command Tables

This section describes the functions which allow the user to associate Lisp functions with keystrokes.

In normal operation, keystrokes are handled by the front-most window. Fred treats every keystroke typed at a Fred window as a command. Associated with every possible keystroke is a Lisp function which implements the command. Some commands are simple, for example typing `#\a` is a command to insert `#\a` in the buffer; some are more complex, for example typing control-meta-f is a command to move forward by one s-expression. Fred makes no distinction between these two kinds of commands. Indeed, you could easily redefine `#\a` to do something complicated.

When you type a key in a Fred window, Fred first translates it into a *keystroke code*. The keystroke code contains a character and two flags called *control* and *meta*. It is encoded as a small integer, with the character code in the least significant 8 bits, the control bit in bit 9 and the meta bit in bit 8.

The binding between keystrokes and the functions they invoke is stored in a data structure called a *comtab* (short for *command table*). There is a global comtab stored in the variable `*comtab*`. In addition, each Fred window may contain a local comtab in an instance variable `comtab`. The bindings in the local comtab override global bindings.

When Fred receives a keyboard event, it translates the event to a keystroke code with the function `event-keystroke`, finds the function associated with the keystroke by using `keystroke-binding`, and then invokes the function with no arguments. When the function is invoked, the current object is the Fred window which received the event (for this reason, Fred command functions are usually `*fred-window*` object-functions). When the function 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. This is useful for functions which scroll the display).

event-keystroke *message modifiers* [*Function*]
takes the message and modifiers fields of a Macintosh event record and returns a keystroke code. It sets the control bit if either the Control key or the Command/shift-Command key (depending on the value of `*emacs-mode*`) was pressed, and the meta bit if the Option key was pressed. The character portion of the keystroke code is set to the ASCII code in the message field; if the Option key was pressed the character portion will contain the standard, rather than the optional character (i.e. it will be #\s rather than #\ß when you type Option-s). This function is called by Fred when it

receives a key-down event. You may redefine this function if, for example, you don't like Fred's default translation of the Macintosh modifier keys into control/meta bits.

keystrcke-code *keystroke-name* [*Function*]
translates a keystroke name to a keystroke code. A keystroke name is either a character, or a list of the form `(:control :meta` *character*`)` or `(:control` *character*`)` or `(:meta` *character*`)`. *keystroke-name* may also be a keystroke code (an integer), in which case it is simply returned.

keystroke-name *keystroke-code* [*Function*]
returns the name of a keystroke code.

make-comtab [*Function*]
returns an empty comtab.

copy-comtab `&optional` (*from-comtab* **comtab**) [*Function*]
returns a new comtab which is initially functionally equivalent to *from-comtab*. If *from-comtab* is specified as `nil`, then it returns a copy of the comtab which was in use when Allegro CL was launched. This is useful if you've badly mangled the current comtab.

comtab-set-key *comtab keystroke function* `&optional` *doc-string* [*Function*]
sets the definition of *keystroke* to *function*. *function* may be a symbol, a compiled function, or a lambda expression (a symbol is recommended), indicating a function to call when *keystroke* is entered. It may also be a comtab, indicating that the keystroke is a prefix character (like control-x) which reads another character and looks it up in its own comtab. It may also be another keystroke (name or code) to indicate that *keystroke* should do whatever the other keystroke would do. Finally, function can also be `nil`, in which case *keystroke* will be made undefined. *keystroke* may be a keystroke code or a keystroke name.

def-fred-command *keystroke function* `&optional` *doc-string* [*Macro*]
is equivalent to `(comtab-set-key *comtab* '`*keystroke* `'`*function doc-string*`)`.

Example
```
? (def-fred-command (:meta #\h) insert-hello)
#<A COMTAB>
```

comtab-get-key *comtab keystroke* [*Function*]
looks up the definition of *keystroke* in *comtab*. This is the reverse of `comtab-set-key`. The value may be a symbol, a compiled function, a lambda expression, another keystroke or `nil`. *keystroke* may be a keystroke code or a keystroke name.

Example
```
? (comtab-get-key *comtab* '(:meta #\h))
INSERT-HELLO
```

keystroke-function *keystroke* [*Fred-window Function*]
does the full Fred command look-up for *keystroke*. Always returns a function or a comtab, never `nil` or another keystroke. First looks up the keystroke in the local comtab, or `*comtab*` if there is no local definition. If the definition is another keystroke, starts the look-up from the beginning. If no definition exists, returns `#'ed-beep` if the (original) keystroke had control or meta set, otherwise it returns `#'ed-self-insert`.

comtab-key-documentation *comtab keystroke* [*Function*]
returns the doc string associated with *keystroke*. *keystroke* may be a keystroke code or keystroke name.

comtab-find-keys *comtab function* [*Function*]
returns a list of all keystrokes which are bound to functions in the *comtab*. Comparison is done using `eql`.

comtab [*Variable*]
the global comtab. You may modify this very comtab, or set the variable to a totally new comtab.

listener-comtab [*Variable*]
the listener's local comtab; whenever a new a new listener is created, the listener's local comtab is set to the value of this variable. By modifying this comtab, you may change the behavior of the listener without affecting other Fred windows. Note that setting this variable to a new comtab (as opposed to modifying the comtab it is set to) will only affect listeners created after the change.

comtab [**fred-window* Variable*]
the window's local comtab.

File System Interface 10

Overview
Pathname Specification
Common Lisp Pathnames
Parsing Pathname Strings
    Parsing Examples
Pathname Escape Character
Using Lisp Pathnames
Macintosh Pathnames
Default Directories
    Macintosh Defaults
    Allegro CL Defaults
Search Path
Wildcards
Logical Pathnames
File System Manipulation
    File and Directory Manipulation
    File Operations
    Volume Operations
    Directory Operations
    User Interface

File System Interface

Overview

This document describes pathname specification and the functions present for manipulating the Macintosh file system. It assumes some familiarity with section 23.1 of *Common Lisp: the Language*.

Pathname Specification

A pathname is a way to specify a particular directory or file. Common Lisp specifies pathnames to have six components: host, device, directory, filename, type and version number. In the Macintosh world, pathnames have three components: directory, filename and an optional volume number. The Common Lisp and Macintosh representations each have advantages, so both are made available. Lisp pathnames are simply referred to as pathnames and Macintosh pathnames are referred to as mac-pathnames.

Common Lisp Pathnames

The host, device, and version components of Common Lisp pathnames are currently ignored in file system operations. However, a standard namestring is defined for them to allow for future extensions, or user-written extensions.

To make a pathname one can use a string representation or use the `make-pathname` function.

In Allegro CL, Lisp pathnames are printed using the `#.` reader macro.

Example:

```
? (make-pathname :directory "hd:" :name "foo")
#.(pathname "hd:foo")
```

All functions that are specified to take pathname arguments also take strings as arguments, so it will hardly ever be necessary to use `(pathname "hd:foo")`. Instead, one can use `"hd:foo"`.

Parsing Pathname Strings

The pathname parser uses the following rules to break strings into five components: host, directory, filename, file type, and version. Unspecified components are represented as `nil`.

- Host
  All characters up to the last exclamation mark character "`!`" in the string are part of the host description.
  Because the host component is ignored, the following two pathnames are equivalent

```
(pathname "coral-vax!coral-macs!j-mac!hd:foo.Lisp")
(pathname "hd:foo.Lisp")
```

- Device
  The device component is currently ignored.

- Directory
  The directory component is identified as the characters between the host component and the last colon or semicolon. The colon is the standard separator character for directories. The semicolon is introduced to implement logical pathnames (see section on logical pathnames) and to allow programmers access to both the Macintosh and Common Lisp notion of the default directory (see section on default directories). A directory name that begins with a colon inherits from the Macintosh default directory. A leading semicolon indicates inheritance from the Lisp defaults.

  The colons and semicolons are not merely separators. They are actually part of the directory component.

  The device section of the pathname is the first directory in the directory tree.

- File Name
  The filename is composed of the characters that follow the directory component until either the end of the string or the first period. The period is only a separator and is not actually part of the filename. To make a filename containing a period, use escape characters as described below. To specify a file that has an empty string as its filename, use a single period after the directory separator character. There is currently no namestring representation for a pathname which has a file type and an unspecified file name.

- File Type
  The file type is composed of the characters that follow the name component until either the end of the string or the first comma. The comma is only a separator and is not actually part of the file type. To make a file type containing a comma, use escape characters as described below. As a convention, the type `"lisp"` is used as the type for source code files and `"fasl"` is used for compiled files.
  These file types should not be confused with Macintosh file types. Macintosh file types are not part of the pathname of a file.

- Version
  The version is composed of the characters that follow the type component until the end of the string. The Macintosh does not support versions for pathnames. Allegro CL does not implement any support.

Parsing Examples

The following table contains some examples of pathname parsing:

	pathname Components			
Namestring	*Directory*	*File name*	*File type*	*version*
`"hd:foo.Lisp"`	`"hd:"`	`"foo""`	`Lisp"`	`nil`
`":foo"`	`":"`	`"foo"`	`nil`	`nil`
`";foo"`	`nil`	`"foo"`	`nil`	`nil`
`"foo"`	`nil`	`"foo"`	`nil`	`nil`
`"foo."`	`nil`	`"foo"`	`nil`	`nil`
`"foo.,"`	`nil`	`"foo"`	`""`	`nil`
`";sub-dir:foo.,"`	`";sub-dir:"`	`"foo"`	`""`	`nil`
`"foo.fasl"`	`nil`	`"foo"`	`"fasl"`	`nil`
`"foo.fasl,"`	`nil`	`"foo"`	`"fasl"`	`nil`
`"hd:"`	`"hd:"`	`nil`	`nil`	`nil`
`"hd:."`	`"hd:"`	`""`	`nil`	`nil`
`"hd:.,"`	`"hd:"`	`""`	`""`	`nil`
`"ccl;foo"`	`"ccl;"`	`"foo"`	`nil`	`nil`

"ccl;foo."	"ccl;"	"foo"	""	nil
"ccl;foo.."	"ccl;"	"foo"	"."	nil
"ccl;foo..,,"	"ccl;"	"foo"	"."	","
"hd:foo.Lisp,2"	"hd:"	"foo"	"Lisp"	"2"
"Coral!hd:foo.Lisp,2"	"hd:"	"foo"	"Lisp"	"2"
"hd:pr,4.1:foo.Lisp,2.1"	"hd:pr,4.1"	"foo"	"Lisp"	"2.1"
"hd:sub-dir:foo.text"	"hd:sub-dir:"	"foo"	"text"	nil
"log-dir;subdir:foo"	"log-dir;subdir:"	"foo"	nil	nil

Table: Pathname Parsing Examples.

Neither defaults nor logical pathnames are merged at parse time. The function `merge-pathname` performs mergings by replacing `nil` components of its first argument with corresponding components of its second argument. The function `expand-logical-pathname` performs the logical to physical pathname translation.

Pathname Escape Character

Although the addition of exclamation marks, semicolons, periods, commas, and asterisks to Allegro CL's pathnames adds functionality, it also limits the use of these characters as part of a pathname. To alleviate this problem, pathnames have a special escape character, #\∂ (Option-d). This escape character works very much like the backslash character in strings.

Any character that is preceded by a "∂" loses any special meaning it might have had in a pathname. The table below illustrates this mechanism:

	Pathname Components			
Namestring	*Directory*	*File name*	*File type*	*version*
"hd:foo∂.Lisp"	"hd:"	"foo∂.Lisp"	nil	nil
"hd:foo.Lisp"	"hd:"	"foo"	"Lisp"	nil
":fo∂o∂.,"	":"	"foo∂.,"	nil	nil
";foo∂.∂,"	";"	"foo∂.,"	nil	nil
";foo∂.∂,.fasl"	";"	"foo∂.,"	"fasl"	nil
";ccl∂;foo"	";"	"ccl∂;foo"	nil	nil
"ccl;fo∂∂o"	"ccl;"	"fo∂∂o"	nil	nil
"Coral∂!hd:foo.Lisp,2"	"coral∂!hd:"	"foo"	"Lisp"	"2"
"hd:fo\"o.Lisp"	"hd:"	"fo\"o"	"Lisp"	nil

Table: Parsing Pathnames with escapes.

Only the *needed* escape characters are retained (i.e. the "∂" before the "o" in the third line is removed, but the one before the period is retained). Of course, this mechanism is meant to work only for the Allegro CL additions; one is not and should not be able to specify a filename that includes a colon, since such a filename is not allowed on the Macintosh.

Note that the escape characters are not part of the truename. They are included only in the Lisp representation of the pathname, not in the Macintosh system's representation of the pathname.

Using Lisp Pathnames

make-pathname &key :host :device :directory :name :type :version :defaults [*Function*]

This Common Lisp function 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 default value of the :defaults argument is a pathname whose host component is the same as the host component of *default-pathname-default*, and whose other components are all nil.

All arguments to make-pathname should be strings or nil. nil specifies that a component should be taken from the default. Directory components beginning with a colon specify inheritance from the Macintosh default directory. make-pathname will endeavor to insert the appropriate escape characters in components which need them. The user need only insert escape characters in front of semicolons that are part of directory components, and in front of ∂'s.

Examples:

```
? (make-pathname :directory "Hd:" :name "foo" :type "Lisp")
#.(pathname "Hd:foo.Lisp")
? (make-pathname :directory ";" :name "foo" :type nil)
#.(pathname "foo")
? (make-pathname :directory nil :name "foo." :type "fasl")
#.(pathname "foo∂.fasl")
? (make-pathname :directory nil :name "foo." :type ",")
#.(pathname "foo∂..∂,")
? (make-pathname :directory "hd;" :name "foo." :type "Lisp")
#.(pathname "hd;foo∂..Lisp")
? (make-pathname :directory ";subdir:" :name "foo.")
#.(pathname ";subdir:foo∂.")
? (make-pathname :name "foo∂")
#.(pathname "foo")
? (make-pathname :name "foo∂∂")
#.(pathname "foo∂∂")
? (make-pathname :host "Coral-vax" :device "bobo" :name "foo" )
#.(pathname "foo")
? (make-pathname :directory "hd:" :type "fasl")
#.(pathname "hd:.fasl")
```

The last example shows a peculiarity of specifying pathnames using namestrings. If there is a type specified but there is no name specified, then the print representation will not read correctly. When read, the name will be taken as the empty-string, rather than as unspecified. The same situation arises if the version is specified without the preceding fields being specified.

merge-pathnames *pathname* &optional *defaults default-version* [*Function*]

This is the Common Lisp function which is generally called to process a filename supplied by the user. It fills in unspecified components of *pathname* from *defaults* and returns a new lisp pathname. The *pathname* and *defaults* arguments may each be a string, a stream, a lisp pathname, or a mac-pathname. The returned value will always be a lisp pathname. *defaults* defaults to the value of *default-pathname-defaults*. *default-version* defaults to nil.

The merge operation replaces the `nil` components of *pathname* with corresponding components from *defaults*, with the following caveats for merging versions: If a pathname does not specify a name, then the version, if not provided, will come from *defaults*. However, if the pathname *does* specify a name, then the version is not affected by *defaults*; it is taken instead from *default-version*. *default-version* defaults to `nil`.

pathname-host *pathname* [*Function*]
pathname-device *pathname* [*Function*]
pathname-directory *pathname* [*Function*]
pathname-name *pathname* [*Function*]
pathname-type *pathname* [*Function*]
pathname-version *pathname* [*Function*]
These Common Lisp functions accept strings, mac-pathnames, streams and Lisp pathnames as arguments. They build a corresponding Lisp pathname, if necessary, and return the specified component of that Lisp pathname.

namestring *pathname* [*Function*]
file-namestring *pathname* [*Function*]
directory-namestring *pathname* [*Function*]
host-namestring *pathname* [*Function*]
enough-namestring *pathname* `&optional` *defaults* [*Function*]
These Common Lisp functions accept strings, mac-pathnames, streams and Lisp pathnames as arguments. `namestring` returns the full form of the *pathname* as a string. `file-namestring` returns a string representing only the *name*, *type* and *version* components of the pathname. `directory-namestring` returns a string representing only the `directory-name` portion. `host-namestring` returns a string for only the *host* portion. Note that the string representation of unspecified components is an empty string.

lisp-pathnamep *thing* [*Function*]
This non-Common Lisp function returns non-`nil` if *thing* is a Lisp pathname. It returns `nil` if *thing* is anything else, including a mac-pathname, string, or stream.

pathnamep *datum* [*Function*]
This Common Lisp function returns `t` if *datum* is either a Lisp pathname or mac-pathname.

```
(pathnamep form) = (or (lisp-pathnamep form) (mac-pathnamep form))
```

Macintosh Pathnames

Mac-pathnames are much simpler than Common Lisp pathnames. Mac-pathnames know nothing about logical directories, types, or version numbers. They have two basic components: directory-name and filename. The directory-name consists of the Common Lisp directory component, and the filename consists of the concatenation of the Common Lisp name, type, and version components.

A third optional component is introduced to distinguish between mounted volumes with the same name. This component holds the volume reference number assigned by the Macintosh to mounted volumes. The user is not expected to specify this field; the system functions that return mac-pathnames will insert the correct volume reference number. Functions that use mac-pathnames

refer to this number when there is an ambiguity caused by mounted volumes having the same name.

Mac-pathnames can be used everywhere other pathnames are used. They are printed using using the #P reader macro.

```
? (make-mac-pathname :directory "hd:ccl:"
                     :name "foo.Lisp"
                     :volume-number -2)
#P"hd:ccl:foo.Lisp"
```

Because volume numbers may vary between programming sessions, they are not used in a mac-pathname's printed representation.

The following functions are provided for manipulating mac-pathnames.

make-mac-pathname &key :directory :filename :volume-number [*Function*]
This non-Common Lisp function constructs and returns a mac-pathname. The :directory and :filename components are strings and the :volume-number must be an integer. This function inserts a colon at the end of the :directory argument if there is not one there, and errors if there is a colon in the :filename argument.

mac-pathname *thing* &optional (*volume* 0) [*Function*]
returns a mac-pathname that corresponds to *thing*. *thing* should be a string, stream, Lisp pathname or a mac-pathname. String arguments are first parsed as a lisp pathname and then converted to mac-pathnames.

The volume component will remain unspecified unless the optional *volume* argument is supplied. mac-pathname will signal an error if it cannot convert its argument to a mac-pathname.

mac-filename *pathname* [*Function*]
mac-directory *pathname* [*Function*]
mac-volume *pathname* [*Function*]
pathname is converted to a mac-pathname, and appropriate component is returned. *pathname* can be a string, stream, Lisp pathname, or mac-pathname.

mac-namestring *pathname* [*Function*]
returns a string that represents how the Macintosh would represent the particular pathname.

```
(mac-namestring pathname) = (concatenate 'string
                              (mac-pathname-directory pathname)
                              (mac-pathname-name pathname))
```

Default Directories

There are three different directories maintained by Allegro. These are used to fill in pathname components specified as default.

Macintosh Defaults

The Macintosh maintains a default directory of its own. Any pathname that begins with a colon specifies the Macintosh default directory. Using the Macintosh default directory is useful with mac-pathnames. Beware that desk accessories and other background processes may change the

default directory without notice. It is therefore suggested that one set the Macintosh default directory directly before accessing it.

The Macintosh default directory is initially the directory holding Allegro CL.

mac-default-directory [*Function*]
returns the Macintosh default directory.

set-mac-default-directory *pathname* [*Function*]
sets the Macintosh default directory to the directory component of *pathname*.

Allegro CL Defaults

default-pathname-defaults [*Variable*]
is used for merging if the default-pathname argument is not specified. The initial value is `#.(pathname "")`.

working-directory [*Variable*]
is used in merging arguments to `probe-file`. `probe-file` is called by all functions that access files from disk. There may be an initial merging by the function which called `probe-file`, but in any case, probe-file does a final merging with `*working-directory*` before actually passing the name to the Macintosh operating system. The initial value is `#.(pathname "")`.

user-homedir-pathname `&optional` *host* [*Function*]
returns a mac-pathname that represents the directory pointed to by the logical pathname `"home;"` (see logical pathnames section).

Search Path

If a pathname has an empty directory component, the Macintosh will look for it in the Macintosh default directory. If one wants to insure that the Macintosh default is used, then one must precede the directory name with a colon. To ensure that the Macintosh default is not used, specify the directory component of the pathname (perhaps by doing a merge with some other default).

Wildcards

Wildcards are implemented for searching the file structure for particular directories and files with the Common Lisp function `directory`. `directory` takes a pathname as argument and returns a list of matching pathnames on the system. For example, `(directory "hd:system folder:")` will return `(#P"hd:system folder:")` if the specified directory exists and `nil` otherwise.

The wildcards are used in three different ways:

- One asterisk, "*", matches zero or more characters within a component.
- Two asterisks, "**", in place of a directory matches the present directory, its subdirectory, all their subdirectories until there are no subdirectories left.
- Two asterisks, "**", in place of the filename components matches any number of components that are left.

The following examples assume that there is a mounted disk with the name "hd".

- To get a list of subdirectories under "hd:" evaluate (directory "hd:*:")
- To get a list of files under "hd:" evaluate (directory "hd:*.*,*") or (directory "hd:**")
- To get a list of all the subdirectories at all levels in all the devices known to the machine, evaluate (directory "**:")
- To get a list of all the files in all the devices known to the machine, evaluate (directory "**:**")
- To get a list of all the files in "hd:" of type "Lisp", evaluate (directory "hd:*.Lisp,*")
- To get a list of the files directly under "hd:" that do not have a type and version number specified, evaluate (directory "hd:*")
- To get 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:", evaluate (directory "**:ccl:*:*:prin*12.**")
- Note that while (directory "h*d:") is equivalent to (directory "h********d:"), (directory "hd:*:) is not equivalent to (directory "hd:*:*:"). The first and second commands return a list of all the devices known to the machine that start with the letter "h" and end with the letter "d". The third command returns a list of the subdirectories under "hd:". The fourth command returns a list of all the directories that are two levels below "hd:".

Logical Pathnames

Logical pathnames serve as variables in a pathname string. Logical pathnames let code with embedded pathname information run under different directory hierarchies.

Logical directories end with semicolons, as opposed to physical directories which end with colons.

When Allegro CL is run, four logical pathnames are set up automatically.
"ccl;" is set to the directory holding the Allegro CL application.
"home;" is set to the directory holding the document which was launched with Allegro CL.
"ccl-doc;" is set to the directory "ccl;ccl-doc:"
"library;" is set to the directory "ccl;library:"

logical-pathname-alist [*Variable*]
an association list which maps between logical and physical pathnames.

def-logical-pathname *logical-string physical-pathname* [*Function*]
defines a new logical pathname and adds it to *logical-pathname-alist*. *logical-string* becomes the name of the logical pathname and *physical-pathname* becomes the associated physical pathname. *physical-pathname* may also contain logical components. To remove a logical pathname from the environment call def-logical-pathname with a *physical–pathname* of nil.

expand-logical-pathname *pathname* &key :no-error [*Function*]
returns a pathname that has all its logical components expanded into physical components. expand-logical-pathname expands logical directories from left to right, calling itself recursively until there are no logical pathnames left.

If there is no physical directory for a logical directory in *pathname*, an error will be signalled or `nil` will be returned, depending on the value of `:no-error`.

`expand-logical-namestring` *name-string* `&key :no-error` [*Function*]
performs the same function as `expand-logical-pathname` except that it takes a string as an argument, and returns a string as its result.

Examples:

```
? (def-logical-pathname "misc" "hd:ccl-misc")  ;creates the logical to
NIL                                            ; physical mapping
? (load "misc;aloysius.Lisp")     ;will load the file "hd:ccl-misc:aloysius.Lisp"
"hd:ccl-misc:aloysius.Lisp"
? (expand-logical-namestring "misc;aloysius.Lisp")
"hd:ccl-misc:aloysius.Lisp"
? (expand-logical-namestring "MISC;aloysius.Lisp")
"hd:ccl-misc:aloysius.Lisp"                    ;note case insensitivity
? (expand-logical-namestring "MISC;sub-dir:aloysius.Lisp")"hd:ccl-
misc:sub-dir:aloysius.Lisp"
? (expand-logical-namestring "misc:aloysius.Lisp")
"misc:aloysius.Lisp" ;no semicolon in the input, so no modification
? (def-logical-pathname "misc" "hd:music")     ;redefine misc
NIL
? (expand-logical-namestring "misc;aloysius.Lisp")
"hd:music:aloysius.Lisp"                       ;uses new mapping
```

File System Manipulation

The following functions return mac-pathnames or lists of mac-pathnames, unless otherwise noted.

File and Directory Manipulation

These functions operate on both directories and files. A directory operation is performed if the filename component is empty (i.e. if the pathname ends in a colon or semicolon), otherwise a file operation is performed.

The functions operate on pathnames, mac-pathnames, strings and streams.

The functions that create a new file or directory have an `:overwrite` keyword. If the `:overwrite` keyword is non-`nil`, then the function will delete anything standing in its way. If the keyword is specified as `nil`, the function will not delete any file or directory and simply return `nil` as sign of failure. If `:overwrite` is not specified, the user is given the option of specifying a new name, overwrite the files in question, or abort computation.

`rename-file` *old-file-or-dir new-file-or-dir* `&key :overwrite` [*Function*]
the specified *old-file-or-dir* is renamed to the result of merging *new-file-or-dir* with *old-file-or-dir*. Both arguments may be a string, stream, lisp pathname, or a mac-pathname. If *new-file-or-dir* is an open stream associated with a file, then the stream itself and the file associated with it are affected.

`rename-file` returns three values if successful. The first value is the renamed *old-file-or-dir*.

The second value is the truename of the *old–file–or-dir* before it was renamed. The third value is the truename of the *old–file–or-dir* after it was renamed. An error is signalled if the renaming operation is not successful.

move-file *file–or–dir new–dir* `&key :overwrite` [*Function*]
the specified *file–or–dir* is moved to the directory component of *new–dir*. Arguments may be strings, pathnames, mac-pathnames or streams. If the first argument is a directory then the whole directory with all its contents is moved to the new location. If the first argument is an open stream associated with a file, then the stream itself and the file associated with it are affected.

If *file–or–dir* and *new–dir* are on different volumes, `move-file` copies *file–or–dir* to *new–dir* and then deletes *file–or–dir* from its original volume. However, when the two arguments are on the same volume, `move-file` performs a fast catalog operation that does not involve moving the actual data of the file.

`move-file` returns three values if successful. The first value is the moved *file-or-dir*. The second value is the truename of the *file–or–dir* before it was moved. The third value is the truename of the *file–or–dir* after it was moved. An error is signalled if the operation is not successful.

delete-file *file–or–dir* `&key :error-if-no-exist :overwrite` [*Function*]
The specified *file–or–dir* is deleted. An error is signalled if *file-or-dir* does not exist and `:error-if-no-exist` is non-`nil` (the default is `t`).

The `:overwrite` keyword is consulted in the regular manner if *file–or–dir* specifies a non-empty directory. If `:overwrite` is not specified, the user is given the chance to delete a different directory or to choose a file—rather than a directory—to delete. `delete-file` returns the truename of the deleted file if successful, otherwise `nil`.

create-file *file–or–dir* `&key :overwrite` [*Function*]
`:mac-file-type`
`:mac-file-creator`
creates an empty file or directory named *file–or–dir*.

For files, the `:mac-file-type` and `:mac-file-creator` keywords default to `:TEXT` and `|:CCL |` (`#\space` is the last character in `|:CCL |`). The keywords are case-sensitive. Directories do not have Macintosh types or creators. `create-file` returns the truename of the created file or directory. An error is signalled if the operation is not successful.

probe-file *file–or–dir* [*Function*]
returns `nil` if there is no file or directory named *file–or–dir*, otherwise it returns a pathname that is the truename of the file. `probe-file` does not accept wild-cards.

file-create-date *pathname* [*Function*]
file-write-date *pathname* [*Function*]
set-file-create-date *pathname time* [*Function*]
set-file-write-date *pathname time* [*Function*]
`file-create-date` returns the time when the volume, directory, or file specified by *pathname* was created. `file-write-date` returns the time when the volume, directory, or file specified by *pathname* was last modified. The corresponding `set-` functions change these file parameters. The time is given in the universal-time format.

file-author *file–or–dir* [*Function*]
This Common Lisp function—not to be confused with `mac-file-creator`—does not make much sense on the Macintosh. It returns an empty string if *file–or–dir* exists and errors otherwise.

File Operations

The functions below operate on files only. These functions, in conjunction with the `directory` function, provide the needed flexibility for operating on directories.

copy-file *file new-pathname* `&key :overwrite` [*Function*]
file is copied to the pathname specified by merging *new-pathname* with *file*. Arguments may be either strings, lisp pathnames, mac-pathnames or streams. If *new-pathname* does not have a filename component then the *file*'s own filename is used. `:overwrite` has the usual meaning.

`copy-file` returns three values if successful. The first value is the *new-pathname* with the filename component filled in. The second value is the truename of the *file* before it was copied. The third value is the truename of the copied *file*. An error is signalled if the copying operation is not successful.

lock-file *file* [*Function*]
unlock-file *file* [*Function*]
file-locked-p *file* [*Function*]
These functions allow one to manipulate the software lock that prevents modifications to a particular file. `file-locked-p` returns `nil` if the file is not locked.

mac-file-type *file* [*Function*]
mac-file-creator *file* [*Function*]
set-mac-file-type *file os-type* [*Function*]
set-mac-file-creator *file os-type* [*Function*]
Every Macintosh file has two parameters specifying the type of the file and the application which created the file. These parameters are called *os-type*s, and are specified by four character keywords. os-types are case-sensitive, and they may contain spaces.

Files created by Allegro have the creator `:|CCL |`, (i.e. "CCL" followed by a space), and the type `:TEXT` or `:FASL`.

`mac-file-type` and `mac-file-creator` return keywords indicating the type and creator parameters of *file*.

`set-mac-file-type` and `set-mac-file-creator` destructively modify the type or creator of the *file*. *os-type* may be a string of four characters or a four character keyword. The new type or creator is returned as a keyword.

Volume Operations

Volume operations take as argument either an integer (the volume number) or a pathname (the volume component 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.

The functions below will signal an error if the number or pathname given as an argument does not correspond to a mounted volume.

volume-number *volume* [*Function*]
returns the volume reference number of the specified volume. If *volume* is valid volume number, it is simply returned.

eject-disk *volume* [*Function*]
ejects the specified *volume* if possible. It is not possible to eject hard disks. `eject-disk` returns the truename of *volume* if successful, otherwise it signals an error.

ejectedp *volume* [*Function*]
returns `t` if the volume is ejected, `nil` otherwise. `ejectedp` signals an error if the specified *volume* is not mounted. `probe-file` can be used to check whether a volume is mounted.

hfs-volume-p *volume* [*Function*]
returns `t` if *volume* uses HFS (Hierarchical File System) and `nil` if it uses MFS (Macintosh File System). Most current Macintoshes only use HFS devices.

flush-volume *volume* `&key :drive-number` [*Function*]
Some file system manipulations are buffered for execution at a later time. `flush-volume` insures that all buffered file manipulations to a specified volume are performed. *volume* may specify either a name or a drive-number. If given, `:drive-number` takes precedence. `flush-volume` returns the name or the drive number of the volume effected.

Directory Operations

do-files-in-directory (*var pathname* [*resultform*]) {*form*}* [*Macro*]
iterates *form*s with the variable *var* bound to a mac-pathname specifying each file in the directory component of *pathname* .

do-directories-in-directory (*var pathname* [*resultform*]) {*form*}* [*Macro*]
iterates *form*s with the variable *var* bound to a mac-pathname specifying each subdirectory in the directory component of *pathname* .

files-in-directory *pathname* [*Function*]
returns a list of files (as mac-pathnames) in the directory component of *pathname* . Signals an error if there is no directory corresponding to the directory component of *pathname*.

`files-in-directory` is equivalent to

```
(let ((names nil))
  (do-files-in-directory (file pathname (nreverse names))
     (setq names (cons file names))))
```

directories-in-directory *pathname* [*Function*]
returns a list of directories (as mac-pathnames) in the directory component of *pathname*. Signals an error if there is no directory corresponding to the directory component of *pathname*.

`directories-in-directory` is equivalent to

```
(let ((names nil))
  (do-directories-in-directory (file pathname (nreverse names))
     (setq names (cons file names))))
```

devices [*Function*]
returns a list of devices (as mac-pathnames) known to the machine.

User Interface

choose-file-dialog `&key :mac-file-type :mac-file-creator :directory` [*Function*]
displays the standard Macintosh choose-file dialog.

`:mac-file-type` and `:mac-file-creator` should be os-type keywords, or lists of os-type keywords. If specified, only files with the given mac-file-types and mac-file-creator will be displayed in the dialog.

The `:directory` keyword specifies the directory shown when the dialog first appears. It defaults to the last directory shown by the choose-file-dialog or choose-new-file-dialog.

choose-new-file-dialog `&key :directory :prompt` [*Function*]
displays the standard Macintosh choose-new-file dialog.

The `:directory` keyword specifies the directory shown when the dialog first appears. It defaults to the last directory shown by the choose-file-dialog or choose-new-file-dialog.

The filename component of `:directory` is used as the default filename in the editable-text item of the dialog.

The `:prompt` keyword specifies the text to display above the file-name type in area. If supplied, `:prompt` should be a string. The default Prompt is "As...".

Debugging 11

Overview
Fred Commands
Inspect
Step
Backtrace
Trace

Debugging

Overview

Allegro CL provides several tools which help programmers examine and debug programs and environments. These include a window-based inspector, a stepper, a stack-backtrace facility, and the standard Common Lisp `trace` function. Most of these tools can be used without any documentation. Here are a few helpful tips.

Fred Commands

Several Fred commands help the programmer get information.

m-. [*Fred Command*]
Typing meta-period when the cursor is in or next to a symbol causes Allegro CL to search for the source-code associated with the symbol. This will be available if the symbol was defined with `*record-source-files*` non-`nil`. If the source-code is found, it is brought up in a Fred window. If the symbol has been defined from several files, the user is given the choice of files.

c-x c-a [*Fred Command*]
Typing control-x followed by control-a when the cursor is in or next to a symbol causes Allegro CL to display the argument list of the function associated with the symbol. This search may be performed in four ways. If the function is a Common Lisp function, the argument-list information is taken from the cl-database.text file in the ccl-doc folder. If the function was defined with `*record-source-file*` or `*save-definitions*` non-`nil`, Allegro CL extracts the argument list from the source code. As a last resort, limited argument-list information is extracted from the compiled function.

c-x c-d [*Fred Command*]
Typing control-x followed by control-d when the cursor is in or next to a symbol causes Allegro CL to search for the documentation string of the function associated with the symbol. If the search is successful, the documentation string is printed. The search will only succeed if the function was defined with `*save-doc-strings*` non-`nil`.

c-x c-i [*Fred Command*]
inspects the current expression. See the section on Inspect, below, for a description.

c-x c-m [*Fred Command*]
macroexpands the current expression with `macroexpand` and pretty-prints the result into the listener.

c-m [*Fred Command*]
macroexpands the current expression with `macroexpand`. The result of each call to `macroexpand-1` is printed in the listener.

c-x c-r [*Fred Command*]
prints the result of reading the current s-expression. This is useful for tracking down read-time bugs, particularly in expressions containing backquotes.

Inspect

Allegro CL supports the Common Lisp function `inspect` with a window-based inspector. Their are three ways of invoking the inspector.

The first is through the **Inspect** menu-item, on the **Tools** menu. Selecting this item brings up a modeless dialog box with several inspect options, including a help function and a function for inspecting system data. The help function is only available if the ccl-docs folder is in the same directory as Allegro CL.

The second method is through the Fred command control-x control-i. This command inspects the current selection or current Lisp expression. When performed in the listener, if there is no selection or current expression, the value of `*` is passed to inspect.

The third method is through a call to the function `inspect`. For example

```
(inspect *menu*)
```

would bring up an inspector window for the menu class.

The form associated with the top inspector window can be accessed through the function `top-inspect-form`. This allows limited data modification through the inspector.

Step

Allegro CL supports the Common Lisp `step` function with a modal window-based stepper. Two types of functions can be stepped: evaluated functions, and functions which were compiled when `*save-definitions*` was non-`nil`. Functions compiled with `*save-definitions*` set to `nil` need to be redefined (either as evaluated functions, or as compiled functions with `*save-definitions*` non-`nil`) before they can be stepped.

Backtrace

When Allegro CL enters a break-loop, the stack backtrace dialog becomes available. If the break-loop is entered through an error (which will happen if `*break-on-errors*` is non-`nil`), the stack backtrace dialog will appear automatically. In other situations, the stack backtrace dialog may be brought up by choosing the **Backtrace** menu-item from the **Tools** menu.

The stack backtrace dialog consists of two tables. The top table shows the functions awaiting return values on the stack. The name of each function is preceded by the address (in hex) of the function's stack frame. Keep in mind that Allegro CL is properly tail-recursive; functions may be missing from the stack, even though they were a part of the calling sequence that led to the break. (See the Implementation Notes appendix for details on tail recursion and the Allegro CL compiler).

When a function is selected in the upper table of the backtrace dialog, the lexical values held in that function's frame are shown in the lower table. Selecting a value causes it to be assigned to the global variable `*d*`. This variable may then be used in forms evaluated from the listener or other windows. When the backtrace dialog is closed, or when a different frame is selected, the value of `*d*` is set to `nil`.

The Allegro CL backtrace mechanism does not currently show the names of lexical values.

Trace

The trace function is implemented to Common Lisp standards, and has been extended to work with Allegro CL's object system. If a function has separate definitions in several objects, the tracing of each is independent. You simply ask the desired object to trace the function.

Example:

```
? (defobfun add-twice (x y)
    (+ x y x y))
ADD-TWICE
? (defobject foo)
FOO
? (defobfun (add-twice foo) (x y)
    (usual-add-twice (+ x x) (+ y y)))
ADD-TWICE
? (trace add-twice)
NIL
? (add-twice 1 1)
 Calling (ADD-TWICE 1 1)
 ADD-TWICE returned 4
4
? (ask foo (add-twice 1 1))
 Calling (ADD-TWICE 2 2)
 ADD-TWICE returned 8
8
? (untrace add-twice)
NIL
? (ask foo (trace add-twice))
NIL
? (add-twice 1 1)
4
? (ask foo (add-twice 1 1))
 Calling (ADD-TWICE 1 1)
 ADD-TWICE returned 8
8
? (trace add-twice)
NIL
? (ask foo (add-twice 1 1))
 Calling (ADD-TWICE 1 1)
  Calling (ADD-TWICE 2 2)
  ADD-TWICE returned 8
 ADD-TWICE returned 8
8
?
```

Low-level System Interface 12

Overview
Sharing Data Between Allegro CL and the Macintosh Operating System
Lisp Data Representation
Calling Macintosh Traps from Allegro CL
- Stack Traps
- Register Traps
- General Trap Calls

Memory Management
- Stack Blocks
- Accessing Memory
- Strings, Pointers, and Handles

Pascal *var* arguments
DefPascal

Low-level System Interface

Overview

This Chapter describes Allegro CL's most direct, complete, efficient and dangerous level of access to the Macintosh ROM. Because Allegro CL provides higher-level access to the most commonly used parts of the Macintosh ROM, many users will not need to use these low-level features. The higher-level tools are easier to use and safer. If used improperly, the low-level calls can easily make the system crash. They are provided so programmers can use traps which aren't included in the high-level interface, can customize calls to particular traps, and can optimize critical sections of code.

These descriptions assume some familiarity with *Inside Macintosh.*

Sharing Data Between Allegro CL and the Macintosh Operating System

Allegro CL manipulates two distinct sets of data: Macintosh data such as windows, patterns, and regions; and Lisp data such as lists, symbols, and objects. Macintosh and Lisp 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. *Every datum belongs on one heap or the other.*

Some Lisp data contain pointers to Macintosh data, but Macintosh data should *never* point to Lisp data. For example, a window object (a Lisp datum) contains a pointer to a Macintosh window record (on the Macintosh heap). Users should never store a Lisp datum on the Macintosh heap (such as putting the window object into the refCon slot of a window record). This is because Lisp data is movable, and pointers to Lisp data on the Macintosh Heap cannot be updated when data is moved. After a garbage collection, any pointers to Lisp data on the Macintosh heap could easily point to the wrong place and would probably cause an immediate or delayed crash.

In general, Macintosh data is only needed for communication with Macintosh ROM calls. Before Allegro CL can pass data to the ROM, the data must be coerced to a form that the ROM can use. This data cannot be stored on the Lisp Heap, but instead must be stored on the Application Heap or on the stack.

Lisp Data Representation

Every Lisp datum is represented in 32 bits as either an immediate or a pointer to the Lisp heap. Immediate data are completely represented in 32 bits and may be stored directly in a variable, cons cell, etc. There are currently only five types of data that are represented as immediates: fixnums, characters, Macintosh pointers, and aliases for the commonly used symbols `t` and `nil`. All other Lisp data is represented by pointers to the actual data.

Every Lisp datum has an associated type that can be quickly determined from the datum. The type of non-immediate data can be determined by looking at the address of the pointer; the type of immediate data is encoded in the datum itself. In brief:

If bit 31 is set, the datum is a fixnum.
If the datum is all zeros, it is `nil`.
If the datum has only the low five bits set, it is `t`.
If the three high bytes of the datum are all zero except for bit 8, it is a character.

If a datum points to the Lisp heap, then its type is determined by the location in the Lisp heap. Otherwise it is a Macintosh pointer (which must point to real memory).

For fixnums, the low 31 bits are interpreted as a two's complement integer. When 32-bit numeric values are required by the ROM, fixnums need to be sign-extended to the full 32 bits. This operation is called *unboxing*. The reverse operation of converting from 32-bit numeric values to 31-bit fixnums is called *boxing*.

The 31-bit limitation on fixnums can (in rare instances) lead to some trouble when the ROM requires a full-precision 32-bit value. It is usually possible to work around the 31-bit limitation of fixnums by passing two successive 16-bit values (i.e. by using the low 16 bits of two fixnums). Allegro CL also provides functions which move data between a fixnum-or-bignum and a 32-bit long word in memory. However, these functions are not as efficient as other data-sharing functions.

Calling Macintosh Traps from Allegro CL

There are two types of Macintosh traps: those which accept arguments and return values on the stack, and those which accept arguments and return values through registers. The type of a trap can be determined by reading the description of the trap in *Inside Macintosh*. If registers are mentioned, it is a register trap; if registers are not mentioned it is a stack trap. In general, the operating system traps are register-based and the toolbox traps are stack-based, but there are exceptions. Calls to these two types of traps have different formats, as outlined below.

Within a single trap call, some arguments are passed as immediate values and some are passed by reference (i.e. a pointer to the value is passed). In general, data four bytes long or less is passed by value, and data longer than four bytes is passed by reference. Check *Inside Macintosh* for the calling sequences of particular traps. Arguments which are passed by reference may be modified by the trap (these are Pascal *var* parameters).

For each trap `TrapName`, Allegro CL defines a constant `_TrapName` (equal to the trap number) and a macro `_TrapName` which expands into the trap call and is recognized by the compiler. The use of these macros is described in the following sections. In addition, the macros `stack-trap` and `register-trap` are described. These macros can be used for calling any traps, including those which are not predefined by Allegro CL.

Note:
No Lisp data should be passed to Macintosh traps (or any foreign function). Only unboxed immediates or pointers to the Mac Heap should be received by traps. If Lisp data is used, the function will not receive the correct data, but will instead receive a pointer to a data structure in the Lisp Heap or a boxed value that will probably be incorrect. If memory is allocated by the foreign function, a garbage-collection may be triggered which will cause the Lisp data to move without updating the pointer. In other words, the data will be invalid, and it will probably crash the system immediately or even worse, crash later.

Neither the Macintosh operating system, nor the Allegro CL low-level trap interface perform significant type-checking on arguments to traps. Therefore, care should be taken when passing arguments to traps.

Stack Traps

The general format of a stack trap call is:

_Foo {*type-keyword argument*}* [*return-value-keyword*] [*Macro*]

where _Foo is the name of the trap.

The *arguments* are evaluated, coerced according to *type-keyword*, and passed as the arguments to the trap. As noted above, the *arguments* should evaluate to fixnums (integers in the range $\pm 2^{31}$) or to pointers to data on the Macintosh Heap or the stack.

The *type-keywords* signify the type coercion to be performed on the corresponding *argument*. Possible *type-keywords* are :word, :long, and :ptr. The keywords operate on the subsequent *argument* according to the following table.

:word..... *argument* is truncated to 16 bits and passed as a 16-bit word.

:long..... *argument* is sign extended from 31 to 32 bits (unboxed) and passed as a 32-bit long word.

:ptr ..... *argument* is passed unmodified as a 32-bit pointer.

return-value-keyword indicates the type of value which the trap returns. 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 fixnum.

:long..... a 32-bit result is read from the stack, truncated to 31 bits (boxed), and returned as a fixnum.

:ptr ..... a 32-bit result is returned from the stack unmodified as a pointer.

:novalue ..... the Macintosh trap does not return a value. The Lisp call will return nil.

Example:

```
(let ((oval-width 12)
      (oval-height 8))
  (%stack-block ((my-rect 8))
       (%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)))
```

This form will call the trap FrameRoundRect with three arguments, a pointer and two words. nil will be returned.

```
(_GetResource :word #x5223     ;ASCII "R#"
              :word #x5354     ;ASCII "ST"
              :word n
              :ptr)
```

This gets STR# resource number n (see *Inside Macintosh* for a complete description of `_GetResource`). Note that the components of the resource type are pushed in reverse order so that they will be in the correct order on the stack. If the resource is found, it will be loaded into memory and a pointer to it will be returned; if it is not found, `nil` will be returned.

Register Traps

The general format of a register trap call is:

_Foo [`:check-error`] {*register-keyword argument*}* [*return-register-keyword*] [*Macro*]

where `_Foo` is the name of the trap.

If the symbol `:check-error` appears as the first argument to a register trap, then Allegro CL will signal an error if the trap returns with a negative value in register d0. Before using `:check-error`, make sure that the trap in question uses this error-signalling protocol.

The *register-keywords* specify the register which will hold the subsequent *arguments*. The *return–register-keyword* specifies which register will hold the value returned by the trap (if any). Recognized register keywords are `:a0` thru `:a6` and `:d0` thru `:d7`.

The *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 fixnums; these will be sign-extended from 31 to 32 bits (unboxed). Arguments placed in address registers are not coerced in any way, and should usually be handles or pointers to non-Lisp memory.

Example:

```
(_Delay :A0 (%int-to-ptr 60))
```

The value 60 must be explicitly coerced from an integer to a pointer (thereby modifying the high bit) because the trap call will not perform this coercion.

The *return-register-keyword* specifies which register contains the value to be returned from the trap. If *return-register-keyword* is not supplied, then `nil` is returned. Values returned from a data register will be truncated to 31 bits and returned as a fixnum. Values returned from an address register are returned unmodified and will usually be a handle or pointer to non-Lisp memory. There is no facility for returning multiple values from register traps.

Example:

```
(setq my-pointer (_newptr :check-error
                          :d0 2000
                          :a0))
```

upon completion `my-pointer` will hold a pointer to a 2000-byte block of memory on the current Macintosh Heap. If the memory cannot be allocated, a Lisp error will be signalled.

General Trap Calls

Two macros—`stack-trap` and `register-trap`—provide a generalized mechanism for calling Macintosh traps. These functions can be used for calling traps which are not defined by Allegro CL, for example, color Quickdraw traps contained in the Macintosh II.

stack-trap [:check-error] *trap-number* [*Macro*]
{*type-keyword argument*}* [*return-value-keyword*]

register-trap [:check-error] *trap-number* [*Macro*]
{*register-keyword argument*}* [*return-value-keyword*]

trap-number should evaluate 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.

The *type-keyword*s, *return-value-keyword*s, *register-keyword*s, and *argument*s have the same meanings as for predefined trap-calls, as does the keyword `:check-error`.

Example:

```
;;; Setting bit 9 of certain Memory Manager traps causes
;;;  allocated memory to be initialized to zero by the ROM.
;;;  This feature saves us a little code and also exploits
;;;  the fact that, for hardware reasons, code in ROM
;;;  executes about 20% faster than code in RAM on many
;;;  versions of the Macintosh.

(defconstant _NewPtr.CLEAR
             (logior _NewPtr (ash 1 9))
             "_NewPtr with bit 9 set")

(defmacro _NewPtr.CLEAR (&rest args)
  (if (eq (car args) :check-error)
      ;; if present, put it in the right place.
    `(ccl::register-trap :check-error
                         _NewPtr.CLEAR
                         ,@(cdr args))
    `(ccl::register-trap _NewPtr.CLEAR
                         ,@args)))


;;; test it :

(defun test-it ()
  (let ((p (_NewPtr.CLEAR :check-error :d0 20 :a0)))
    (unwind-protect
      (dotimes (i (_GetPtrSize :check-error :a0 p :d0) t)
        (unless (zerop (%get-byte p i))
          (error
           "_NewPtr.CLEAR didn't work as advertised.")))
      (_DisposPtr :check-error :a0 p))))

(test-it)
```

Memory Management

Allegro CL works cooperatively 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 will sometimes trigger a garbage collection.

You are responsible for releasing memory allocated on the Macintosh heap (this can be done with _disposptr and _disposhandle. It will not be garbage collected even if all pointers to the memory are lost. Also, care should be taken so that once memory is released, nothing in Lisp still refers to it. This could cause a crash if that memory location is later used for Lisp data and a garbage collection occurs.

`_NewPtr` and `_NewHandle` are automatically called by `make-record`, described in the chapter Pascal Records.

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` special form 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, so any remaining pointers into the stack block will be invalid. `%stack-block` should only be used for well-defined temporary storage, for example in setting up rectangles or I/O parameter blocks to be passed to traps, or as a temporary place to put *var* arguments.

`%stack-block` ({(*variable size*)}$^+$) {*form*}$^+$ [*Special Form*]1

for each *variable* and *size*, `%stack-block` allocates a block of storage *size* bytes long, and binds *variable* to a pointer to the block. If there is not enough room on the stack to allocate the requested memory, an error is signalled. The forms are executed in the resulting environment. The maximum total size for an individual stack block form is 32k bytes. Every *size* must be a positive fixnum, or a compile-time error will be signalled. `%stack-block` is semantically equivalent to doing a `_NewPtr`/`_DisposPtr` pair for each *variable*, but much more efficient.

The bindings and the storage created by `%stack-block` have lexical scope and dynamic extent. They do not have indefinite extent. They disappear when the `%stack-block` exits. The variables cannot be accessed outside of the `%stack-block` form and they cannot be declared as special variables. They cannot be closed over.

Example:

```
(let ((oval-width 12)
      (oval-height 8))
(%stack-block ((my-rect 8))
       (%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)))
```

An eight-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 trap `_FrameRoundRect`.

Accessing Memory

Once memory for a structure has been allocated, programs need methods for directly reading from and writing to the memory. Allegro CL provides the following low-level functions for reading and writing to memory locations. (The functions `rref` and `rset` may be used. See the chapter Pascal Records for details.) Most calls to these functions are compiled inline for efficiency.

Each of the following functions takes an *offset* as an optional argument. No type-checking is performed on the *offset*. It is treated as a fixnum (i.e. an immediate datum), and sign-extended from 31 to 32 bits (unboxed).

%get-byte *pointer* `&optional` *offset* [*Function*]
gets the byte (8 bits) at *pointer* + *offset* and returns it as a fixnum.

%get-signed-byte *pointer* `&optional` *offset* [*Function*]
gets the byte (8 bits) at *pointer* + *offset*, sign-extends it, and returns it as a fixnum.

%get-word *pointer* `&optional` *offset* [*Function*]
gets the word (16 bits) at *pointer* + *offset* and returns it as a fixnum. *Note that on a 68000 (but not a 68020) a fatal error will occur when a word is accessed at an odd memory address.*

%get-signed-word *pointer* `&optional` *offset* [*Function*]
gets the word (16 bits) at *pointer* + *offset*, sign-extends it, and returns it as a fixnum. *Note that on a 68000 (but not a 68020) a fatal error will occur when a word is accessed at an odd memory address.*

%get-long *pointer* `&optional` *offset* [*Function*]
gets the long-word (32-bits) at *pointer* + *offset* , truncates it to 31 bits (by removing the high bit) and returns it as a fixnum. *Note that on a 68000 (but not a 68020) a fatal error will occur when a long-word is accessed at an odd memory address.*

%get-ptr *pointer* `&optional` *offset* [*Function*]
gets the long-word (32-bits) at *pointer* + *offset* and returns it unmodified. The data should be a valid pointer or the garbage collector will be confused by it and probably crash. *Note that on a 68000 (but not a 68020) a fatal error will occur when a long-word is accessed at an odd memory address.*

%get-safe-ptr *pointer* `&optional` *offset* [*Function*]
gets the long-word (32-bits) at *pointer* + *offset*, checks to see if it is a pointer to valid memory, and returns it if it is, otherwise returns `nil`. *Note that on a 68000 (but not a 68020) a fatal error will occur when a long-word is accessed at an odd memory address.*

%get-string *pointer* `&optional` *offset* [*Function*]
gets the Pascal string at *pointer* + *offset* and returns it as a Lisp string. (Pascal strings are formatted differently from Lisp strings. The Macintosh Toolbox uses Pascal strings.)

%get-ostype *pointer* `&optional` *offset* [*Function*]
gets the four bytes at *pointer* + *offset* and returns it as a keyword of four characters.

%put-byte *pointer data* `&optional` *offset* [*Function*]
stores the low eight bits of *data* at *pointer* + *offset*.

%put-word *pointer data* `&optional` *offset* [*Function*]
stores the low 16 bits of *data* at *pointer* + *offset*. *Note that on a 68000 (but not a 68020) a fatal error will occur when a word is stored into an odd memory address.*

%put-long *pointer data* `&optional` *offset* [*Function*]
sign-extends *data* from 31 to 32 bits and stores the result at *pointer* + *offset*. *Note that on a 68000 (but not a 68020) a fatal error will occur when a long-word is stored into an odd memory address.*

%put-full-long *pointer data* `&optional` *offset* [*Function*]
stores the integer value of data at *pointer* + *offset*. *data* should be an integer (fixnum or bignum). This operation is less efficient than `%put-long`, but it allows full 32-bit accuracy. *Note that on a 68000 (but not a 68020) a fatal error will occur when a long-word is stored into an odd memory address.*

%put-ptr *pointer data* `&optional` *offset* [*Function*]
stores *data* at *pointer* + *offset*. The full 32-bits of data will be written unmodified, so care should be taken to not put Lisp data into non-Lisp memory space and not put Macintosh data into Lisp memory space. *Note that on a 68000 (but not a 68020) a fatal error will occur when a long-word is stored into an odd memory address.*

%put-string *pointer string* `&optional` *offset* [*Function*]
stores *string* as a Pascal string starting at *pointer*+ *offset*. *string* should have a maximum length of 255.

%put-ostype *pointer string* `&optional` *offset* [*Function*]
string should have a length of four characters. `%put-ostype` stores *string* as four bytes at *pointer* + *offset*. *Note that on a 68000 (but not a 68020) a fatal error will occur when an os-type is stored into an odd memory address.*

Strings, Pointers, and Handles

The following utilities are provided for using strings, pointers, and handles.

with-pstrs ({(*variable string*)}+) {*form*}+ [*Macro*]
with-returned-pstrs ({(*variable string*)}+) {*form*}+ [*Macro*]
Strings are passed from Allegro CL to the Macintosh operating system with such frequency that Allegro CL provides a macro, `with-pstrs`, to expedite the process. `with-pstrs` saves the trouble of allocating memory and filling it with individual characters every time a Macintosh accessible string is needed.
Memory for each *string* is allocated, the *string* is stored in this memory in the Pascal string format, and the corresponding *variable* is bound to a pointer to the memory. Each *string* can have a maximum length of 255 characters. The *forms* are evaluated in the resulting environment.

Traps which use the strings as *var* arguments for returning values should be called through the macro `with-returned-pstrs`. `with-returned-pstrs` allocates 256 bytes for each string (rather than trying to optimize the amount of memory allocated). This guarantees that the returned string will not overwrite other segments of memory.

`pointerp` *thing* [*Function*]
returns `t` is *thing* is a pointer to non-Lisp memory, `nil` otherwise. A datum is considered a pointer if it is not a fixnum and it points anywhere besides the Lisp heap.

`zone-pointerp` *thing* [*Function*]
returns `t` if *thing* is a pointer to a non-relocatable system or application HeapZone memory block, otherwise `nil`. The test is performed heuristically by first determining if *thing* points into a Macintosh HeapZone, and if so, seeing if the long word before the address pointed at by *thing* is equal to the HeapZone. A zone pointer is different from a generic pointer since it points to a memory block that was allocated using `_NewPtr`, and the various memory manager pointer traps such as `_DisposPtr` may be used on it. See the Memory Manager chapter in *Inside Macintosh* for more information on the structure of zone pointers.

`handlep` *thing* [*Function*]
returns `t` is *thing* is a Macintosh handle, otherwise `nil`. This is determined heuristically by first checking if *thing* points into the system or application HeapZone, and if so, indirecting through *thing* to see if the long word prior to the address plus the Zone pointer is equal to *thing*. See the Memory Manager chapter in *Inside Macintosh* for more information on the structure of handles.

`with-dereferenced-handles` ({(*variable handle*)}$^+$) {*form*}$^+$ [*Macro*]
executes *forms* with each *variable* bound to the locked, dereferenced *handle*. Only previously unlocked handles are locked. Upon exit, only handles which were unlocked on entry are unlocked. (This prevents an annoying and difficult to find bug that occurs when programming the Macintosh with other languages). Unlocking of handles is unwind-protected, so the handles are guaranteed to be left in the same state before and after the `with-dereferenced-handles`, even if termination is abnormal.

`with-pointers` ({(*variable pointer-or-handle*)}$^+$) {*form*}$^+$ [*Macro*]
for each *pointer-or-handle* which is a pointer, binds the *variable* to the pointer; for each *pointer-or-handle* which is a handle, binds the *variable* to the locked, dereferenced handle, as with `with-dereferenced-handle`. `with-pointers` is useful if you are unsure whether *pointer-or-handle* will be a pointer or a handle. An error is signalled if *pointer-or-handle* is neither a pointer nor a handle.

`%inc-pointer` *pointer number* [*Function*]
increments (or decrements) *pointer* by adding it to *number*, and returns a new pointer. *pointer* should be a Mac pointer. *number* should be a fixnum.

`%ptr-to-int` *pointer* [*Function*]
returns a fixnum coerced from *pointer*, that is, the numerical address *pointer* points to. This involves setting the high-bit.

`%int-to-ptr` *fixnum* [*Function*]
returns a pointer coerced from *fixnum*, that is, a pointer to the numerical address *fixnum*. This involves unboxing *fixnum*, i.e., setting bit-31 to the same value as bit-30.

Pascal *var* arguments

Pascal *var* arguments are passed by reference, rather than by value; i.e., you pass the function a pointer to a datum, rather than the datum itself. The called function may then side-effect the datum and in this way communicate information to the caller. Implementing *var* arguments in Allegro CL is very easy. Just allocate memory of the appropriate size for the datum (either on the stack or on the Macintosh heap, depending on how long you want to use the datum) and pass a pointer to the memory as the *var* argument.

DefPascal

The defpascal macro allows programs to create Lisp procedures which can be called by the Macintosh toolbox. Defpascal has the general form

defpascal *name* ({(*argument type*)}* (:result *return-type*)) [*Macro*]
[*doc-string*] {*form*}*
defpascal *name* ((*argument* :reg)) [*Macro*]
[*doc-string*] {*form*}*

The syntax is similar to defun, except that the lambda list contains pairs rather than arguments, and ends with a type specifier for the value returned by the procedure. Each pair is the name of the argument followed by the argument's type. &optional, &keyword, &rest, and &aux arguments are not permitted because they are not supported by the Pascal calling sequence.

In the first format, arguments may be of type :word, :long, :ptr. :word's and :long's will be passed to the procedure as fixnums, and :ptr's will be passed to the procedure as pointers. *return-type* may be :word, :long, :ptr or :void. :void is used when the the procedure does not return a value. (Omitting *return-type* is equivalent to a return-type of :void.)

In the second format, only a single *argument* is given. This should be a pointer to a Pascal record of type regbuf. When the function is called, the values of the machine registers are copied into the record. The values of the record can be set and accessed with rref and rset (see the chapter Pascal Records for details). The value returned from the function is the pointer to the record.

name will be set to a pointer to the procedure. This pointer may be passed to traps.

The following procedure could be passed to _TrackControl and used to track scroll bars:

```
(defpascal my-track-proc ((my-control PPTR) (partCode PWORD))
 (_SetCtlValue :ptr my-control :word
   (+ (_GetCtlValue :ptr my-control :word)
    (case partCode
       (20 -1)                  ;scroll back one line
       (21 1)                   ;scroll forward on line
       (22 (- *page-height*))     ;scroll back on page
       (23 *page-height*)))))     ;scroll forward one page
```

It could be used as follows:

```
(if (logbitp 8
      (_TrackControl :ptr the-control :long mouse-point
                     :ptr my-track-proc :word))
    (update-window))
```

Pascal Records 13

Overview
The Structure of Records
Record Functions

Pascal Records

Overview

The functions and macros described below allow Coral Lisp programs to manipulate data using Pascal record formats. This is useful because the Macintosh Operating System usually stores data as Pascal records. Data structures created at run-time (such as windows and text-edit records), as well as Macintosh resources may be conveniently accessed and altered using these functions.

There are some caveats to sharing data between CCL and the Macintosh operating system. These are described in the first three sections of the chapter Low Level System Interface. You should be familiar with the information from those sections before reading the rest of this chapter.

Records may be thought of in two ways, how they are stored and used, and how they are passed around by Lisp. As stored and used, a record is a contiguous, highly formatted block of memory of a specific size, on the stack or Macintosh heap. As passed around, a record is a simple pointer into memory, with no formatting or length information. To use a record (which Lisp sees as a mac-pointer), a program must provide a record-type, telling the system how the data pointed at should be interpreted. It is up to your program to remember the types of all the records you create. This system allows you to map over a single block of memory in several different ways, as if it were several different types of record. It also allows you to take pointers returned by Macintosh traps and use them as records.

In the discussion below, the word *record* can mean either a block of memory, or a pointer into memory, depending upon 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. Below we would simply say a record is allocated and returned.

The Structure of Records

A *record* is a contiguous block of memory on the Macintosh heap. 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 symbolically access portions of a record and field types are used to determine the size of each field and how the information in the field should be encoded/decoded (for example, a field may itself be a record and therefore contain subfields).

The same portion of a record may be accessed in different ways by using *variant fields*. Variant fields allow an area of a record to be mapped to different sets of fields. For example, variant fields could be used to access one part of a record as a single long word or as four bytes. Variant fields (like records in general) are useful mnemonic aids and shortcuts. The size of a variant field is equal to the largest total size of one of the sets of fields in the variant portion of a record.

New record types are defined with `defrecord`. `record-type-p` can be used to determine if something is a record type. `record-length` and `record-storage`, `record-default`, and `record-fields` return information about a record type. `describe-record-field` returns information about a specific field in a type of record. Records may be created with `make-record`. Records allocated by `make-record` will remain until explicitly de-allocated using `dispose-record`. Records may be temporarily allocated (on the stack) with `rlet`. They can be copied using `copy-record`. Fields within records are accessed with `rref` and set with `rset`. `set-record` allows setting multiple fields in a record using keywords to specify fields. `record-string` may be used to get a printed representation of a record.

Record Functions

defrecord *name-and-storage* [*doc-string*] {*field-description*}+ [*Macro*]
`defrecord` is used to define new record types. *name-and-storage* should be 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 type of record and whose `cadr` is one of the keywords `:pointer`, `:handle`, or `:either` which specifies the default type of storage used for the record type. `:pointer` means that unless otherwise specified, records of that type will be assumed to be pointers, `:handle` means that records of that type are assumed to be Macintosh handles and `:either` means no assumptions are made; the record is examined at run time to determine whether it is a pointer or a handle (this is flexible but inefficient). If no storage type is specified, `:pointer` is assumed.

Records stored as handles are less likely to cause fragmentation of the Macintosh heap, but they cannot be allocated using `rlet`. In addition, the Macintosh ROM is very strict about whether it gets records passed to it by handle or by pointer. The default storage type for a record can be overridden for many record instructions (such as `rref` and `rset`) by specifying the storage type with the `:storage` keyword. *Be very careful when overriding the default record storage. A crash is very likely if a handle is used as a pointer or vice versa.*

In addition to defining a record type, `defrecord` creates a record to be used as a default value whenever that record type is used as a field in another record type. This default record uses the default values of all of the types that it is composed of.

A standard *field-description* has the form:

(*field-name type* `&optional` *default*)

Field-name is the name which will be used to access the field in the record. *type* is the type of data which will be held in the field. It is used to determine the field's length. *type* must be one of the pre-defined types (see table below), a previously defined record type, or a list whose `car` is the symbol `string` and whose `cadr` is a fixnum from `1` to `255`, that is used to specify the space to be allocated for the string.

The pre-defined field types, lengths, and defaults are :

type	length	default
`boolean`	1 bytes	`nil`
`byte`	1 byte	0
`character`	1 byte	`#\null`
`handle`	4 bytes	`nil`
`integer`	2 bytes	0
`longint`	4 bytes	0
`ostype`	4 bytes	`"????"`
`point`	4 bytes	`#@(0 0)`
`pointer`	4 bytes	`nil`

Fields that are two or more bytes will always begin at word boundaries (i.e. at even memory locations). This means that fields that are one byte long will sometimes be padded out to two bytes, so that the next field can begin at an even address.

The following standard Macintosh record types are pre-defined by Allegro CL. Their definitions can be found in the file Library:records.lisp or through the `record-types` option of the inspector.

```
bitmap
bits16
cursor
dialog
dialog-item
event
fontinfo
grafport
pattern
penstate
rect
regbuf
sfreply
window
```

Example:

```
(defrecord PenState  (pnLoc  point)
                     (pnSize point)
                     (pnMode integer)
                     (pnPat  pattern))
```

This call creates a new record type called `PenState` with four fields.

If a field-description includes a *default*, the corresponding field will be initialized to *default* when a new record is created, otherwise the default from the table above will be used.

A field description may contain variant fields, in which case it will have the form

(`variant` ({*field-description1*}$^+$)
({*field-description2*}$^+$)
...
({*field-descriptionN*}$^+$))

The part 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.

Example:

```
(defrecord Rect
    (variant ((top Integer 0) (left Integer 0))
             ((topleft point)))
    (variant ((bottom integer 0) (right integer 0))
             ((bottomright point))))
```

A `rect` record may be accessed either as two points or as four coordinates. All the fields of new records will have a default initial value of zero. When a record is created, only the default values of the first of any set of variants are used.

The following functions take a record-type as an argument. They do not take an actual record, but only a type, or template, which has been created with `defrecord`.

`record-type-p` *record-type* [*Function*]
returns `t` if *record-type* is a defined type of record, `nil` otherwise.

`record-length` *record-type* [*Function*]
returns the length, in bytes, of the record type specified by *record-type*. Signals an error if there is no record type *record-type* .

`record-storage` *record-type* [*Function*]
returns the keyword that specifies the default storage for records of type *record-type*. This will be one of `:pointer`, `:handle`, or `:either`. Signals an error if there is no record type *record-type* .

`record-default` *record-type* [*Function*]
returns the record of type *record-type* that is used as a default whenever a record is created that includes fields that have type *record-type*. The fields of the default record may be changed. Signals an error if there is no record type *record-type*.

`record-fields` *record-type* [*Function*]
returns a list of the names of all the fields of *record-type* including the names of variant fields. Signals an error if there is no record type *record-type*.

`field-info` *record-type field-name* [*Function*]
returns three values that describe the field of *record-type* named by *field-name* The values are the byte offset of the field within the record, the field-type, and the default value of the field. Signals an error if there is no record type *record-type* or no field within *record-type* with name *field-name*.

`record-info` *record-type* `&optional (`*errorp* `nil)` [*Function*]
returns a list of descriptions of all the fields of record-type including variant fields. Each field descriptor is a list of four values: the field name, type, offset and default value. If there is no record type *record-type* and *errorp* is non-`nil` an error is signalled; if *errorp* is `nil`, `nil` is returned.

`make-record` *record-type* `&key :storage` {*field-name contents*}* [*Function*]
Allocates space on the Macintosh heap for a record of type *record-type* and returns a pointer or a handle to it. If `:storage` is not specified, it will be a pointer if the default storage for record-type is `:pointer`, otherwise it will be a handle. *Be very careful when overriding the default record storage. A crash is very likely if a handle is used as a pointer or vice versa.*

The fields of the record may be initialized by using the *fieldnames* as keywords. Any fields not initialized by keywords will be initialized to their default values for the record type. The memory used by the new record will not be automatically garbage-collected; it must be disposed of with `dispose-record`.

Some records, such as windows, must be created and initialized by specific toolbox routines. Such records should not be created with `make-record` but with the appropriate toolbox traps. Their fields can, of course, still be accessed with `rref` and `rset`.

`set-record` *record record-type* `&key` {*field-name value*}* [*Function*]
Sets multiple fields of *record* at one time. Each *field-name* should be a keyword that names the field of *record-type* that will be set to *value* in *record*. The storage type of *record* is determined at the time of the call, so any storage type may be used for any record type. *record* is returned.

`copy-record` *source-record record-type* `&optional` *dest-record* [*Function*]
copies all of the fields of *source-record* which should be of type *record-type* into *dest-record*. If *dest-record* is not supplied, a new one will be allocated that has the same storage type as *source-record*. Returns *dest-record*.

`dispose-record` *record* [*Function*]
disposes of the given *record*. If *record* contains pointers to other records, these other records will not be automatically disposed by `dispose-record` since other pointers to those records may exist. `dispose-record` only has an effect if *record* is a pointer or a handle to a Macintosh Memory Manager block. `nil` is returned.

`print-record` *record record-type* `&optional` *print-level* [*Function*]
prints the values of the fields of *record* with type *record–type*. This pays attention to the values of `*print-length*` and `*print-level*`. *print-level* is the current print nesting, it defaults to zero and is normally used only internally. No values are returned.

`get-record-field` *record record-type field-name* [*Function*]
returns the value of the *field-name* field of *record* which should be of type *record-type*. The macro `rref` should normally be used to get the values since it is more efficient. `get-record-field` is provided so that there is a functional way to get the values of fields.

`set-record-field` *record record-type field-name value* [*Function*]
sets the value of the *field-name* field of *record* which should be of type *record-type* to *value* and returns `nil`. The macro `rset` should normally be used to set the values of fields since it is more efficient. `set-record-field` is provided so that there is a functional way to change the values of fields.

`rref` *record accessor* `&key :storage` [*Macro*]
is used for accessing the fields of a record. *record* is a pointer or handle to a record. *accessor* tells `rref` what record and definition to apply and what field to access, and `:storage` tells `rref` what kind of storage to assume for the record. It should be `:pointer`, `:handle`, or `:either`. `:storage` is not usually specified, being taken instead from the record-type of *accessor*.

accessor has the form:

record-type{.*field*}$^{+}$

where the *record-type* must be a previously defined record type, and *field* must be a field in a record of type *record-type* (with extensions if *record-type*'s first field is a record, see below). 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* as long as all but the last is a record type.

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 `window` record, so the grafport's `portrect` could be referred to as `window.portrect` as an abbreviation for `window.grafport.portrect`.

If `:storage` is not supplied, then the default storage type for *record-type* is used. If it is supplied it should be `:pointer`, `:handle`, or `:either`. *Be very careful when overriding the default record storage. An immediate or delayed crash is very likely if a handle is used as a pointer or vice versa.*

If the final *.field* of *accessor* is 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. An error is signalled at macro-expansion time if an attempt is made to get a pointer to a record within a handle, since the memory block pointed to by the handle may move. If that is desired, use a `with-dereferenced-handle` form around the call and specify `:storage` to be `:pointer`. Warning: such a pointer to a record within a handle will be valid only for the duration of the `with-dereferenced-handle`.

`rref` is very efficient. It macro-expands into a simple call to a low-level memory accessing function (unless `:storage` is `:either`) which is in turn compiled inline. For the curious, try experimenting with how `rref` expands to see how it works.

Examples:

```
(rref my-rect rect.top)
(rref wptr window.portrect.bottomright)
(rref tePtr terec.viewrect.left :storage :pointer)
(rref my-control control.contrlvalue)
```

rset *record accessor value* `&key :storage` [*Macro*]
is used to change the values of the fields of a record. *record* is a pointer or handle to a record, *accessor* has the same format as the accessor for `rref`, and *value* is the new value to store into 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.*

If `:storage` is not supplied, then the default storage type for *record-type* is used. If it is supplied it should be `:pointer`, `:handle`, or `:either`. *Be very careful when overriding the default record storage. An immediate or delayed crash is very likely if a handle is used as a pointer or vice versa.*

`rset` is very efficient. It macro-expands into a simple call to a low-level memory accessing function (unless `:storage` is `:either`) which is in turn compiled inline. For the curious, try experimenting with how `rset` expands to see how it works.

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.contrlvalue 200)
```

rlet ({(*var record-type* {*field-keyword value*}*)}+) {*form*}+ [*Macro*]
is used to efficiently create temporary records. `rlet` has the same general form as `let`. For every *var* a new binding is created. Space is allocated on the stack for a *record-type* record, the record is initialized according to the *field-keywords* and *values* as in `make-record`. A pointer to the new record is stored in the corresponding *var*. The *forms* are evaluated in the resulting environment, and the value of the final form is returned.

Note that `rlet` differs from `make-record` in that fields not explicitly initialized by a *field–keyword/value* pair will contain garbage instead of a default value. This is so that `rlet` will be as efficient as possible in cases where you do not care what is in the fields, such as when a record is to be filled in by a subsequent function call.

`rlet` cannot create records that are accessed through handles. Also, the records are stored on the stack and are therefore ephemeral. When the evaluation of *forms* is done, all the records vanish irretrievably, so make sure you do not have any dangling pointers when an `rlet` terminates.

Appendix A: Implementation Notes

Overview
Reader Macro Characters
Numbers
Characters
Arrays
Packages
Modules
Memory Management
Function Swapping
Garbage Collection
Evaluator
Compiler
- Error Messages
- Constants
- Tail-recursion
- Compiler Declarations
- File Compiler

Appendix A
Implementation Notes

Overview

This chapter describes the particulars of Allegro CL's Common Lisp implementation. It includes information on cases which are ambiguous in Common Lisp and technical information on memory management, the compiler, and other aspects of the Allegro CL system.

The information in this chapter is complemented by the information in the chapter Low -level System Interface.

Reader Macro Characters

In addition to supporting the standard Common Lisp reader macro characters, Allegro CL adds definitions for the following characters, which are undefined by Common Lisp.

`#@`..... transforms the subsequent list of two fixnums, into a point
`#p`..... causes the following string to be read as a mac-pathname

Numbers

• Fixnums are stored as immediate data using a two's complement representation. Fixnums are 31 bits long (the thirty-second bit is used as a type marker). `eql` fixnums are `eq` (although portable code should not rely on this fact).

• There is only one float type, corresponding to IEEE double floats. Each float is 64 bits (8 bytes), consisting of a sign bit, an 11-bit binary exponent (allowing exponents in the range -1022:1023 inclusive), and a 52-bit significand. The significand precision is 53 bits. If the Macintosh contains the MC68881 floating point processor, Allegro CL uses the processor for all floating point computations. Otherwise, Apple's SANE package is used. If Allegro CL is using the MC68881, the symbol `:MC68881` will be present on the `*features*` list.

Characters

Characters are represented as immediate data. The character code is the 8-bit extended ascii value. There are no bits or font information—the char-code and the char-int of a character are the same. `eql` characters are `eq` (although portable code should not rely on this fact). Strings may contain any character (i.e. any character is a string-char).

The characters with ascii codes of 0 through 31 inclusive may use the reader macros #\^@ through #\^_. More generally, #\^c, for any character c, is equivalent to (code-char (logxor 64 (char-code #\c))). In addition #\nnn, where nnn are two or three octal digits, is equivalent to (code-char #onnn).

The variable `CCL::*name-char-alist*` contains names for special characters. It is used by the `char-name` and `name-char` functions. In addition to supporting the standard ascii character set, Allegro CL supports the Macintosh optional character set.

Arrays

The following distinct array element types are supported:

```
bit.....1 bit per element
character.....8 bits per element
(signed-byte 8).....8 bits per element
(signed-byte 16)...16 bits per element
T.....One node (32 bits) per element
```

In addition, Allegro CL supports the special non-CL array element type `CCL::short`, which is similar `(signed-byte 16)` except that setting an array element to a fixnum which is not in the `(signed-byte 16)` range truncates the value to 16 bits rather than causing an error. This is useful in certain system programming applications.

Only simple vectors are supported. All arrays of rank other than 1 are implemented as displaced arrays. In addition to the memory required to store the elements, a simple vector has 8 bytes of overhead (9 bytes in the case of bit vectors). A complex (displaced) array has about 32+(4*rank) bytes of overhead. The rank of an array must be less than #x2000.

The theoretical limits on the sizes of arrays are as follows:

```
bit.....#x7FFFFF1
character.....#x1000000
(signed-byte 8).....#x1000000
(signed-byte 16)...#x800000
T.....#x400000
```

All these limits represent arrays requiring approximately 16 megabytes of contiguous memory.

There is no a priori limit on the size of individual dimensions of an array aside from that imposed by the array total size limit.

Packages

The only external symbols of the `LISP` package are the 775 symbols of Common Lisp. The `CCL` package uses the `LISP` package. Its exported symbols consist of extensions to Common Lisp provided by Allegro CL. The `CCL` package shadows none of the Common Lisp symbols. The `USER` package uses both the `CCL` and the `LISP` packages.

Modules:

There are three ways you can tell the `require` function how to look for a module.

If the second argument to `require` is given, it should be a pathname or a list of pathnames which make up the module.

If the second argument to `require` is not provided, `require` first looks in the variable `*module-file-alist*`, which is bound to an alist. In this alist, 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. `require` will load all the files listed. Initially, `*module-file-alist*` is empty.

Example:

```
(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*`. `*module-search-path*` should be bound to a list of pathnames, each specifying the directory and possibly a file type (the name component is ignored, and is replaced by the name of the module). If no file type is given, both "lisp" and "fasl" are looked for, and the more recent file is used.

Example:

```
(push "misc;" *module-search-path*)
```

will cause `(require 'tools)` to look for the file "misc;tools.fasl" or "misc;tools.lisp".

```
(setq *module-search-path*
      (list* "misc;.fasl"
             *module-search-path*))
```

will cause `(require 'tools)` to look for "misc;tools.fasl" before searching for other versions of the tools file.

The initial value of `*module-search-path*` is `'("home;" "ccl;" "library;")`.

Memory Management

Allegro CL can access up to 8-megabytes of RAM (the current maximum supported by the Macintosh operating system). Configuration for different memory sizes occurs automatically at launch time.

The Macintosh divides memory into three areas: the system heap (used by the Macintosh operating system), the application heap (used to store program data), and the stack. Running under the Switcher, there may be several application heaps, but only one is active at a given time. Allegro CL adds two additional memory areas: a Lisp heap, and a control stack. For efficiency and flexibility, Lisp data is stored on the Lisp heap rather than on the application heap. The Lisp heap and application heap are dynamically resized in response to requests for memory. See the chapter Low-level System Interface for further details.

Allegro CL uses a BiBOP (Big Bag Of Pages) memory and type management system. Pages are 4 k-bytes each.

Function Swapping

A large number (over 2600) of the compiled functions in the Allegro CL kernel participate in a "load-on-call" memory management system. These functions are not loaded into memory until they are first called. When garbage collection occurs, any of these functions which are not active (i.e. not awaiting return values on the stack), are purged. If called again, they will have to be

reloaded. Depending on disk speed, there may be a slight pause when functions are loaded into memory.

The total size of all swappable functions is somewhat greater than 600K bytes.

The load-on-call system is biased towards memory conservation, speed sometimes being sacrificed. Users with sufficient memory who wish to optimize speed may use the following functions to affect the swapping mechanism.

Functions written by the user do not participate in the load-on-call system.

purge-functions `&optional` *boolean* [*Function*]
globally enables or disables purging of swappable functions. If *boolean* is non-`nil`, purging will be enabled. If *boolean* is `nil`, purging will be disabled. If *boolean* is not specified, the function has no side-effect.

`purge-functions` returns `t` is purging is enabled, otherwise `nil`.

`purge-functions` allows the user to disable function-purging before executing programs in which purging and re-loading is undesirable. In general, purging should be disabled if Allegro CL has access to more than 1 megabyte of RAM.

In environments with restricted memory, this feature should be used with caution: there is no way of changing your mind in the middle of a garbage collection. If functions cannot be purged, an out-of-memory error may occur.

preload-all-functions [*Function*]
loads all swappable functions into memory. Functions will still be purged during garbage collection, unless the appropriate call to `purge-functions` has been made.

`preload-all-functions` is primarily intended for use in conjunction with `purge-functions`. The following code could be placed in an init file to completely remove swapping.

```
(progn
  (preload-all-functions)
  (purge-functions NIL))
```

This is recommended for configurations with greater than 1.5 megabytes of RAM.

Garbage Collection

Allegro CL uses a mark/compact/forward garbage-collector. Garbage collection occurs automatically, as memory is needed. This can happen when the Macintosh operating system or Allegro CL need memory. Garbage collection can be invoked manually through the function `gc`.

Evaluator

Allegro CL offers two evaluation options: a standard evaluator, and a compiling evaluator. The standard evaluator conforms to Common Lisp standards and supports `evalhook` and `applyhook`. The compiling evaluator compiles non-trivial expressions and then runs them. For

looping or self-recursive constructs the compiling evaluator is much faster. However, the compiling evaluator does not support `*evalhook*` and `*applyhook*`. When the variable `*fast-eval*` is non-`nil` the compiling evaluator is used. In the default environment, `*fast-eval*` is `nil`.

When `*compile-definitions*` is non-`nil`, the standard evaluator will attempt to compile definitions, thereby creating compiled, rather than evaluated, functions. This attempt will fail if there are lexical bindings in the evaluation environment. For example,

```
(eval '(defun foo (n) (+ n n)))
```

will produce a compiled function if `*compile-definitions*` is non-`nil`. However,

```
(eval '(let ((x 5)) (defun foo (n) (+ n n))))
```

will not produce a compiled definition, regardless of the value of `*compile-definitions*`. (It will, however, produce a compiled definition if the compiling evaluator is used).

In the default mode, `*compile-definitions*` is `t`.

Compiler

Unless otherwise directed, the compiler attempts to produce compact, correct, safe, and reasonably fast code. With very few exceptions, system functions do run-time type checking of their arguments and signal errors when the number or types of arguments are incorrect (the notable exceptions are the trap interface functions and the low-level memory access primitives). Compiled functions normally check for stack overflow conditions and arrange to do periodic event processing. The overhead associated with maintaining this safe run-time environment is intended to be very low.

The vast majority of calls to built-in kernel functions—and all calls to user functions—are compiled out-of-line. The remaining set, which consists primarily of common, computationally inexpensive Lisp primitives such as `cons`, `eq`, and `cdr`, are implicitly declared to be `inline` and typically compile into calls to hand-coded subprimitives in the Lisp kernel. In addition, the accessor functions defined by `defstruct` are, by default, proclaimed to be `inline` and typically compile to code which does appropriate type and bounds checking with minimal function-call overhead. Appropriate `notinline` declarations can be used to override this behavior if, for example, you wish to trace calls to `defstruct` functions.

Lexical closure objects produced by the compiler are always full (upward) funargs.

Run-time processing of `&key` arguments never involves consing (unless, of course, an `&rest` argument is involved).

Error Messages

Error messages produced by the compiler are intended to be self-explanatory. Whenever possible they contain the name of the function being compiled. One source of confusion arises from the fact that many side-effect-free expressions with self-evaluating arguments are evaluated at compile-time. When this occurs, errors may be signalled by the evaluation of the expression, rather than by the compiler itself.

Example:

```
(defun silly (x y)
   (- x (+ y (+ 2 3 'z))))
```

The compiler recognizes that 2, 3, and 'z are all constants, and so it attempts to evaluate the expression (so that it can replace the expression with its result). Compiling `silly` will cause an error to be signalled by + at compile-time, because the symbol z is not numeric. It will not mention that the error occurred while compiling `silly`.

Constants

The compiler folds `equal` constant expressions. As a consequence, `defconstant` does not consider it to be an error to re-define a constant symbol as long as the new value is `equal` to the old. See p.78 of *Common Lisp: the language* for a discussion of the implications of constant folding.

Tail-recursion

By default, the compiler attempts to produce code which uses minimal stack space. To this end it is properly tail-recursive. Function calls which return the values returned by the called function are compiled in such a way as to re-use the calling function's stack frame. For example, if `foo` calls `bar` and `foo` returns the value returned by `bar`, `foo`'s stackframe will be re-used when `bar` is called. This means that the function call history displayed in a stack backtrace is not necessarily complete: the functions listed are only those which would be *returned to* if the computation resumed (i.e. only those functions awaiting return values). The compiler's license to collapse stack frames is governed by the compile-time value of the global variable `*nx-tailcalls*`; When non-`nil` (the default), stack space is conserved at the expense of debugging convenience.

Example:

```
(eval-when (compile eval)
  (setq *nx-tailcalls* t))

(defun foo (n)
 (if (= n 0)
   (break "Foo: n finally reached 0.")
                              ;; calls to BREAK are not compiled
                              ;; tail-recursively; see below.
   (foo (1- n))))

(foo 20)
```

When executed, this program will call itself 20 times and then enter a break loop. Examining the backtrace will show one instance (the most recent) of `foo` on the stack. If one sets `*nx-tailcalls*` to `nil` and then recompiles the definition of `foo`, the backtrace will show 20 additional instances of `foo` and the consumption of a few dozen additional bytes of stack space.

As a special case, the compiler never compiles calls to `error`, `cerror`, `warn`, or `break` tail-recursively. This means that the stack frame of the calling function is always available if a break loop is entered.

Compiler Declarations

`special` declarations are treated in accordance with the language specification.

`ignore` declarations are processed and used to generate/suppress compile warnings during file compilation.

`ftype` and `function` declarations, and declarations of the form `(TYPE <type> ...)` are ignored by the current version of the compiler.

Declarations of the form `(<type> ...)` are processed and the results are used to facilitate code optimization in some cases (see below).

The primary interpretation given to `inline` and `notinline` declarations is to control the compiler's treatment of `defstruct` accessor functions and primitive operations. In addition, `inline` declarations which refer to a given function advise that, within the body of that function, self-referential calls can be made as simple branch or call instructions. In the absence of such `inline` declarations, self-referential function calls are compiled exactly like all other function calls. This default behavior is consistent with the semantics of Object Lisp, allows function tracing, and is "correct" in pathological cases in which a function redefines itself. However, it involves slightly higher overhead than the case in which an `inline` declaration is used.

`optimize` declarations are given the following treatment:

The quality `compilation-speed` is effectively ignored; setting it to 3 disables a fairly quick branch-shortening pass in the code generator and results in slightly larger, slightly slower object code.

The other recognized qualities—`safety`, `space`, and `speed`—are used to select various optimization strategies. Each of the global variables described below is bound to a predicate function which, given arguments representing the current values of the `safety`, `space`, and `speed` qualities, returns `t` if the indicated optimization is to be performed. The user is free to redefine these predicates according to their own requirements.

`*nx-trust-declarations*` [*Variable*]
Specifies whether variable type declarations and type specifier arguments to the `the` special form should be trusted. The default predicate returns `t` when `speed` is greater than `safety`.

`*nx-fixnums-remain-fixnums*` [*Variable*]
Specifies whether arithmetic operations involving only arguments known or declared as fixnums will not cause fixnum overflow or underflow. The default predicate returns `t` when `speed` is 3 and `safety` is 0 or 1.

`*nx-open-code-in-line*` [*Variable*]
Specifies whether certain arithmetic operations are to be performed as out-of-line subprimitive calls or as in-line instruction sequences. The in-line sequences are usually much larger and slightly faster. The default predicate returns `t` when `speed` is 2 or more units greater than `space`.

nx-inline-car-cdr [*Variable*]
Specifies whether calls to `car` and `cdr` should perform run-time type checking on arguments whose type is unknown. Omission of this type-check can lead to system failure. The default predicate returns `T` when `speed` is `3` and `safety` is `0` or `1`.

In addition, declaring `speed 3` and `safety 0` inhibits the usual safety checks which are performed on function entry. Functions may omit number-of-arguments checking and stack-overflow checking and may skip periodic event processing. Needless to say, this combination of optimization settings should only be used in small sections of well-debugged code where execution speed is of great importance.

When `safety` is declared to be `0`, `defstruct` accessor functions not declared `notinline` are open-coded as memory access operations with no type-checking.

Lastly, declaring `space` to be more important than `safety` results in the suppression of the event-processing usually performed on backward branches. This results in iterative loops being somewhat smaller and faster but may make them uninterruptible.

The compiler also recognizes the declaration specifiers of the form

`(object-variable` *var1 var2 ... varN*)

The sole effect of `object-variable` declarations is to suppress compiler warnings about free references to any of the *var*s specified. The `objvar` macro wraps an `object-variable` declaration around its argument and is the preferred way of communicating object-variable declarations. Global object-variable declarations have the danger of suppressing free-variable warnings which should not be suppressed.

File Compiler

`compile-file` has been extended to accept the following additional keyword arguments:

`:verbose` When non-`nil`, causes a message to be printed as each file is compiled.
`:print` When non-`nil`, causes a digested version of each top-level form to be printed as it is compiled.
`:load` When non-`nil`, causes the result of a successful compilation to be loaded into the current environment.
`:features` should be a symbol or a list of symbols. These symbols are pushed onto the reader's `*features*` variable for the duration of the compilation.
`:warnings` When non-`nil` (the default), causes warnings about undeclared special variables and unused lexical variables to be printed after compilation.

Appendix B: System Parameters

Overview
Screen Information
Window Configuration
Allegro CL Menus
Environment Parameters
Modules
Compiler
Miscellaneous

Appendix B
System Parameters

Overview

Allegro CL provides a range global variables which describe and affect the system configuration. These global variables are described in this chapter. The standard Common Lisp globals are not discussed here.

screen information

screen-width [*Variable*]
screen-height [*Variable*]
the number of pixels of width and height of the current screen. On a Macintosh Plus or Macintosh SE, width will be 512 and height will be 342. On a Macintosh II with multiple screens, the values will refer to the main screen.

pixels-per-inch-x [*Variable*]
pixels-per-inch-y [*Variable*]
the number of pixels per inch in the horizontal and vertical directions. On a Macintosh Plus or Macintosh SE, both numbers will be 72. On a Macintosh II with multiple screens, the values will refer to the main screen.

menubar-bottom [*Variable*]
the vertical coordinate of the first quickdraw point below the menubar. On most Macintoshes, this will be 38. The number is provided so that programs do not accidently place windows behind the menubar.

Window configuration

fred-window-position [*variable*]
fred-window-size [*variable*]
listener-window-position [*variable*]
listener-window-size [*variable*]
These variables hold the default size and position for many of the windows that come up under Allegro CL. The values are given as points (see the chapter Macintosh Basics), and changed by the user.
`*fred-window-position*` and `*fred-window-size*` determine the size and position of newly created fred-windows.
When the listener is created, its size and position are determined by `*listener-window-position*` and `*listener-window-size*`. For these variables to have an effect, they must be set before the listener is created (i.e. from an initialization file, such as init.lisp).

top-listener [*Variable*]
bound to the top-most listener.

Allegro CL Menus

apple-menu [Variable]
file-menu [Variable]
edit-menu [Variable]
eval-menu [Variable]
tools-menu [Variable]
These variables are bound to the menus which appear when you first boot Allegro CL. The menus are normal menu-objects, and may be modified by Lisp code. Menu-items can be renamed, added, and deleted.

undo-menu-item [Variable]
This variable is bound to the **Undo** menu-item. The title of this item is sometimes changed to reflect the specifics of an undo operation (for example, after a cut, it may say **Undo Cut** rather than simply **Undo**).

Environment Parameters

The following global variables control user-selectable environment options. These can be conveniently set in the init file, or at any other time. They are also accessible through the **Environment...** menu-item.

compile-definitions [Defaults t] [*Variable*]
if t, all function definitions will be compiled, even if the defun was evaluated.

record-source-file [Defaults t] [*Variable*]
If t, causes the compiler and evaluator to associate a definition with the name of the file contaning the definition. Used for the Warn on Redefine and Edit Definition mechanisms.

save-doc-strings [Defaults nil] [*Variable*]
If non-nil, documentation strings are retained; if nil they are thrown away (which saves space).

save-definitions [Default t] [*Variable*]
If non-nil, the definitions of of compiled functions will be saved. These definitions can then be used by the stepper. If nil, the definitions are not saved (and therefore do not take up room).

load-verbose [Defaults nil] [*Variable*]
If non-nil, whenever a file is loaded, the name of the file is printed to the listener.

verbose-eval-selection [Defaults nil] [*Variable*]
This variable is used when a selection from a Fred window is evaluated. If it is t, the result of each evaluated expression is printed in the listener. Otherwise only the result of the last evaluated expression is printed.

warn-if-redefine [Default t] [*Variable*]
When lisp source files are loaded, the name of the source file is put on the property list of the names of defun and defmacro calls. If you load a source file that contains a definition for a name previously defined in a different file, and *warn-if-redefine?* is non-nil, then a warning message will be printed in the Listener.

break-on-warnings [Default `nil`] *[Variable]*
If non-`nil`, warnings will cause the Allegro CL system to enter a break loop, and a stack backtrace will be shown. Note that Allegro CL is properly tail-recursive, so some functions may not be visible in the stack-backtrace.

break-on-errors [Default `t`] *[Variable]*
If non-`nil`, errors will cause the Allegro CL system to enter a break loop.

backtrace-on-break [Default `nil`] *[Variable]*
If non-`nil`, entering a break loop through an error will cause the backtrace dialog to be displayed. At any point in a break when the backtrace is not shown, it may be brought up by selecting the **Backtrace** menu-item from the **Tools** menu.

fast-eval [Default `nil`] *[Variable]*
If non-`nil`, Allegro CL will evaluate expressions by first compiling them, and then running the compiled result. This can be faster for complex expressions, but `*apply-hook*` and `*eval-hook*` will not function.

emacs-mode [Default `nil`] *[Variable]*
If non-`nil`, the clover key is used for control, and clover-shift is used for command. If `nil` the clover key is used for command, and clover-shift is used for control. See the chapter Getting Started for details.

Modules

See appendix Common Lisp Implementation for details on the use of the following variables.

module-file-alist [Default `()`] *[Variable]*
an alist. The `car` of each element should be the name of a module. The `cdr` of each element should be a pathname or list of pathnames making up the module. This variable is used by `require`.

module-search-path [Default `("Library")`] *[Variable]*
a list of pathnames. These pathnames are used by `require` to locate modules.

Compiler

nx-tailcalls [Default `t`] *[Variable]*
when non-`nil`, the compiler will re-use stack-frames on tail-recursive function calls. This conserves stack space but is a little bit slower and can make debugging more confusing (because function calls disappear from the stack-backtrace). See the appendix Implementation Notes for details.

nx-trust-declarations [*Variable*]
nx-fixnums-remain-fixnums [*Variable*]
nx-open-code-in-line [*Variable*]
nx-inline-car-cdr [*Variable*]
these variables contain predicate functions used in conjunction with `speed`, `safety`, and `space` declarations. The predicates determine whether certain compiler transforms and optimizations and transforms take place. They are described in more detail in the appendix Implementation Notes.

Miscellaneous

d [*Variable*]
During debugging with the backtrace dialog, `*d*` is bound to the currently selected value from the currently selected frame. This variable may be used in expressions during debugging sessions, much as *, **, and *** are used during normal Lisp interactions.

next-screen-context-lines [Default 2] [*Variable*]
should always be a fixnum. This is the number of lines that are retained from screen to screen when the user scrolls through editor buffers using the `control-v` and `meta-v` commands.

killed-strings [*Variable*]
a list of the killed strings. This variable is used by Fred the editor.

white-pattern [*Variable*]
black-pattern [*Variable*]
gray-pattern [*Variable*]
light-gray-pattern [*Variable*]
dark-gray-pattern [*Variable*]
These variables hold patterns which may be used in conjunction with Quickdraw calls.

font-list [*Variable*]
a list of all the font-names (as strings) available for use by Allegro CL.

Appendix C: Quickdraw Graphics

Overview
Windows, Grafports, and Portrects
Points and Rectangles
General Window State
Cursor Primitives
Pen and Line-drawing Routines
Drawing Text
Calculations with Rectangles
Graphics Operations on Rectangles
Graphics Operations on Ovals
Graphics Operations on RoundRects
Regions
Graphics Operations on Arcs
Calculations with Regions
Graphics Operations on Regions
Bitmaps
Pictures
Polygons
Graphics Operations on Polygons
Miscellaneous Procedures

Appendix C
Quickdraw Graphics

Overview

This chapter outlines a set of interface functions between Allegro CL and Quickdraw, the Macintosh graphics package. The code which implements these functions is included as an example file. Before calling any of the functions described in this chapter, the user must load the Quickdraw file, which can be found in the Library directory. Some familiarity with the Quickdraw chapter of *Inside Macintosh* is assumed.

The interface routines support all of the functionality found in Quickdraw for Macintoshes with 64K ROMS. For newer Quickdraw features, traps may be called directly, as outlined in the chapter Low Level System Interface.

Some Quickdraw functions may only be performed within window objects (i.e. by asking the window to do them). These are the functions that depend on a grafport and are labeled as *window functions* in brackets along the right column. All other functions may be performed globally.

The arguments to the Allegro CL Quickdraw functions generally parallel the arguments to the Pascal Quickdraw function given in *Inside Macintosh.* In several cases we have extended the Pascal functionality by taking advantage of Common Lisp's optional arguments. In some cases we have changed the order of arguments to make the mapping of the optional arguments more effective. In some cases *var* arguments have been eliminated, and instead a value is returned.

Windows, Grafports, and Portrects

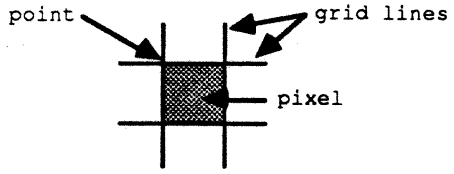
All drawing on the Macintosh takes place inside grafports. Grafports are usually associated with windows. In low level Macintosh drawing, there are several levels of initialization in setting up windows and grafports for drawings. Once they have been created, you must keep track of the current grafport when you do any drawing. This process is simplified for the graphics routines given in this chapter. When you create a window, an initialized grafport is automatically created. Drawing commands are defined as object functions for windows; when you `ask` a window to perform one of the commands, grafports are set and restored as needed.

Drawing inside windows is automatically cropped to the inside of the window, and to the portions of the window that are visible (i.e. not covered up by other windows). Drawing is also affected by the clip-region (described below), and the portrect. The portrect is an arbitrary rectangle designating the outermost bounds in which drawing can occur. It supplies a frame of reference for the window. The default portrect is infinitely large; it can be set with low-level calls (though you usually won't need to worry about the portrect at all).

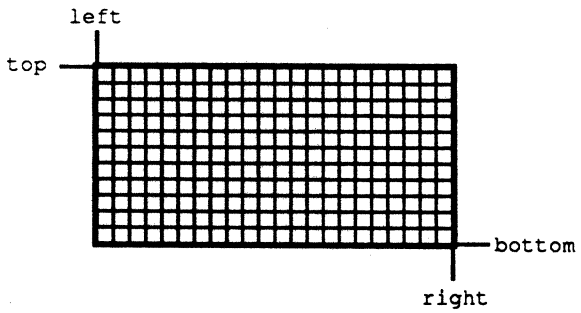
Points and Rectangles

In Quickdraw points are specified by two coordinates, the horizontal coordinate (called `h`) and the vertical coordinate (called `v`). `h` increases to the right and `v` increases down. The upper left hand corner of a window is usually the point 0,0 but it may have any coordinates. This may be changed by using the `set-region` function.

Points are stored in encoded fixnums. A general description of the Allegro CL point data format can be found in the chapter Macintosh Basics. Points lie at the intersection of two grid lines on the QuickDraw plane. Note: points and pixels are not equivalent. The point associated with a given pixel is at the upper-left corner of the pixel.



The Macintosh stores rectangles as eight-byte records. (Records are blocks of non-Lisp data stored on the Macintosh heap; see the chapter Pascal Records for details.) Rectangle records can be thought of as two points (top-left and bottom-right), or four edges (left, top, right, and bottom). Allocating memory for rectangle records can be inefficient, so Allegro CL provides several ways of doing this. `make-record` is used to allocate memory for long-lived rectangles, and `rlet` is used to allocate records for short-lived rectangles (see the chapter Pascal Records for details). For many of the Allegro CL Quickdraw functions that take rectangles, rectangle records do not need to be allocated at all; the rectangles can be specified by four coordinates, two points, *or* a rectangle record. In general, if a rectangle is used only once, it is all right to pass it as two points or four coordinates. However, if the rectangle is used several times, it is more efficient to create and pass an actual rectangle record.

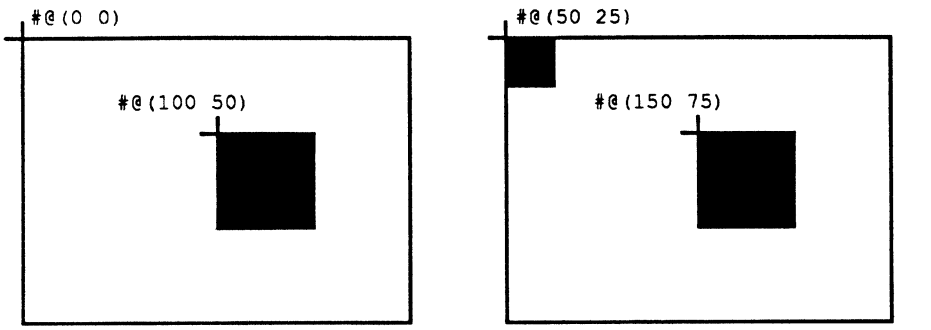


In cases where alternate forms of a point or a rectangle are accepted as arguments, the flexible argument appears last. This is to avoid ambiguity as to which argument is which, and explains why the order of arguments is sometimes different from the order given in *Inside Macintosh.*

General Window State

origin [*Window Function*]
set-origin *h* &optional (*v* nil) [*Window Function*]
origin returns the coordinates of the top-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.

set-origin sets the origin to the point specified by *h* and *v*. If only *h* is given, it is taken to be an encoded point. The contents of the window are not moved; only future drawing is effected.




```
(set-origin #@(50 25))
(fill-rect 50 25 75 50)
```

clip-region &optional *save-region* [*Window Function*]
set-clip-region *new-region* [*Window Function*]
A clip-region allows drawing in a window to be constrained to an arbitrary region. Drawing will only occur within the clip-region. The default clip-region is arbitrarily large, so no clipping takes place.

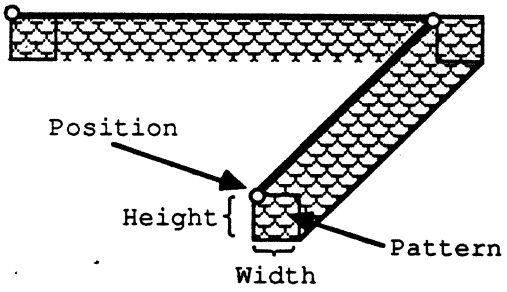
clip-region returns the window's current clip-region. If *save-region* is supplied, it is used to hold the returned region, otherwise the clip-region is returned in a newly allocated region. Note: regions must be explicitly disposed of; they are not automatically garbage collected.

set-clip-region sets the window's clip-region to *new-region*. *new-region* is returned. See the section on Regions, below, for functions which allocate and manipulate regions.

clip-rect *rectangle* [*Window Function*]
clip-rect *top-left-point bottom-right-point*
clip-rect *left top right bottom*
sets the window's clip region to be a rectangular region equivalent to *rectangle*. nil is returned.

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, no drawing commands will actually have an effect on the screen. In addition to being hidden or shown, a pen has a size, position in the window, and pattern used for drawing.



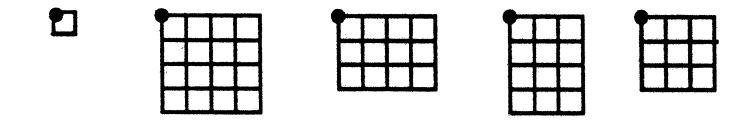
Pen attributes

pen-show [*Window Function*]
pen-hide [*Window Function*]
pen-shown-p [*Window Function*]
`pen-show` shows the pen.
`pen-hide` hides the pen. If the pen is hidden, no drawing will occur.
`pen-shown-p` returns `t` if the pen is shown, `nil` if the pen is hidden.

pen-position [*Window Function*]
returns a point corresponding to the pen position in local coordinates.

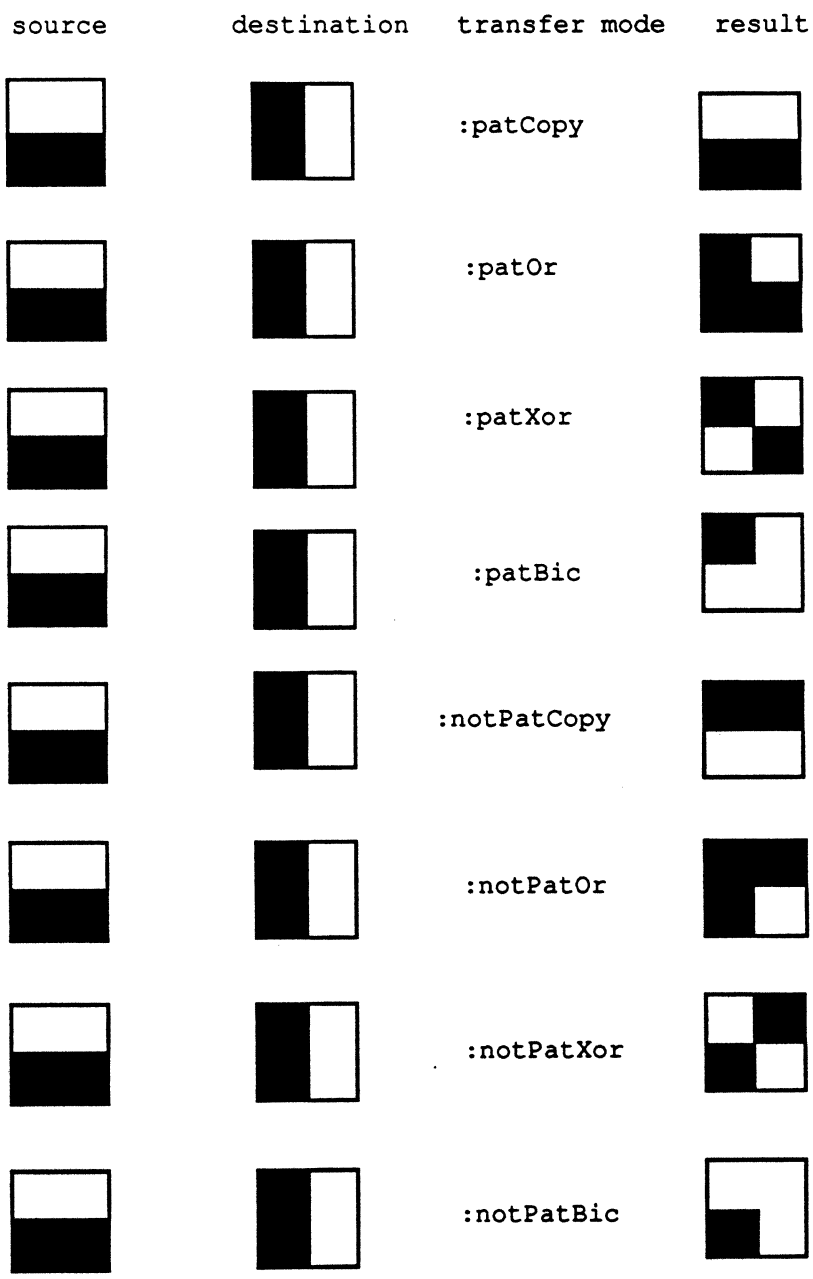
pen-size [*Window Function*]
set-pen-size *point*
`pen-size` returns the current pen-size as a point (expressing a width and height).
`set-pen-size` sets the pen-size to *point*.

```
QuickDraw PenSizes:
• - indicates pen location of the pen
```



pen-mode [*Window Function*]
set-pen-mode *new-mode* [*Window Function*]
pen-modes effect 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` returns a keyword indicating the window's current pen-mode.

`set-pen-mode` sets the window's current pen-mode. *new-mode* should be one of the following keywords: `:patCopy`, `:patOr`, `:patXor`, `:patBic`, `:notPatCopy`, `:notPatOr`, `:notPatXor`, `:notPatBic`.

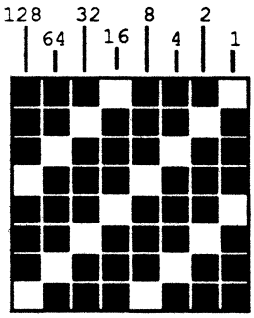


pen-pattern `&optional` *save-pattern* [*Window Function*]
set-pen-pattern *new-pattern* [*Window Function*]
`pen-pattern` returns the window's pen pattern. If *save-pattern* is given, it should be a pattern record; the pattern will be returned in this record. If *save-pattern* is not given, a new pattern record will be allocated for holding the returned pattern. Pattern records (like all records) will continue to take up space on the Macintosh heap until they are explicitly disposed of.

`set-pen-pattern` sets the window's pen-pattern. *new-pattern* should be a pattern record.

A pattern is stored as a 64-bit block of memory. The definition of a pattern record allows patterns to be accessed as 8 bytes or 4 words.

```
(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))))
```




```
128 + 64 + 32 + 8 + 4 + 2 = 238 --> b0
128 + 64 + 16 + 8 + 4 + 1 = 221 --> b1
128 + 32 + 16 + 8 + 2 + 1 = 187 --> b2
 64 + 32 + 16 + 4 + 2 + 1 = 119 --> b3
128 + 64 + 32 + 8 + 4 + 2 = 238 --> b4
128 + 64 + 16 + 8 + 4 + 1 = 221 --> b5
128 + 32 + 16 + 8 + 2 + 1 = 187 --> b6
 64 + 32 + 16 + 4 + 2 + 1 = 119 --> b7
```

There are five patterns stored as constants by Allegro CL: `*white-pattern*`, `*black-pattern*`, `*gray-pattern*`, `*light-gray-pattern*`, and `*dark-gray-pattern*`.

pen-state `&optional` *save-pen-state* [*Window Function*]
set-pen-state *new-pen-state* [*Window Function*]
`pen-state` returns the current pen-state, a record containing the pen's location, size, mode (as an integer), and pattern. If *save-pen-state* is given, it should be a pointer to a pen-state record; the returned state will be stored in this record. If *save-pen-state* is not given, the pen-state will be returned in a newly allocated record. Pen-state records (like all records) will continue to take up space on the Macintosh heap until they are explicitly disposed of.

`set-pen-state` sets the window's pen-state. *new-pen-state* should be a pen-state record.

Pen-state records represent the pen-mode as an integer. This integer will equal 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*)`.

The definition of a penstate record is:

```
(defrecord PenState
       (pnLoc point)
       (pnSize point)
       (pnMode integer)
       (pnPat pattern))
```

pen-normal [*Window Function*]
sets the pen's size, pattern, and mode to normal. The pen-size is set to `#@(1 1)`, pen-mode to `:patCopy`, and pen-pattern to `*black-pattern*`. The pen location is not changed.

move-to *h* `&optional (v nil)` [*Window Function*]
moves the pen to the point specified by *h* and *v*. If *v* is `nil`, *h* is interpreted as an encoded point. No drawing occurs. The point moved to is returned.

move *h* `&optional (v nil)` [*Window Function*]
moves the pen *h* points to the right and *v* points down. If *v* is `nil`, *h* is interpreted as an encoded point and its two coordinates are used. No drawing occurs. The encoded form of *h* and *v* is returned.

line-to *h* `&optional (v nil)` [*Window Function*]
draws a line from the pen's current position to *h* and *v*. If *v* is `nil`, *h* is interpreted as an encoded point.

line *h* `&optional (v nil)` [*Window Function*]
draws a line to a point *h* points to the right and *v* points down from the current pen position. If only *h* is given, it is interpreted as a point, and its two coordinates are used.

Drawing Text

Text is drawn in windows by using a window as an output stream. Text is drawn 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 return (i.e. from `terpri` and `fresh-line`) and backspace have no effect. Note that when a window is created, its pen-position is `#@(0 0)`. This means that any text drawn into it will be above the visible portion of the window until the pen-position is lowered.

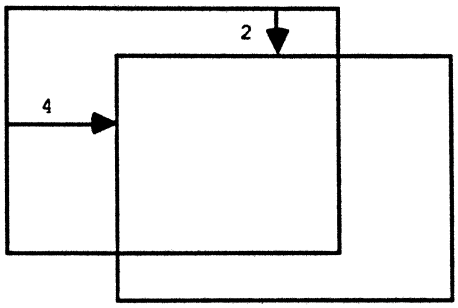
stream-tyo *char* [*Window Function*]
This function draws char at the current pen position, in the current font using the current text transfer mode, and moves the pen to the right the width of the character. Since windows are streams, all stream output functions (such as `prin1`) can be performed to them. `stream-tyo` is not normally called directly but is called by stream output functions.

Calculations with Rectangles

These functions do not draw, they simply perform calculations. They do not depend on a grafport, and so they are defined globally, rather than as object functions.

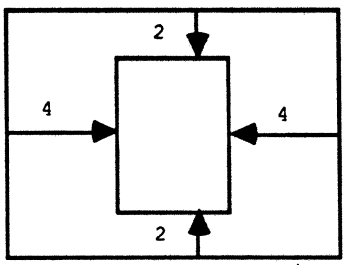
offset-rect *rectangle h* &optional (*v* nil) [*Function*]
moves *rectangle h* to the right and *v* down. If only *h* is given, it is interpreted as an encoded point. *rectangle* is destructively modified and returned.

```
(offset-rect rect 4 2)
```

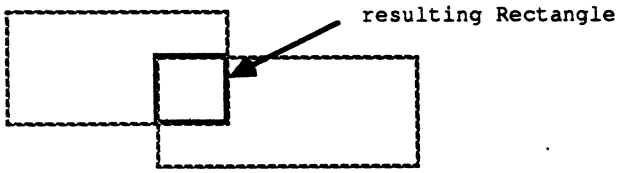


inset-rect *rectangle h* &optional (*v* nil) [*Function*]
insets *rectangle* by *h* and *v* . If only *h* is supplied it is interpreted as an encoded point. *rectangle* is destructively modified and returned.

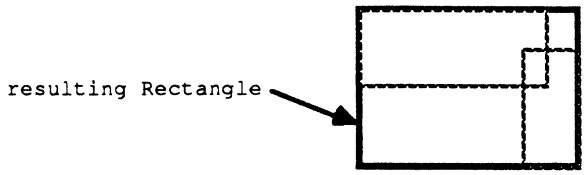
```
(inset-rect rect 4 2)
```



intersect-rect *rect1 rect2 dest-rect* [*Function*]
stores the rectangle that is the intersection of *rect1 and rect2* into *dest-rect* and returns *dest–rect* . *rect1* or *rect2* may be used as *dest-rect.*



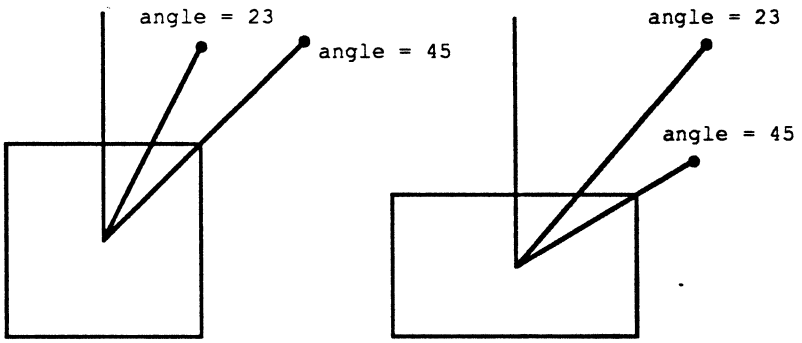
union-rect *rect1 rect2 dest-rect* [*Function*]
stores the smallest rectangle that encloses both *rect1 and rect2* into *dest-rect* and returns *dest–rect. rect1* or *rect2* may also be used as *dest-rect.*



point-in-rect-p *rectangle h* `&optional (v nil)` [*Function*]
returns t if the point specified by *h* and *v* is inside of *rectangle*, otherwise `nil`. If only *h* is given, it is interpreted as a point and its coordinates are used.

points-to-rect *point1 point2 dest-rect* [*Function*]
stores the smallest rectangle that encloses both *point1* and *point2* into *dest-rect* and returns *dest-rect. point1* and *point2* will be the top-left and bottom-right corners of the returned rectangle.

point-to-angle *rectangle h* `&optional (v nil)` [*Function*]
returns an "angle number" calculated from *rectangle* and the point specified by *h* and *v*(see *Inside Macintosh* for details). If only *h* is given, it is interpreted as an encoded point.

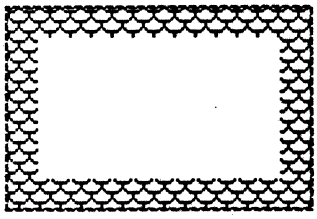


equal-rect *rect1 rect2* [*Function*]
returns `t` if *rect1* and *rect2* are equal, `nil` otherwise.

empty-rect-p *rectangle* [*Function*]
empty-rect-p *top-left-point bottom-right-point*
empty-rect-p *left top right bottom*
returns t if *rectangle* is empty (contains no points), `nil` otherwise. A rectangle is empty if its bottom is equal to or above its top, or if its left side is equal to or to the right of its right side.

Graphics Operations on Rectangles

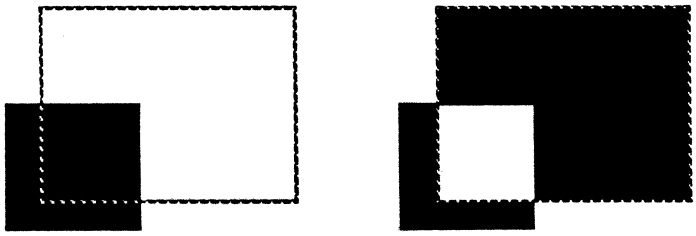
frame-rect *rectangle* [*Window Function*]
frame-rect *top-left-point bottom-right-point*
frame-rect *left top right bottom*
draws a line just inside the boundaries of *rectangle* using the current pen.



paint-rect *rectangle* [*Window Function*]
paint-rect *top-left-point bottom-right-point*
paint-rect *left top right bottom*
fills *rectangle* with the current pen pattern and mode.

erase-rect *rectangle* [*Window Function*]
erase-rect *top-left-point bottom-right-point*
erase-rect *left top right bottom*
fills *rectangle* with the current background pattern using patCopy mode.

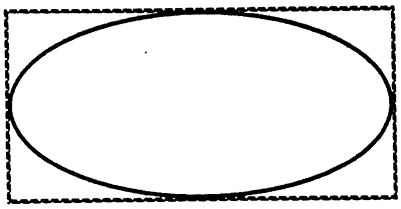
invert-rect *rectangle* [*Window Function*]
invert-rect *top-left-point bottom-right-point*
invert-rect *left top right bottom*
inverts the pixels inside of *rectangle.*



fill-rect *pattern rectangle* [*Window Function*]
fill-rect *pattern top-left-point bottom-right-point*
fill-rect *pattern left top right bottom*
fills *rectangle* with *pattern* using patCopy mode.

Graphics Operations on Ovals

Ovals are drawn inside of rectangles.



frame-oval *rectangle* [*Window Function*]
frame-oval *top-left-point bottom-right-point*
frame-oval *left top right bottom*
draws a line just inside the boundaries of the oval specified by *rectangle* using the current pen.

paint-oval *rectangle* [*Window Function*]
paint-oval *top-left-point bottom-right-point*
paint-oval *left top right bottom*
fills the oval specified by *rectangle* with the current pen pattern and mode.

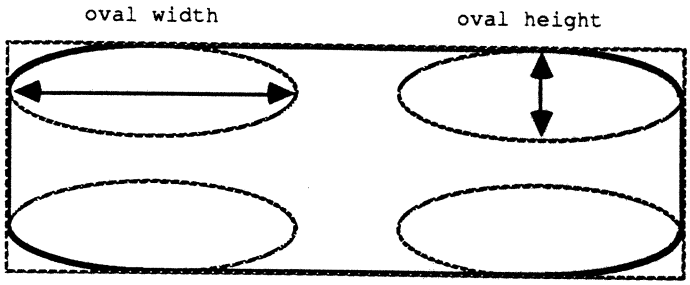
erase-oval *rectangle* [*Window Function*]
erase-oval *top-left-point bottom-right-point*
erase-oval *left top right bottom*
fills the oval specified by *rectangle* with the current background pattern using patCopy mode.

invert-oval *rectangle* [*Window Function*]
invert-oval *top-left-point bottom-right-point*
invert-oval *left top right bottom*
inverts the pixels enclosed by the oval specified by *rectangle.*

fill-oval *pattern rectangle* [*Window Function*]
fill-oval *pattern top-left-point bottom-right-point*
fill-oval *pattern left top right bottom*
fills the oval specified by *rectangle* with *pattern* using patCopy mode.

Graphics Operations on RoundRects

A roundrect is a rectangle whose corners are rounded. The shapes of the corners are determined by ovals associated with the roundrects.



frame-round-rect *oval-width oval-height rectangle* [*Window Function*]
frame-round-rect *oval-width oval-height top-left-point bottom-right-point*
frame-round-rect *oval-width oval-height left top right bottom*
draws a line just inside the boundaries of the roundrect specified by *rectangle*, *oval-width* and *oval–height* using the current pen.

paint-round-rect *oval-width oval-height rectangle* [*Window Function*]
paint-round-rect *oval-width oval-height top-left-point bottom-right-point*
paint-round-rect *oval-width oval-height left top right bottom*
fills the roundrect specified by *rectangle*, *oval-width* and *oval-height* with the current pen pattern and mode.

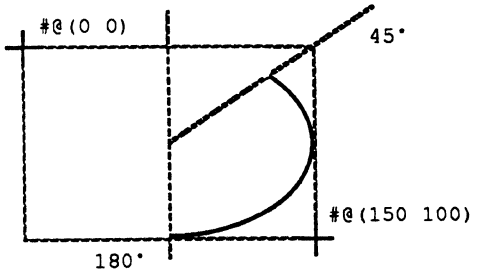
erase-round-rect *oval-width oval-height rectangle* [*Window Function*]
erase-round-rect *oval-width oval-height top-left-point bottom-right-point*
erase-round-rect *oval-width oval-height left top right bottom*
fills the roundrect specified by *rectangle*, *oval-width* and *oval-height* with the current background pattern using patCopy mode.

invert-round-rect *oval-width oval-height rectangle* [*Window Function*]
invert-round-rect *oval-width oval-height top-left-point bottom-right-point*
invert-round-rect *oval-width oval-height left top right bottom*
inverts the pixels enclosed by the roundrect specified by *rectangle*, *oval-width* and *oval-height.*

fill-round-rect *pattern oval-width oval-height rectangle* [*Window Function*]
fill-round-rect *pattern oval-width oval-height top-left-point bottom-right-point*
fill-round-rect *pattern oval-width oval-height left top right bottom*
fills the roundrect specified by *rectangle*, *oval-width* and *oval-height* with *pattern* using patCopy mode.

Graphics Operations on Arcs

frame-arc *start-angle arc-angle rectangle* [*Window Function*]
frame-arc *start-angle arc-angle top-left-point bottom-right-point*
frame-arc *start-angle arc-angle left top right bottom*




```
(frame-arc 45 135
           0 0 150 100)
```

draws a line just inside the arc specified by *rectangle*, *start-angle* and *arc-angle* using the current pen.

paint-arc *start-angle arc-angle rectangle* [*Window Function*]
paint-arc *start-angle arc-angle top-left-point bottom-right-point*
paint-arc *start-angle arc-angle left top right bottom*
fills the arc specified by *rectangle*, *start-angle* and *arc-angle* with the current pen pattern and mode.

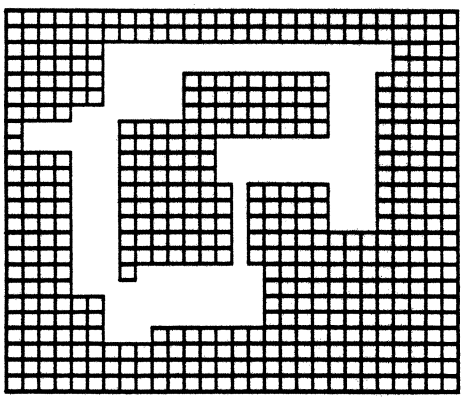
erase-arc *start-angle arc-angle rectangle* [*Window Function*]
erase-arc *start-angle arc-angle top-left-point bottom-right-point*
erase-arc *start-angle arc-angle left top right bottom*
fills the arc specified by *rectangle*, *start-angle* and *arc-angle* with the current background pattern using patCopy mode.

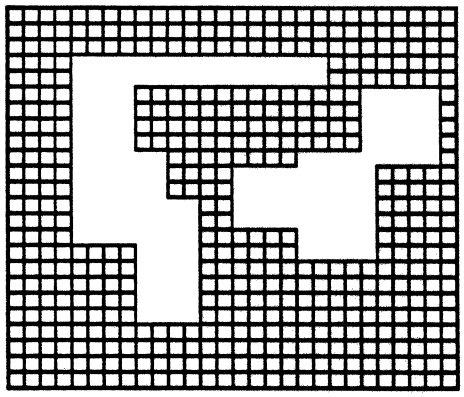
invert-arc *start-angle arc-angle rectangle* [*Window Function*]
invert-arc *start-angle arc-angle top-left-point bottom-right-point*
invert-arc *start-angle arc-angle left top right bottom*
inverts the pixels enclosed by the arc specified by *rectangle*, *start-angle* and *arc-angle*.

fill-arc *pattern start-angle arc-angle rectangle* [*Window Function*]
fill-arc *pattern start-angle arc-angle top-left-point bottom-right-point*
fill-arc *pattern start-angle arc-angle left top right bottom*
fills the arc specified by *rectangle*, *start-angle* and *arc-angle* with *pattern* using patCopy mode.

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 be any arbitrary shape.





The storage for regions is not automatically garbage-collected; region storage must be manually reclaimed with the function `dispose-region`. Within 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.

`new-region` [*Window Function*]
allocates a new empty region and returns it.

`dispose-region` *region* [*Window Function*]
reclaims storage space used by *region* and returns `nil`.

`copy-region` *region* `&optional` *dest-region* [*Window Function*]
if *dest-region* is supplied, changes it to be equivalent to *region* and returns it. If *dest-region* is not supplied, creates a new region equivalent to *region* and returns it. Note that if a new region is created, it must be explicitly disposed of to reclaim its storage space.

`set-empty-region` *region* [*Window Function*]
destructively modifies *region* to be an empty region and returns it.

`set-rect-region` *region rectangle* [*Window Function*]
`set-rect-region` *region top-left-point bottom-right-point*
`set-rect-region` *region left top right bottom*
sets *region* to be a rectangular region equivalent to *rectangle* and returns it.

`open-region` [*Window Function*]
hides the pen and begins recording a region. Subsequent drawing commands to the window will add to the region. The recording will be terminated when `close-region` is called. It is an error to call `open-region` a second time without first calling `close-region`. Returns `nil`.

close-region &optional *dest-region* [*Window 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. If *dest-region* is specified, it will hold the returned value. Otherwise a new region will be created. It is an error to do a close-region before an open-region has been done. Note that if a new region is created, it must be explicitly disposed of to reclaim its storage space.

Calculations with Regions

offset-region *region h* &optional (*v* nil) [*Function*]
destructively offsets *region* by *h* to the right and *v* down and returns it. If only *h* is given, it is interpreted as an encoded point and its coordinates are used.

inset-region *region h* &optional (*v* nil) [*Function*]
destructively insets *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.

intersect-region *region1 region2* &optional *dest-region* [*Function*]
returns a region that is the intersection of *region1* and *region2*. The result is returned in *dest-region* if supplied, or else in a newly created region. Note that if a new region is created, it must be explicitly disposed of to reclaim its storage space.

union-region *region1 region2* &optional *dest-region* [*Function*]
returns a region that is the union of *region1* and *region2*. The result is returned in *dest-region* if supplied, or else in a newly created region. Note that if a new region is created, it must be explicitly disposed of to reclaim its storage space.

difference-region *region1 region2* &optional *dest-region* [*Function*]
returns a region that is the difference of *region1* and *region2*. The result is returned in *dest-region* if supplied, or else in a newly created region. Note that if a new region is created, it must be explicitly disposed of to reclaim its storage space.

xor-region *region1 region2* &optional *dest-region* [*Function*]
returns a region that consists of all the points that are in *region1* or *region2* but not both. The result is returned in *dest-region* if given, or else in a newly created region. Note that if a new region is created, it must be explicitly disposed of to reclaim its storage space.

point-in-region-p *region h* &optional (*v* nil) [*Function*]
returns t if the point specified by *h* and *v* is contained in *region*,otherwise nil. If only *h* is given, it is interpreted as an encoded point.

rect-in-region-p *region rectangle* [*Function*]
rect-in-region-p *region top-left-point bottom-right-point*
rect-in-region-p *region left top right bottom*
returns t if the intersection of *rectangle* and *region* contains at least one point, otherwise nil

equal-region-p *region1 region2* [*Function*]
returns t if *region1* and *region2* are identical in size, shape and position, nil otherwise.

empty-region-p *region* [*Function*]
returns `t` if *region* contains no points, `nil` otherwise.

Graphics Operations on Regions

frame-region *region* [*Window Function*]
draws a line just inside the boundaries of *region* using the current pen.

paint-region *region* [*Window Function*]
fills *region* with the current pen pattern and mode.

erase-region *region* [*Window Function*]
fills *region* with the current background pattern using patCopy mode.

invert-region *region* [*Window Function*]
inverts the pixels enclosed by *region.*

fill-region *pattern region* [*Window Function*]
fills *region* with *pattern* using patCopy mode.

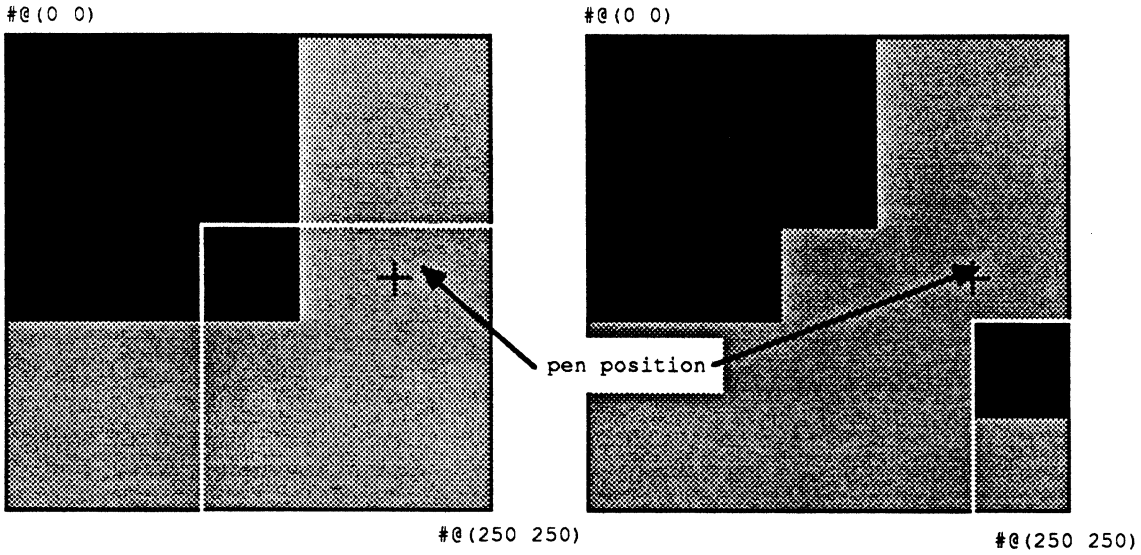
Bitmaps

Bitmaps are rectangular arrays of pixels which are either black or white.

make-bitmap *rectangle* [*Function*]
make-bitmap *top-left-point bottom-right-point*
make-bitmap *left top right bottom*
returns a new bitmap the size of *rectangle.* This bitmap is not displayed anywhere, but can be used for calculations and storage.

copy-bits *bitmap1 bitmap2 rect1 rect2* `&optional` *pen-mode region* [*Function*]
copies and scales the bits inside *rect1* of *bitmap1* to the bits inside *rect2* of *bitmap2* using the the transfer mode *pen-mode.* If *region* is given, the transferred bitmap is clipped to *region.*and *pen–mode* defaults to `:srcCopy`.

scroll-rect *rectangle h* &optional (*v* nil) [*Window Function*]
shifts the bits inside *rectangle h* pixels to the right and *v* pixels down within rectangle, erases the uncovered region and adds it to the window's update region. If only *h* is given, it is interpreted as an encoded point and its coordinates are used.




```
(scroll-rect 100 100 250 250
             100 50)
```

Pictures

A picture is a recording of a sequence of QuickDraw commands which may be played back at a later time, into the same window or into a different window. The Allegro CL picture commands are slightly different from the QuickDraw ones, since Allegro CL takes care of some of the memory management automatically. There are also some additional capabilities for manipulating pictures not found in QuickDraw.

start-picture &optional *rectangle* [*Window Function*]
start-picture &optional *top-left-point bottom-right-point*
start-picture &optional *left top right bottom*
hides the pen and starts recording Quickdraw commands in a picture with frame *rectangle,* if supplied, otherwise with the window's portrect as a frame. You must terminate a `start-picture` with a `get-picture` before another `start-picture` can be done. Returns `nil`.

get-picture [*Window Function*]
shows the pen and returns a new picture that consists of all the Quickdraw commands since the last `start-picture`. It is an error to do a `get-picture` before a `start-picture` has been done for a window. Note that pictures must be explicitly disposed of using `kill-picture` to reclaim their storage space.

draw-picture *picture* &optional *rectangle-or-point* [*Window Function*]
draws *picture* in the window. If *rectangle-or-point* is not given, the picture is drawn in its original size and position. If *rectangle-or-point* is a point, *picture* is drawn in its original size, but

with its upper left corner at the local coordinate given by *rectangle-or-point*. If *rectangle-or-point* is a rectangle, *picture* is drawn scaled into the rectangle. 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 Allegro CL), then scaling will produce no drawing (since the drawing frame is so much smaller than the creation frame). `draw-picture` returns *picture*.

`kill-picture` *picture* [*Window Function*]
reclaims the storage space used by *picture* and returns `nil`.

Polygons

The Allegro CL polygon commands are different from the QuickDraw ones, since Allegro CL handles some of the memory management automatically.

`start-polygon` [*Window 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, you must terminate a `start-polygon` with a `get-polygon` before another `start-polygon` can be done.

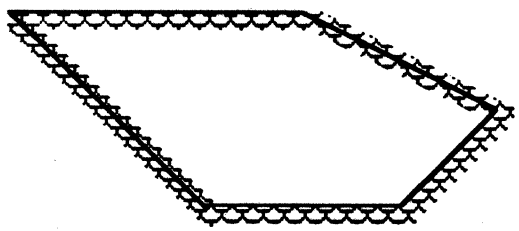
`get-polygon` [*Window Function*]
shows the pen and returns a polygon that consists of all the `line` and `line-to` commands since the last `start-polygon`. It is an error to do a `get-polygon` before a `start-polygon` has been done for a window. Note that polygons must be explicitly disposed of using `kill-polygon` to reclaim their storage space.

`kill-polygon` *polygon* [*Window Function*]
reclaims storage space used by *polygon* and returns `nil`.

`offset-polygon` *polygon h* `&optional (v nil)` [*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.

Graphics Operations on polygons

`frame-polygon` *polygon* [*Window Function*]
draws a line just inside the boundaries of *polygon* using the current pen.



`paint-polygon` *polygon* [*Window Function*]
fills *polygon* with the current pen pattern and mode.

`erase-polygon` *polygon* [*Window Function*]
fills *polygon* with the current background pattern using patCopy mode.

invert-polygon *polygon* [*Window Function*]
inverts the pixels enclosed by *polygon*.

fill-polygon *pattern polygon* [*Window Function*]
Fills *polygon* with *pattern* using patCopy mode.

Miscellaneous Procedures

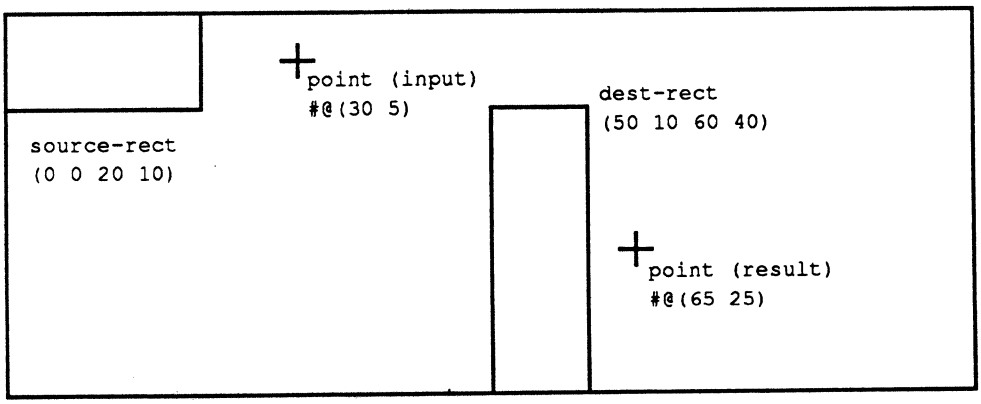
local-to-global *h* `&optional (v nil)` [*Window Function*]
returns a global point that corresponds to the window's local point specified by *h* and *v*. If only *h* is given, it is taken to be an encoded point.

global-to-local *h* `&optional (v nil)` [*Window 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.

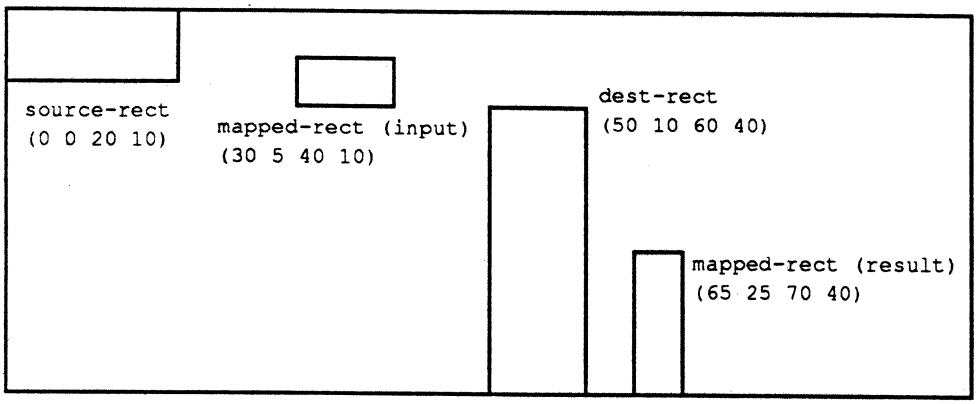
get-pixel *h* `&optional (v nil)` [*Window Function*]
returns `t` if the pixel specified by *h* and *v* is black and within the window's VisRgn, `nil` otherwise. If only *h* is given, it is interpreted as an encoded point.

scale-point *rect1 rect2 h* `&optional (v nil)` [*Function*]
returns a points whose h and v values are the scaled h and v values of the point specified by *h* and *v*. If only *h* is give, it is interpreted as an encoded point. The scaling corresponds to the ratios of *rect1*'s width and height to *rect2*'s width and height.

map-point *source-rect dest-rect h* `&optional (v nil)` [*Function*]
returns a point that corresponds to *dest-rect* in the way that the point specified by *h* and *v* corresponds to *source-rect*. If only *h* is give, it is interpreted as an encoded point.



map-rect *source-rect dest-rect mapped-rect* [*Function*]
returns a rectangle that corresponds to *dest-rect* in the same way that *mapped-rect* corresponds to *source-rect*. *mapped-rect* is destructively modified to hold the return value. This function is performed by calling `map-point` on the corner points of `mapped-rect`.



map-region *source-rect dest-rect region* [*Function*]
returns a region that corresponds to *dest-rect* in the same way that *region* corresponds to *source-rect*. *region* is destructively modified to hold the return value. This function is effectively performed by calling `map-point` on all the points in the region.

map-polygon *source-rect dest-rect polygon* [*Function*]
returns a polygon that corresponds to *dest-rect* in the same way that *polygon* corresponds to *source-rect*. *polygon* is destructively modified to hold the return value. This function is effectively performed by calling `map-point` on all the points which define the polygon.

Appendix D: Selected Bibliography

Overview
Common Lisp References
Lisp Tutorials
Artificial Intelligence
Macintosh

Appendix D
Selected Bibliography

Overview

This appendix list books that you may find useful while programming in Allegro CL for the Macintosh.

Common Lisp References

Steele, Guy *Common Lisp: the Language*
Digital Press, Manard, MA, 1984
465 pages. The standard reference manual and language specification for Common Lisp. A must.

Simpson, Rosemary *Common Lisp the Index*
Coral Software Corp, Cambridge, MA, 1987
74 pages. An extensive cross-referenced index to Guy Steele's *Common Lisp: the Language.*

Lisp Tutorials

Abelson, Harold & Gerald Sussman *Structure and Interpretation of Computer Programs*
MIT Press, Cambridge, MA, 1985
542 pages. Originally developed for the introductory programming course at MIT. Stresses large scale issues of program design and implementation. Good chapter on writing an evaluator. Uses Scheme (rather than Common Lisp) for all examples.

Brooks, Rodney *Programming in Common Lisp*
John Wiley & Sons, New York, NY. 1985
303 pages. Contains tutorial and problems (answers included).

Friedman, Daniel and Matthias Fellestien *The Little Lisper, trade edition*
MIT Press, Cambridge, MA. 1987
186 pages. Introductory text. Exercises in Lisp that have made complex concepts transparent to many. Does not use the Common Lisp but sticks to a very few set of lisp functions and includes compatibility notes. Strong on recursion.

Tatar, Deborah *A Programmer's Guide to Common Lisp*
Digital Press, Maynard, MA. 1987
327 pages. The last chapter contains the code and English explanation of a toy expert system.

Touretzky, David *Lisp: A Gentle Introduction to Symbolic Computing*
Harper & Row, 1984
384 pages. Uses the most common features of Common Lisp in the examples. Gentle, as the title suggests.

Winston, Patrick and Bertold Horn *Lisp* [2nd edition]
Addison Wesley, Reading, MA. 1984
Lots of AI code examples. The 2nd edition uses Common Lisp.

Wilensky, Robert *Common LispCraft*
Norton & Co. New York, NY. 1986
The last two chapters describe a pattern matcher and an associative data base.

Artificial Intelligence

Buchanon, Bruce & Edward Shortliffe *Rule-Based Expert Systems*
Addison-Wesley, Reading, MA. 1985
Describes the Mycin experiments of the Stanford Heuristic Programming Project. A classic.

Charniak, Eugene , Christopher Resbeck, Drew McDermott *Artificial Intelligence Programming*
Lawrence Erlbaum Assoc. Hillsdale, NJ. 1980
The original version came out in 1980 and used UCI Lisp in the examples. An upcoming 2nd edition uses Common Lisp.

Harmon, Paul & David King *Expert Systems*
John Wiley & Sons, NY, NY. 1987
283 pages. Lots of graphics.

Shapiro, Stuart *Encyclopedia of Artificial Intelligence*
John Wiley & Sons New York, NY. 1987
Large collection of articles on diverse topics in AI. Good as a glossary.

Macintosh

Apple Computer, *Macintosh User Guide*
Apple Computer, 1987
The 200 page manual that comes with every Macintosh. Explains the basics of using the machine.

Apple Computer, *Inside Macintosh* volumes 1, 2, 3, 4, 5
Addison Wesley, Reading, MA. 1985-87
Technical reference for system-level Macintosh programming. Difficult but sometimes necessary. Excellent as reference.

Apple Computer, Macintosh Technical Notes
Apple Computer, ongoing
Technical notes describing system-level details, caveats, bugs, and work-arounds. Available through Apple Programmer's and Developer's Association, 290 SW 43rd Street, Renton, WA 98055. (206) 251-6548.

Chernicoff, Stephen *Macintosh Revealed* volumes 1, 2
Hayden Book Co., Hasbrouck Heights, NJ. 1986
Introduction to Macintosh-style event driven programming, with windows, menus, etc. Good introduction, but many concepts may not be applicable to programming in Allegro CL.

Knaster, Scott *How to Write Macintosh Software*
Hayden Book Co., Hasbrouck Heights, NJ 1986
510 pages. Advanced topics in Macintosh programming, including the handling of some tricky situations.

Glossary

Glossary

ancestor — An object from which another object inherits, directly or indirectly. If foo inherits from bar, then bar is an ancestor of foo. In addition, if bar inherits from quux, then quux is an ancestor of both bar and foo. See also child, descendent, parent. Pages 3-3 to 3-9.

buffer — A Fred data type that holds a sequence of characters, as does a string. Buffers are optimized for efficient manipulation of text, including insertion and deletion. Buffers are displayed in Fred windows. Pages 1-1, 9-1

caret — See insertion point. See also cursor.

cell — A component of a table dialog. Page 7-11

class — An object from which other objects are created. Conceptually, it represents a category, such as dog, whereas an instance represents a specific example of a category, such as Lassie. Page 3-9.

command key — The key used for invoking menu-items through keyboard equivalents. In Macintosh editing mode, clover. In Emacs editing mode, shift clover. Page 2-3.

command table — A table of Fred keystrokes and pointers to the functions they invoke. Page 9–10.

comtab — See command table

control key — The key used to invoke Fred commands. In Macintosh editing mode, shift-clover. In Emacs editing mode, clover. Page 2-3.

cursor — The screen image corresponding to the mouse. As the mouse moves, the cursor moves on the screen. This is distinguished from the caret or insertion point that indicates the position in text where typed characters will appear. Pages 2-2, 8–6.

datum, Lisp A Lisp data object such as a list, array, number, character, string, etc. The words datum and data, rather than the traditional 'object' , are used to avoid possible confusion with the word object in object-oriented programming.

descendent An object which inherits, directly or indirectly from another object. If foo inherits from bar, then foo is an descendent of bar. In addition, if bar inherits from quux, then foo is as a descendent of both bar and quux. Pages 3-3 to 3–9.

device The component of a pathname that specifies the physical storage device, such as a hard disk. On the Macintosh this is the first part of the directory specification. Equivalent to a Macintosh volume. Page 10-1.

dialog A window containing dialog items, i.e., controls, which allows formatted interaction between a program and a user. Page 7-1.

dialog, modal A type of dialog that requires a response from the user before the user can perform any other action. Page 7-1.

dialog, modeless A type of dialog that can remain on the screen while the user performs other actions. Page 7-1.

dialog-item An element of a dialog that may cause an action when clicked by the mouse. When dialog-items are disabled they are shown in a grayed out format and do not respond to mouse clicks. Types of dialog-items include: button, check box, radio button, static text, editable text, and table. Page 7-1.

directory The component of the pathname that specifies the file directory. A Macintosh folder. Directory specifications end in a colon, for example "letters:", or a semi-colon in the case of logical directories, for example "ccl;". See also logical pathnames. Page 10-2.

Emacs mode The operating mode of Allegro in which the clover key is used as a control key and shift-clover is the command key. Page 1-2.

Emacs-style editor An editor whose command set is extensible by the user and which is extremely rich in keyboard commands. Emacs editors follow a set of conventions for the mapping between keystrokes and functionality. FRED is an Emacs-style editor. Page 1-1.

event An occurance outside of normal program flow, usually generated by the user. Examples include keystrokes, mouse clicks, and floppy disk inserts. Pages 1–2, 8-1.

event-handler A piece of code that responds to and processes events. Page 8-1.

file name The component of a pathname that specifies the name of the file. For example: "init". Page 10-2.

file type The component of a pathname that specifies the type of file. For example: "fasl". Page 10-2.

font-spec Specifies some or all of the characteristics of a font, including name, size, type, transfer mode. For example: ("monaco" 12 :BOLD) Pages 4-1, 4-2.

Fred The Emacs-style editor in Allegro. Pages 2-1, 9-1.

frame The local environment of an object, containing its variable and function bindings. The complete environment of an object consists of all the frames of all its ancestors and the global, or top-level Allegro environment. Frame is also used to denote stack-frame, a different data structure that contains the information for an active function (primarily lexical values). Page 3-5.

host The component of the pathname that specifies an operating system. For example: "coral-macs!". The host component of a pathname is currently ignored by Allegro CL. Page 10-1.

inheritance A mechanism whereby objects are linked in a heterarchy such that lower level objects can use procedures and variables defined in higher level objects. Pages 3-1, 3-3 to 3-9.

inheritance, multiple An inheritance mechanism in which an object may have multiple parents. The resulting structure can therefore be a heterarchy, which is more complex than a simple tree; it cannot be a general net since links flow only in one direction. Page 3-7.

insertion point The location between two characters where text may be entered into a window. See also cursor. Pages 1-1, 2-2.

inspector A tool that permits easy interactive examination of data structures. Page 11-2.

instance An object, particularly when it is being referenced in relation to its class. For example, "win-1 is an instance of the class *window*". Page 3-9.

instance variables Variables which maintain the state of an instance, usually defined when the instance is created. Page 3-9.

kill-ring A buffer in which killed strings are saved. It is integrated with the Macintosh keyboard and may be thought of as a multi-level clipboard. The top item in the kill ring is the equivalent to the Macintosh clipboard. The full kill-ring is accessable through a dialog. It is stored as a list of strings bound to the variable `*killed-strings*`. Pages 1-2, 9-2.

Listener A special window designed for interacting with Lisp. Allegro CL's listener includes such features as the blinking of parentheses to highlight matching pairs, and the use of different type styles to differentiate between what the user types in and Allegro CL's response. Page 1-1.

logical pathname An alias for a physical (or regular) pathname. Facilitates writing portable code with embedded file names. Page 10-8.

Macintosh mode The operating mode of Allegro in which the clover key is used as a command key and shift-clover is the control key. Page 1-2.

mark A pointer into a position in a buffer. Page 9-1.

menu On the Macintosh, the term menu denotes a pull-down menu i.e. a list of items from which the user can choose to initiate an action. Page 5-1

menu-item An element of a menu that when selected by mouse or keyboard equivalent causes an action to occur. When menu-items are disabled they are shown in a grayed out format and do not respond to user selection. Page 5-1

menubar A list of the titles of several menus. The standard Macintosh interface displays the menubar as a horizontal bar at the top of the screen. Clicking in the menubar displays the menus. Page 5-1

meta key — A modifier key used for invoking Fred commands. The option key is used as the meta key. In the Fred documentation, "meta" is shown as `M-` in the commands which use it. For example, the delete word command, is shown as `M-d`, where `M` is the meta key and `d` is the "d" key. Page 2-3.

mode line — The first non-blank line in a text file, used for giving the Lisp system information on the file. In Allegro CL, only the package specification is recognized. For example:

```
;;; -*- package: <package-name>  -*-
```

Page 2-1.

modifier keys — Keyboard keys that do not register as keystrokes but which modify the value of other keys. Examples include shift and clover. Page 1-2.

object — A data type that may contain data and/or procedures for handling that data. Objects also inherit data and procedures from other objects. Page 3-1.

object license — The integer that uniquely identifies an object to Allegro CL. License numbers are regenerated each time Allegro CL is loaded, and as such may change from session to session. Page 2-19.

parent — The immediate ancestor of an object. Page 3-3.

pathname — A method of specifying a particular file or directory. The Common Lisp standard pathname has six components: host, device, directory, filename, type, and version number. In Allegro CL the host, device and version number are ignored. Page 10-1.

pointer — A datum which is an address into memory. Page 12-8.

search path — The sequence of directories to be searched when the file system is looking for the physical pathname of a partially specified file. Page 10-7.

stack-backtrace — A debugging tool that examines the stack, showing stack-frames and their contents. Page 11-2.

stepper — A debugging tool that evaluates code one expression at a time, displaying all intermediate values. Page 11-2.

undo mechanism — The ability to reverse an operation, or series of operations, restoring the former state. Page 6-5.

version The component of a pathname that specifies which version of a particular file. Not used in Allegro CL. Page 10-2.

wildcard characters Characters that match to multiple other characters. For example, pathnames may contain a "*" to refer to a set of files. Page 10-7.

Index

#@,
use in creating points 4–1
as an Allegro reader macro character A–1
#@(1,1), as the normal setting for pen–size C–7
68000 A–trap instruction, as argument to stack-trap and register-trap 12–5
A–trap instruction, 68000, as argument to stack-trap and register-trap 12–5
:a0 thru :a6, as register–keyword for register trap macros 12–4
abort (command–period), use in terminating a modal dialog 7–2
abort-character, use in defining an abort key 1–2
aborting Lisp operations 1–2, 2–6
accessing,
Fred commands 9–11
function, source code for a 2–3
memory 12–7
object, current 3–17
object, function binding in a 3–15
object, license number for a 3–19
the optional Macintosh character set 2–5
record,
fields 13–5
window, on the Macintosh heap 6–5
symbol, object value of a 3–15
windows, 6–1
open 1–6
by title 6–1
action, as a property of dialog–items 7–1
action to be taken when,
dialog–item, initializing 7–6
dialog–item, is selected 7–6
activate window event, handler for 8–2
activating,
a dialog window 7–3
a modal dialog 7–2
windows 1–6, 6–5
active editor window
buffer,
evaluating and compiling the entire 1–5
finding a definition in 1–5
closing 1–3
evaluating the current selection in the 1–4
active window,
copying selected regions of the 1–4
deleting
selected regions of the 1–4
without saving the selected region from 1–4
inserting clipboard contents into the 1–4
printing 1–4
replacing selections in the 1–4
saving 1–3
searching in the 1–4
selecting the entire contents of 1–4
add-dialog-items, dialog function description 7–4
add-menu-items, menu function description 5–4
add-points, function description 4–2
add-self-to-dialog, dialog–item function description 7–8
adding
dialog–items
to a dialog 7–4, 7–8
to a menu 5–4
selections to the kill–ring 2–5
address registers, data type for 12–4
algorithm for representing Lisp data 12–1
Allegro,
Common Lisp
default directories 10–7
implementation concepts A–1
menu, system parameters B–2
environment, description of 1–1
exiting 1–4
installation disks contents ii–2
launching ii–2
memory management, division of memory in A–3
menubar,
as user customizable 1–3
Edit menu 1–4
Eval menu 1–4
File menu 1–3
Tools menu 1–5
as a pseudo multi–tasking system 1–2
as superset of Common Lisp ii–1
windows menu, component descriptions 1–6
allocating,
memory for rectangles C–2
space on a Macintosh heap for a Pascal record 13–4
static records, use of make–record for 13–1
:allow-returns, as keyword for exist editable–text–dialog–item function 7–9
allow-returns, editable–text–dialog–item variable description 7–9
altered editor window, detecting 2–2
ancestors, of an object, finding the 3–18
angle, calculating from a rectangle and a point C–9
Apple menu, item description 1–3
apple-menu, variable definition B–2
Application Heap,
allocating memory on 12–6
as storage area for Machintosh data 12–1

application HeapZone memory block, testing for a pointer to 12–9
applyhook, as supported by the standard evaluator A–5
apropos, as tool for locating Allegro elements 1–5
Apropos Tools menu item description 1–5
arc,
  drawing a border inside an C–13
  erasing an C–13
  filling an C–13
  inverting an C–13
argument list, for a function, obtaining the 2–3
arguments,
  to functions, lexical bindings as 3–12
  trap,
    characteristics of 12–2
    register, description of register–keyword for 12–4
    stack, description of type–keyword for 12–3
    type–checking not performed on 12–2
array,
  dialog–item,
    setting the dimensions of 7–16
    initializing 7–16
  how stored by Allegro A–2
  of pixels, bitmaps as rectangular C–16
array-dialog-item,
  as a dialog–item subclass 7–5
  as the array dialog–item class object 7–15
  variable description 7–15
arrow-cursor, variable description 8–7
ascii codes, how represented in Allegro A–1
ask,
  examples of use in manipulating objects 3–2
  macro description 3–14
  use contrasted with talkto 3–14
assigning
  function bindings of objects 3–15
  values to
    an object 3–15
    fields in a record 13–5
    variables in an objects, examples of 3–2
auto–refreshing graphics windows, as subclass of windows 6–1
backspace, affect of relative to cursor 2–2
BACKSPACE, use to delete character 2–5
backtrace dialog, controlling the display of B–3
Backtrace Tools menu item description 1–5
backtrace-on-break, variable definition B–3
backtracing, how to invoke 11–2
beeping sound, generating a 4–4
beeps, use in debugging 4–4
BiBOP, use in Allegro memory management A–3
binding,
  function,
    testing anywhere in the system 3–16
    testing current object 3–16
    testing hierarchy of current object 3–16
  inheritance rules 3–7
  inherited, examples of shadowing of 3–3
  lexical,
    as arguments to functions 3–12
    let statements as source of 3–12
    use in communicating values between instances 3–11, A–5
  lexically apparent, affect on incremental compilation 1–1
  object,
    as free references 3–12
    determining which will be used 3–5
    function,
      creating 3–16
      deleting 3–16
    shadowing by lexical bindings 3–11
    variable, deleting 3–16
  %stack-block, lexical scope and dynamic extent of 12–6
  value,
    testing anywhere in the system 3–16
    testing current object 3–16
    testing hierarchy of current object 3–16
  of variable and value in an object, examples of 3–2
bitmap,
  concepts and forms C–16
  copying C–16
  creating C–16
  Macintosh standard record type, as Allegro pre–defined 13–3
bits16, Macintosh standard record type, as Allegro pre–defined 13–3
black-pattern,
  as an Allegro pen pattern C–6
  as the normal setting for pen–pattern C–7
  variable definition B–4
:bold,
  as a font–style keyword 4–2
  as a menu–item font style 5–6
:boolean,
  as return–value–keyword for stack trap macros 12–3
  as type–keyword for stack trap macros 12–3
border, drawing,
  around a rectangle C–10
  inside a region C–16
  inside a rounded rectangle C–12
  inside an arc C–13

inside an oval C–11
bound-anywhere-p, function description 3–17
boundp, function description 3–16
boxing of fixnums 12–2
break, calls to, not tail–recursive A–6
break loop,
controlling the entry to B–3
obtaining a stack backtrace while in a 1–5
break-on-errors,
use to control affect of clover–period 1–2
variable definition B–3
break-on-warnings, variable definition B–3
:buffer, as keyword for exist *fred–window* function 9–6
buffer,
checking position argument for 9–2
as a component of a Fred window 9–1
as container for text being edited 9–1
creating 9–2, 9–3
current, printing information about 2–3
deleting characters from 9–5
editor, inserting strings from kill–ring into 1–4
evaluating and compiling the entire active editor window 1–5
finding
characters in a 9–5
a definition in the active editor window 1–5
a string in a 9–5
for a window 9–7
as a Fred data type 9–1
function description 9–2
inserting
a character in 9–4
a kill–ring string into 2–4, 2–5
location, use of marks in handling 9–1
moving
the cursor to end of 2–4
the cursor to start of 2–4
obtaining
a character from 9–4
the default mark for 9–2
a Lisp expression from 9–5
the modification count of 9–3
the number of lines in a 9–4
the property list of 9–4
the size of 9–3
a substring from 9–5
property list,
obtaining a property from 9–4
putting a property on 9–4
purging, closing the listener as means of 1–1
reading a file into 2–6, 9–6
replacing a character in 9–4
saving 2–6
selecting the entire 2–4, 9–10
testing for 9–2
writing
a file from a 9–6
to a file 2–6
buffer-bwd-sexp, function description 9–6
buffer-capitalize-region, function description 9–5
buffer-char, function description 9–4
buffer-char-pos, function description 9–5
buffer-char-replace, function description 9–4
buffer-column, function description 9–4
buffer-current-sexp-start-pos, function description 9–5
buffer-current-sexp, function description 9–5
buffer-delete, function description 9–5
buffer-downcase-region, function description 9–5
buffer-fwd-sexp, function description 9–6
buffer-getprop, function description 9–4
buffer-insert, function description 9–4
buffer-insert-file, function description 9–6
buffer-line, function description 9–4
buffer-line-end, function description 9–4
buffer-line-start, function description 9–4
buffer-mark, function description 9–2
buffer-modcnt, function description 9–3
buffer-not-char-pos, function description 9–5
buffer-plist, function description 9–4
buffer-position, function description 9–2
buffer-putprop, function description 9–4
buffer-size, function description 9–3
buffer-string-pos, function description 9–5
buffer-substring, function description 9–5
buffer-substring-p, function description 9–6
buffer-upcase-region, function description 9–5
buffer-word-bounds, function description 9–6
buffer-write-file, function description 9–6
bufferp, function description 9–2
button dialog–item class object 7–8

button-dialog-item,
as a dialog-item subclass 7–5
as button dialog-item class object 7–8
dialog-item, variable description 7–8
buttons, as dialog-item class 7–1
byte-offset, of a field within a record-type, obtaining the 13–4
bytes,
reading from memory 12–7
writing to memory 12–8
C, as symbol for control in editing Fred commands 2–3
C-\, Fred command that brings up Fred help window 2–3
C-=, Fred command that provides information about the current buffer 2–3
C-a, Fred command that moves cursor to start of line 2–4
C-b, Fred command that moves cursor back one character 2–4
C-d, Fred command that deletes characters to right of cursor 2–5
C-e, Fred command that moves cursor to end of line 2–4
C-f, Fred command that moves cursor forward one character 2–4
C-k, Fred command that kills rest of line or selection 2–5
C-m,
as Fred command useful for debugging 11–1
Fred command that macroexpands current expression 2–6
C-M-b, Fred command that moves cursor back one expression 2–4
C-M-BACKSPACE, Fred command that deletes backward delimiters 2–5
C-M-d, Fred command that deletes forward delimiters 2–5
C-M-f, Fred command that moves cursor forward one expression 2–4
C-M-k, Fred command that kills current expression 2–5
C-M-q, Fred command that reindents current expression 2–4
C-M-SPACE, Fred command that selects the current expression 2–6
C-n, Fred command that moves cursor down one line 2–4
C-o, Fred command that inserts new line without moving the cursor 2–4
C-p, Fred command that moves cursor up one line 2–4
C-q, Fred command that accesses the optional Macintosh character set 2–5
C-RETURN, Fred command that is equivalent to RETURN TAB 2–4
C-s, Fred command that brings up the search dialog box 2–6
C-t, Fred command that transposes two characters 2–5
C-v, Fred command that scrolls forward one screenful 2–4
C-w, Fred command that deletes the current selection and adds it to the kill ring 2–5
C-x C-a,
as Fred command useful for debugging 11–1
Fred command that prints the argument list for a symbol 2–3
C-x C-c, Fred command that compiles current expression 2–6
C-x C-d,
as Fred command useful for debugging 11–1
Fred command that prints the documentation string for a symbol 2–4
C-x C-e, Fred command that evaluates current expression 2–6
C-x C-i,
as Fred command useful for debugging 11–1
Fred command that inspects the current expression 2–4
C-x C-m,
as Fred command useful for debugging 11–1
Fred command that macroexpands current expression and pretty-prints it 2–6
C-x C-r,
as Fred command useful for debugging 11–1
Fred command that pretty prints current expression into the Listener 2–6
C-x C-s, Fred command that saves the top Fred window 2–6
C-x C-v, Fred command that gets a file 2–6
C-x C-w, Fred command that writes the top Fred window to a file 2–6
C-x h, Fred command that selects entire buffer 2–4
C-y, Fred command that yanks current kill ring string into buffer 2–4
calculating,
difference of two regions C–15
intersection
of two rectangles C–8
of two regions C–15
union of two rectangles C–9
with regions C–15
call-by-reference, trap arguments which are 12–2
call-by-value, trap arguments which are 12–2

:cancel,
as target of a throw from a Cancel dialog choice 4–3, 7–3
Cancel dialog choice, :cancel as the target of a throw from 4–3
capitalizing
the current word or selection 2–5
words 9–5
caps-lock-key-p,
function description 8–4
use during execution of menu–item–action 5–1
CCL::*name-char-alist*, how used for special characters A–1
CCL package, contents of A–2
cell contents,
drawing 7–13
obtaining 7–13
cell coordinates, table dialog, obtaining 7–14
cell-contents, table–dialog–item function description 7–13
cell-deselect, table–dialog–item function description 7–14
cell-position, table–dialog–item function description 7–14
cell-select, table–dialog–item function description 7–14
cell-selected-p, table–dialog–item function description 7–14
:cell-size, as keyword for exist table–dialog–item function 7–12
cell–size,
table dialog–item, initializing 7–12
table–dialog–item function description 7–13
cell-to-index, sequence–dialog–item function description 7–15
cell-to-subscript, array–dialog–item function description 7–16
cells, table dialog–item,
deselecting 7–14
obtaining a list of selected 7–14
selecting 7–14
testing for selected 7–14
cerror, calls to, not tail–recursive A–6
Change Font Edit menu item description 1–4
changed editor windows, detecting 2–2
changing
active window contents, using command–f for 1–4
current editable text for a dialog 7–4
default button for a dialog 7–4
font–spec for a window 6–4
size of a
dialog–item 7–7
window 6–4
title of a window 6–4
character,
deleting 2–5
from a buffer 9–5
downcasing 9–5
finding in a buffer 9–5
how represented in Allegro A–1
as immediate data 12–1
inserting in a buffer 9–4
moving the cursor
back one character 2–4
forward one character 2–4
obtaining from a buffer 9–4
replacing in a buffer 9–4
special, use of CCL::*name-char-alist* for naming A–1
transposing two 2–5
upcasing 9–5
check character, menu–item,
obtaining the 5–6
setting the 5–6
check mark, as component of menu–item 5–1
check-box-check, check–box–dialog–item function description 7–9
:check-box-checked-p, as keyword for exist check–box–dialog–item function 7–9
check-box-checked-p,
check–box–dialog–item function description 7–9
check-box-dialog-item,
as check box dialog–item class object 7–9
as a dialog–item subclass 7–5
variable description 7–9
check-box-uncheck,
check–box–dialog–item function description 7–9
check–box, as dialog–item class 7–1
:check-error, as keyword for register trap macros 12–4
checking a dialog–item's check–box 7–9
children, of an object, finding the 3–18
choose-file-dialog, function description 10–13
choose-font-dialog function description 4–3
choose-new-file-dialog, function description 10–13
choosing fonts 4–3
class, as a type of object 3–9
Clear Edit menu item description 1–4
clicks, mouse, handler for 8–1
clip-rect, window function description C–3
clip-region,
affect on drawing C–1
obtaining the current C–3

setting C–3
to be a rectangular region C–3
window function description C–3
clipboard,
for cutting and pasting between windows 2–2
as integrated with Fred kill–ring 1–2
kill–ring as extension to 9–2
as save area for
copied regions 1–4
deleted regions 1–4
as source for pasting operation 1–4
CLOS (Common Lisp Object System), as Common Lisp oops 3–1
close box,
as non–modifiable property of a window 6–2
list of which window types it is available on 6–2
window, initializing 6–2
Close File menu item description 1–3
`:close-box-p`, as keyword for `exist` window function 6–2
`close-region`, window function description C–15
`:closed`, as argument to return–from–modal–dialog 7–3
closed windows,
testing for 6–5
`wptr` as flag for 6–3
closing
a dialog window 7–3, 7–4
regions C–15
the active editor window 1–3
the listener, affects of 1–1
windows 2–2, 6–3
clover–. (clover–period), as abort key 2–6
clover key,
as command key for Macintosh mode 1–2
as control key 2–2
as control key for Emacs mode 1–2
clover–period (clover–.), use in stopping Lisp operations 1–2
clover–shift–1, Macintosh command that ejects the internal floppy disk 2–6
clover–shift–2, Macintosh command that ejects the external floppy disk 2–6
clover–shift–3, Macintosh command that creates a Macpaint file of the current screen 2–6
clover–shift–4, Macintosh command that prints the current screen 2–6
code, facilities to aid optimization of 12–1
coercing a pointer from a fixnum 12–9
`collapse-selection`, *fred–window* function description 9–8
command,
Fred,
accessing 9–11
defining 9–11
documentation, obtaining 9–11
key,
Emacs mode, shift–clover keys as 1–2, 2–2
Macintosh mode, clover key as 1–2, 2–2
tables,
copying 9–11
creating 9–11
command–a, use in selecting the entire contents of the active window 1–4
command–c, use in copying selected regions 1–4
command–e, use in evaluating the current selection in the active editor window 1–4
command–f, use in search and replace operations 1–4
`:command-key`, as keyword for `exist` menu–item function 5–5
`command-key`
equivalent, for menu–items, generating 5–5
menu–item function 5–6
`command-key-p`,
function description 8–4
use during execution of menu–item–action 5–1
command–l, use in activating the listener 1–6
command–n, use in creating an editor window for a new file 1–3
command–o, use in creating an editor window for an existing file 1–3
command–p, use in printing the active window 1–4
command–s, use in saving the active editor window 1–3
command–v, use in pasting the contents of the clipboard into the active window 1–4
command–x, use in deleting a selected region 1–4
command–z, use to support undo 1–4
comment marks, group, inserting 2–5
Common Lisp
clarifications, Allegro, as subject of manual ii–1
language extensions, Allegro, as subject of manual ii–1
the Language, as source of documentation for Allegro ii–1
communicating with objects, using ask for 3–14
compilation, automatic, turning off 1–1
Compile File Eval menu item description 1–5
`*compile-definitions*`,
as control for definition compilation A–5

use in controlling the action of command–e 1–4
use in controlling the actions of the Eval Buffer menu item 1–5
variable definition B–2
use in turning off automatic compilation 1–1
compile-file, Allegro keyword extensions to A–8
compiler,
Allegro as an incremental 1–1
characteristics of A–5
declarations A–7
optimization, controlling B–4
speed, controlling B–4
system parameters B–3
compiling
the current expression 2–6
definitions, vs evaluating, use of *compile-definitions* to control 1–4
the entire active editor window buffer 1–5
evaluator, as an evaluation option A–4
files 1–5
complexity, object–oriented programming as aid in managing 3–1
comtab,
as a Fred command table 9–10
copying 9–11
creating 9–11
fred–window variable description 9–11
comtab,
use in calculating editor keyboard commands 1–5
variable description 9–11
comtab-find-keys, function description 9–11
comtab-get-key, function description 9–11
comtab-key-documentation, function description 9–11
comtab-set-key, function description 9–11
:condense,
as a font–style keyword 4–2
as a menu–item font style 5–6
constant definitions, examining 1–5
constants, how compiled A–6
control key,
Emacs mode, clover key as 1–2, 2–2
Macintosh mode, shift–clover keys as 1–2, 2–2
physical, support for 1–2
control-key-p, function description 8–4
CONTROL-RETURN keys,
affect in the Lisp listener window 1–1
use in inserting line breaks 1–1
controlling the programming environment, parameters for 1–2
converting a
global point to a local point C–19
local point to a global point C–19
Copy (command–c) Edit menu item description 1–4
copy,
dialog function description 7–5
fred–window function description 9–10
copy-bits, function description C–16
copy-comtab, function description 9–11
copy-down, of text, use of RETURN and ENTER for 1–1
copy-file, function description 10–11
copy-record,
function description 13–5
use in copying records 13–1
copy-region, window function description C–14
copying
bitmaps C–16
command tables 9–11
the current selection onto the kill–ring 2–5
files 10–11
using Save as File menu item for 1–3
records 13–5
regions C–14
selected regions 1–4
text
between windows 2–2
from editable text dialog–items 7–5
create-file, function description 10–10
creating,
bitmaps C–16
buffers 9–2, 9–3
command tables 9–11
a default value record for a record type 13–2
an editor window for
a new file 1–3
an existing file 1–3
Fred commands 9–11
files 10–10
function bindings for objects 3–16
functions for objects 3–14
and initializing objects, function descriptions 3–13
logical pathnames 10–8
Macintosh pathnames 10–6
marks 9–2, 9–3
menu objects from the class *menu* 5–1
new objects, example of use of kindof in 3–1
object variable bindings 3–16
a Pascal record 13–4
pathnames 10–4
pictures C–17
points 4–1, 4–2
polygons C–18
a rectangle from two points C–9
rectangular regions C–14

a region C–14
static records, use of make–record for 13–1
temporary records 13–7
cropping, of drawing C–1
current
expression, meaning in the editing environment 2–3
object,
accessing 3–17
setting the 3–14
testing function and value bindings of 3–16
current-editable-text, dialog function description 7–4
current-event,
as event being handled 8–1
variable description 8–4
cursor,
handling the 8–7
Macintosh standard record type, as Allegro pre–defined 13–3
manipulation, concepts and forms 8–6
shape,
determining 8–7
updating 8–7
caret,
as flashing vertical bar between characters 2–2
hook for handling 8–7
located between two characters 2–2
mark, as a component of a Fred window 9–1
moving 2–2
back one character 2–4
back one expression 2–4
back one word 2–4
down one line 2–4
forward one character 2–4
forward one expression 2–4
forward one word 2–4
to end of line 2–4
to end of buffer 2–4
to start of buffer 2–4
to start of line 2–4
up one line 2–4
obtaining the mark for 9–7
cursorhook, variable description 8–7
cut, as function that deletes the current selection and adds it to the kill ring 2–5
Cut (command–x) Edit menu item description 1–4
cut,
dialog function description 7–5
fred–window function description 9–10
cutting,
copying, and pasting text between windows 2–2
selected regions 1–4
the selected region without saving 1–4
d, variable definition B–4
:d0 thru :d7, as register–keyword for register trap macros 12–4
dark-gray-pattern,
as an Allegro pen pattern C–6
variable definition B–4
data, as component of instances 3–9
formats,
font–specs as 4–1
points as 4–1
immediate,
alias for nil as 12–1
alias for t as 12–1
characters as 12–1
fixnum as 12–1
Macintosh pointers as 12–1
Lisp,
algorithm for representing 12–1
how different from Macintosh data 12–1
how represented 12–1
Macintosh, how different from Lisp data 12–1
registers, data type for 12–4
sharing between
32–bit long word and fixnums 12–2
Allegro and the Macintosh operating system 12–1
types, for address and data registers 12–4
deactivate window event, handler for 8–2
deactivating a
dialog window 7–3
a window 6–5
memory for a record, use of dispose-record for 13–4
records 13–5
static records, use of dispose-record for 13–1
debugging,
concepts and forms 11–1
obtaining a stack backtrace while 1–5
stepping through program execution 11–2
tracing
calls through the stack 11–2
objects 11–3
using
beeps in 4–4
the inspector to examine data structures 11–2
variable, use of B–4
declarations, how compiled A–7
decrementing pointers 12–9
def-fred-command, macro description 9–11
def-logical-pathname, function description 10–8

default button,
dialog window, initializing 7–3
for a dialog,
changing the 7–4
obtaining the 7–4
directory,
concepts and forms 10–6
finding 10–7
menubar, finding the 5–2
value, of a field within a record–type, obtaining the 13–4
:default-button, as keyword for exist dialog–window function 7–3
default-button, dialog function description 7–4
default-menubar, variable description 5–2
default-pathname-default, variable description 10–7
:defaults, as a keyword for make–pathname 10–4
defaults, instance variables, establishing 3–14
defining
Fred commands 9–11
functions for objects 3–14
new record types 13–2
objects 3–13
points 4–1
procedures which can be called by the Macintosh toolbox 12–10
definition,
in the active editor window buffer, finding a 1–5
record, as template that defines fields 13–1
in active editor window, evaluating and compiling 1–4
defobfun, example of use in defining object functions 3–5
defobject, macro description 3–13
defrecord,
macro description 13–2
use in defining new record types 13–1
defstruct accessor functions, how controlled by compiler A–7
deinstalling a
menu 5–3
menubar 5–2
delete-file, function description 10–10
deleting
all function bindings for objects 3–16
backward delimiters 2–5
characters 2–5
characters from a buffer 9–5
expressions 2–5
files 10–10
forward delimiters 2–5
Fred commands for 2–5
lines 2–5
a mark 9–3
menus 5–3
objects,
function bindings for 3–16
variable bindings for 3–16
pictures C–18
polygons C–18
records 13–5
regions C–14
selected 1–4
selected without saving 1–4
the current selection 2–5
static records, use of dispose–record for 13–1
text from editable text dialog–items 7–5
words 2–5
delimiters,
backward, deleting 2–5
forward, deleting 2–5
describe-record-field, use in finding information on a field of a record type 13–1
deselecting cells in a table dialog–item 7–4
desk accessories, use with Apple menu 1–3
:device, as a keyword for make–pathname 10–4
device, as component of pathname 10–1
devices,
function description 10–13
obtaining a list of 10–13
diagnostics, using beeps in 4–4
dialog,
as dialog window class object 7–2
variable description 7–2
dialog,
adding dialog–items to a 7–8
as a type of window 7–1
changing the current editable text for 7–4
choices, Cancel, :cancel as the target of a throw from 4–3
font, setting the 7–2
items, concepts and forms 7–5
Macintosh standard record type, as Allegro pre–defined 13–3
modal,
activating 7–2
characteristics of 7–1
returning from a 7–2
modeless, characteristics of 7–1
obtaining the
current editable text for 7–4
default button for 7–4
removing dialog–items from 7–8
window,
activating 7–3
closing 7–3, 7–4
initializing 7–3
as subclass of windows 6–1

dialog-item,
as an abstract class 7–5
as the dialog–item class 7–5
as generic class for dialog–item objects 7–1
use in obtaining a list of dialog–items 7–4
variable description 7–5
dialog–item,
as component of dialog 7–1
action to be taken when selected 7–6
button, class object 7–8
checking a check–box 7–9
drawing 7–7
editable text, class object 7–8
finding the dialog object that owns 7–7
Macintosh standard record type, as Allegro pre–defined 13–3
moving 7–6
obtaining the
font of 7–7
position of 7–6
size of 7–7
text of 7–7
setting the
font of 7–7
position of 7–6
size of 7–7
text of 7–7
static text, class object 7–8
subclasses, list of 7–5
testing for a checked 7–9
unchecking a check–box 7–9
:dialog-item-action, as keyword for exist dialog–item function 7–6
dialog-item-action, dialog–item function description 7–6
dialog-item-disable, dialog–item function description 7–7
dialog-item-draw, dialog–item function description 7–7
dialog-item-enable, dialog–item function description 7–7
:dialog-item-enabled-p, as keyword for exist dialog–item function 7–6
dialog-item-enabled-p, dialog–item function description 7–7
:dialog-item-font, as keyword for exist dialog–item function 7–6
dialog-item-font, dialog–item function description 7–7
:dialog-item-position, as keyword for exist dialog–item function 7–5
dialog-item-position, dialog–item function description 7–6
:dialog-item-size, as keyword for exist dialog–item function 7–6
dialog-item-size, dialog–item function description 7–7
:dialog-item-text, as keyword for exist dialog–item function 7–6
dialog-item-text, dialog–item function description 7–7
:dialog-items, as keyword for exist dialog–window function 7–3
dialog–items,
adding 7–4
adding to a dialog 7–8
classes of 7–1
dialog function description 7–4
disabled, characteristics of 7–1
disabling 7–6, 7–7
editable text,
copying text from 7–5
deleting text from 7–5
replacing text from 7–5
enabling 7–6, 7–7
finding 7–4
initializing 7–5
radio button 7–10
obtaining a list of 7–4
removing 7–4
from a dialog 7–8
testing for enabled 7–7
dialogs,
Allegro turnkey 4–3
concepts and forms 7–1
terminating 4–3
turnkey,
choose-font-dialog 4–4
get-string-from-user 4–3
message-dialog 4–3
y-or-n-dialog 4–3
diamond symbol, in About Allegro CL menu–item, as process indicator 1–2
difference-region, function description C–15
dimensions,
array dialog–items, initializing 7–16
table dialog–item,
initializing 7–12
obtaining 7–13
setting 7–13
directories, default, finding 10–7
directories-in-directory, function description 10–12
:directory, as a keyword for make–pathname 10–4
directory,
as component of pathname 10–2
home, finding 10–7

manipulation, concepts and forms 10–9
directory-namestring, function description 10–5
:disabled, as keyword for exist menu-item function 5–5
disabled
dialog-items, characteristics of 7–1
menus, characteristics of 5–1
disabling
dialog-items 7–6, 7–7
event handling 8–6
menu-items 5–5, 5–6
menus 5–4
disk,
ejecting 10–12
floppy, ejecting 2–6
insertion event, handler for 8–3
space requirements for running Allegro ii–1
dispose-record,
function description 13–5
use in deallocating memory for a record 13–4
use in deleting static records 13–1
dispose-region,
use in reclaiming regions C–14
window function description C–14
_disposhandle, use in releasing memory from the Macintosh Heap 12–6
disposing of records 13–5
_DisposPtr
/_NewPtr trap pair, as semantically equivalent to %stack-block
trap, use with zone pointers 12–9
use in releasing memory from the Macintosh Heap 12–6
do-all-objects, macro description 3–19
do-directories-in-directory, macro description 10–12
do-files-in-directory, macro description 10–12
do-object-functions, macro description 3–20
do-object-variables, macro description 3–20
:document,
as a window type used for modeless dialogs 7–1
as a window-type keyword for exist dialog-window function 7–2
as a window-type keyword for exist window function 6–2
:document-with-grow,
as a window type which can have a close box 6–2
as a window-type keyword for exist dialog-window function 7–2
as a window-type keyword for exist window function 6–2
:document-with-zoom,
as a window type which can have a close box 6–2
as a window-type keyword for exist dialog-window function 7–2
as a window-type keyword for exist window function 6–2
documentation string,
function, obtaining the 2–4
controlling the retention of B–2
Documents Tools menu item description 1–5
double click, testing for 8–3
double quotes, inserting 2–5
double-click-p, function description 8–3
double-click-spacing-p, function description 8–3
:double-edge-box,
as a window type used for modal dialogs 7–1
as a window-type keyword for exist dialog-window function 7–2
as a window-type keyword for exist window function 6–2
DOWNARROW, use to move cursor 2–4
downcasing
characters 9–5
the current word or selection 2–5
draw contents of window event, handler for 8–2
draw-cell-contents, table-dialog-item function description 7–13
draw-picture, window function description C–17
drawing
affect of moving the pen on C–7
cell contents of table dialog-item 7–13
commands, as object functions for windows C–1
cropping of C–1
a dialog-item 7–7
factors which affect C–1
how affected by state of the pen C–4
lines C–7
pictures C–17
text C–7
affect of current font specification on C–7
affect of windows as streams on C–7
dynamic extent, %stack-block bindings have 12–6
E-Machines' Big-Picture monitor, as supported by Allegro ii–1
ed-arglist, function description 2–3
ed-backward-char, function description 2–4
ed-backward-sexp, function description 2–4
ed-backward-word, function description 2–4

ed-beep function description 4–4
ed-beginning-of-buffer, function description 2–4
ed-beginning-of-line, function description 2–4
ed-capitalize-word, function description 2–5
ed-compile-top-level-sexp, function description 2–6
ed-copy-region-as-kill, function description 2–5
ed-delete-bwd-delimiters, function description 2–5
ed-delete-char, function description 2–5
ed-delete-fwd-delimiters, function description 2–5
ed-delete-word, function description 2–5
ed-downcase-word, function description 2–5
ed-end-of-buffer, function description 2–4
ed-end-of-line, function description 2–4
ed-eval-current-sexp, function description 2–6
ed-eval-or-compile-current-sexp, function description 2–5
ed-forward-char, function description 2–4
ed-forward-sexp, function description 2–4
ed-forward-word, function description 2–4
ed-get-documentation, function description 2–4
ed-help, function description 2–3
ed-indent-for-lisp, function description 2–4
ed-indent-sexp, function description 2–4
ed-insert-double-quotes, function description 2–5
ed-insert-killed-string-from-menu, function description 2–5
ed-insert-parentheses, function description 2–5
ed-insert-quoted, function description 2–5
ed-insert-sharp-comment, function description 2–5
ed-inspect-current-sexp, function description 2–4
ed-kill-line, function description 2–5
ed-kill-sexp, function description 2–5
ed-macroexpand-1-current-sexp, function description 2–6
ed-macroexpand-current-sexp, function description 2–6
ed-move-over-close-and-reindent, function description 2–4
ed-newline-and-indent, function description 2–4
ed-next-line, function description 2–4
ed-next-line, function description 2–4
ed-next-screen, function description 2–4
ed-open-line, function description 2–4
ed-previous-line, function description 2–4
ed-previous-line, function description 2–4
ed-previous-screen, function description 2–4
ed-read-current-sexp, function description 2–6
ed-rubout-char, function description 2–5
ed-rubout-word, function description 2–5
ed-select-current-sexp, function description 2–6
ed-self-insert, function description 2–4
ed-transpose-chars, function description 2–5
ed-upcase-word, function description 2–5
ed-what-cursor-position, function description 2–3
ed-yank, function description 2–4
Edit Definition,
  establishing the source file linkage for B–2
  Tools menu item description 1–5
Edit menu,
  Allegro menubar, item descriptions 1–4
  items,
    Change Font 1–4
    Clear 1–4
    Copy (command–c) 1–4
    Cut (command–x) 1–4
    Insert Killed String 1–4
    Paste (command–v) 1–4
    Search (command–f) 1–4
    Select All (command–a) 1–4
    Undo (command–z) 1–4
edit-menu, variable definition B–2
edit-select-file, as function that gets a file 2–6
editable text,
  as dialog–item class 7–1
  dialog–item
    class object 7–8
    initializing 7–9
  testing for permitted returns 7–9
editable–text item, use with get–string–from–user 4–3
editable-text-dialog-item,
  as a dialog–item subclass 7–5
  as editable text dialog–item class object 7–8
  dialog–item variable description 7–8
editing,
  Emacs style, using Fred for 2–1

Macintosh style, using Fred for 2–1
editor,
Allegro, as subject of manual ii–1
buffer,
changing fonts, font sizes, and font styles in 1–4
detecting when it has been changed 2–2
inserting strings from kill-ring into 1–4
how to tell which have been changed and not saved 1–6
commands, use in listener 1–1
keyboard commands,
obtaining a window of 1–5
use of *comtab* to calculate 1–5
window
buffer, evaluating and compiling the entire active 1–5
buffer, finding a definition in the active 1–5
closing the active 1–3
detecting a changed 2–2
evaluating the current selection in the active 1–4
for an existing file, creating an 1–3
new file, creating 1–3
:either,
affect on rref 13–6
as a value returned by record-storage 13–4
use
in defining new record types 13–2
with rref 13–5
with rset 13–5
eject-disk, function description 10–12
ejectedp, function description 10–12
ejecting
a disk 10–12
external floppy disk 2–6
internal floppy disk 2–6
elt, use in obtaining a pen-mode keyword C–7
Emacs
conventions, Macintosh-based exceptions 2–3
mode,
turning on and off B–3
typing keyboard equivalents in 5–1
use of clover key as control key 2–2
use of shift clover key as command key 2–2
style editing, using Fred for 2–1
Emacs-mode, clover key as control key for 1–2
emacs-mode, variable definition B–3
Emacs-style editor, as component of Allegro programming environment 1–1
empty-rect-p, function description C–9
empty-region-p, function description C–16
enable status, as component of menu-item 5–1
enabled
menu-item, testing for 5–6
menus, testing for 5–4
enabling
dialog-items 7–6, 7–7
menu-items 5–6
menus 5–4
encoded fixnums, points stored as C–2
enough-namestring, function description 10–5
ENTER key,
affect in the Lisp listener window 1–1
how different for RETURN key 1–1
use
for copy-down and evaluation 1–1
in evaluation expressions in an editor window 1–1
to evaluate current expression 2–5
environment,
global, as both global and root object 3–1
programming,
controlling 1–2
setting up 1–1
system parameters B–2
Environment Tools menu item
description 1–6
use in controlling the environment 1–2
eq, use in comparing points 4–1
equal-rect, function description C–9
equal-region-p, function description C–15
equality,
testing rectangles for C–9
testing regions for C–15
erase-arc, window function description C–13
erase-oval, window function description C–11
erase-polygon, window function description C–18
erase-rect, window function description C–10
erase-region, window function description C–16
erase-round-rect, window function description C–12
erasing a
polygon C–18
rectangle C–10
region C–16
rounded rectangle C–12
arc C–13
oval C–11
error,
calls to, not tail-recursive A–6

messages, characteristics of A–5
errors, register trap, signalling 12–4
escape character, pathname 10–3
Eval Buffer eval menu item description 1–5
Eval menu,
Allegro menubar, item descriptions 1–4
use in evaluating expressions in an editor window 1–1
item,
Compile File 1–5
Eval Buffer 1–5
Eval Selection (command–e) 1–4
Load 1–5
Eval Selection (command–e) eval menu item description 1–4
eval-enqueue,
function description 8–5
use by menu–items to initiate programs 5–1
eval-menu, variable definition B–2
evalhook, as supported by the standard evaluator A–5
evaluating
the current expression 2–5, 2–6
the current selection in the active editor window 1–4
the entire active editor window buffer 1–5
expressions, from an editor window 1–1
evaluation,
diamond symbol in About Allegro CL menu–item as indicator of 1–2
of object variables, examples of 3–2
options, compiling evaluator as an A–4
results of, controlling the printing of B–2
use of RETURN and ENTER for 1–1
evaluator,
compiling, characteristics of A–4
standard, characteristics of A–4
event
concepts and forms 8–1
dispatcher 8–4
dispatching interval,
obtaining 8–4
setting 8–5
handler,
selecting 8–5
concepts and forms 8–1
handling,
disabling 8–6
queueing programs during 8–5
by top window 6–1
information function descriptions 8–3
interaction, disabled during menu–item–action 5–1
Macintosh standard record type, as Allegro pre–defined 13–3
management system, concepts and forms 8–4
processing,
as a non–interruptable task 1–2
handling of by run–time environment A–5
windows as handlers for 6–1
event-dispatch, function description 8–4
event-keystroke, function description 9–10
event-ticks, function description 8–4
eventhook, variable description 8–4
examining Lisp object definitions 1–5
examining the structure of Lisp objects 1–5
exist,
array–dialog–item function description 7–16
as an object–function that can be defined for a class 3–10
check–box–dialog–item function description 7–9
dialog function description 7–3
dialog–item function description 7–5
editable–text–dialog–item function description 7–9
fred–window function description 9–6
function description 3–13
menu function description 5–3
menu–item function description 5–5
as the Object Lisp protocol for initializing instances 3–9
radio–button–dialog–item function description 7–10
sequence–dialog–item function description 7–14
table–dialog–item function description 7–12
use by oneof 3–13
window function description 6–1
existing file, creating an editor window for an 1–3
exiting Allegro 1–4
expand-logical-namestring, function description 10–9
expand-logical-pathname, function description 10–8
expanding logical pathnames 10–8
expression
compilation, controlling B–3
compiling the 2–6
deleting 2–5
evaluating 2–5, 2–6
macroexpanding 2–6
moving
into the Listener 2–6
the cursor back one 2–4
the cursor forward one 2–4
pretty printing 2–6
selecting 2–6
in a editor window, evaluating 1–1

:extend,
as a font-style keyword 4-2
as a menu-item font style 5-6
extension, of a compiled file, .fasl as the 1-5
extent, dynamic, %stack-block bindings have 12-6
.fasl, as extension of a compiled file 1-5
fasl files, loading 1-5
fast-eval,
use in selecting the evaluator option A-5
variable definition B-3
fbound-anywhere-p, function description 3-17
fboundp, function description 3-16
:features, compile file keyword, as Allegro extension A-8
fhave, function description 3-16
field,
byte-offset of, obtaining for a record-type 13-4
as component of record 13-1
data type, record type, specifying 13-2
default value of, obtaining for a record-type 13-4
description,
obtaining for a particular field of a record-type 13-4
obtaining a list for a record-type 13-4
record type, specifying 13-2
field-type of, obtaining for a record-type 13-4
names,
record type, specifying 13-2
obtaining a list for a record-type 13-4
of a record-type, obtaining a description of 13-4
in a record,
setting the values of 13-5
accessing 13-5
obtaining the value of 13-5
type, record, chart of pre-defined types, lengths and defaults 13-2
values, use of rref to obtain 13-6
variant, as multiple field mapping mechanism 13-1
field-info, function description 13-4
field type, of a field within a record-type, obtaining the 13-4
file
author, obtaining 10-11
compiling 1-5
copying 10-11
creating 10-10
an editor window for 1-3
creation date,
obtaining 10-10
setting 10-10
deleting 10-10
loading 1-5
controlling message verbosity during B-2
locking 10-11
manipulation, concepts and forms 10-9
File menu,
Allegro menubar, item descriptions 1-3
item descriptions 1-3
menu items,
Close 1-3
New (command-n) 1-3
Open (command-o) 1-3
Open Selected 1-3
Page Setup 1-4
Print (command-p) 1-4
Quit 1-4
Revert 1-3
Save as 1-3
Save (command-s) 1-3
file
modification date,
obtaining 10-10
setting 10-10
moving 10-10
printing 1-4
reading into a buffer 2-6, 9-6
renaming 10-9
retrieving the last version of a 1-3
selected, opening 1-3
system
interface, concepts and forms 10-1
manipulation, concepts and forms 10-9
testing for
the existence of 10-10
being unlocked 10-11
type, Macintosh,
obtaining 10-11
setting 10-11
unlocking 10-11
using Save as File menu item for copying 1-3
volume reference number, obtaining 10-12
writing
from a buffer 9-6
the current buffer to a 2-6
file-author, function description 10-11
file-create-date, function description 10-10
file-locked-p, function description 10-11
file-menu, variable definition B-2
file-namestring, function description 10-5
file-write-date, function description 10-10
:filename, as keyword for exist *fred-window* function 9-6

filename,
as component of pathname 10–2
associated with a Fred window, obtaining 9–8
string, as a component of a Fred window 9–1
files-in-directory, function description 10–12
filetype, as component of pathname 10–2
fill-arc, window function description C–13
fill-oval, window function description C–11
fill-polygon, window function description C–19
fill-rect, window function description C–10
fill-region, window function description C–16
fill-round-rect, window function description C–12
filling a
polygon C–18
rectangle C–10
region C–16
rounded rectangle C–12
arc C–13
oval C–11
filters, creating through the manipulation of function inheritance 3–8
find-dialog-item, dialog function description 7–4
find-menu, function description 5–2
find-window, function description 6–1
finding
Allegro elements, using apropos for 1–5
characters in a buffer 9–5
the default
directory 10–7
pathname 10–7
size and position of Allegro windows B–1
storage of a record–type 13–4
a definition in the active editor window buffer 1–5
dialog–items 7–4
the dialog object that owns a dialog–item 7–7
the fonts that can be used with Allegro B–4
the home directory 10–7
the length of a record–type 13–4
a Lisp expression location 9–6
menu,
an installed 5–2
all 5–3
a menu–item 5–4
modules B–3
the number of pixels per inch B–1
objects 3–17
the ancestors of an 3–18
the children of an 3–18
function definitions in 3–17
the name of an 3–18
the parents of an 3–18
strings 2–6
in a buffer 9–5
a word location 9–6
fixnums,
boxing of 12–2
coercing a pointer from 12–9
as immediate data 12–1
how stored in Allegro A–1
transforming into a point A–1
unboxing of 12–2
working around the 31–bit limitation 12–2
floating point
numbers, how stored in Allegro A–1
processor, MC68881, how used by Allegro A–1
flush-volume, function description 10–12
fmakunbound, function description 3–16
fmakunbound-all, function description 3–16
font,
dialog,
setting the 7–2
window, initializing 7–3
dialog–item,
initializing 7–6
obtaining the 7–7
setting the 7–7
information, coded as font–specs 4–2
mode, current, affect on text drawing C–7
size,
current, affect on text drawing C–7
changing, in editor buffer 1–4
specifiers, use in drawing text C–7
style,
current, affect on text drawing C–7
menu–item, list of 5–6
menu–item, setting the 5–6
changing, in editor buffer 1–4
type, current, affect on text drawing C–7
using a font–spec to find information about a 4–3
window,
initializing 6–1
obtaining the 6–4
font-info, function description 4–3
font-list, variable
definition B–4
description 4–2
font-name, as component of font–spec 4–2
font-names, Allegro, list of B–4
font-size, as component of font–spec 4–2

font-spec,
concepts and forms 4–2
as data formats 4–1
definition 4–2
example of legal 4–2
use in
choosing fonts 4–3
determining the pixel width of a string 4–3
to find information about a font 4–3
initializing window fonts 6–1
intializing dialog-item fonts 7–6
testing for existing fonts 4–3
window,
changing the 6–4
obtaining the 6–4
font-style,
as component of font-spec 4–2
as component of menu-item 5–1
fontinfo, Macintosh standard record type, as Allegro pre-defined 13–3
fonts,
changing, in editor buffer 1–4
choosing 4–3
obtaining a list of all installed fonts 4–2
testing for existing 4–3
use in finding out what fonts are available 4–2
fownp, function description 3–16
frame, drawing
around a rectangle C–10
inside an arc C–13
inside an oval C–11
inside a region C–16
inside a rounded rectangle C–12
frame-arc, window function description C–13
frame-oval, window function description C–11
frame-polygon, window function description C–18
frame-rect, window function description C–10
frame-region, window function description C–16
frame-round-rect, window function description C–12
_FrameRoundRect stack trap, used in example in stack trap discussion 12–3
_FrameRoundRect trap, as example of use of %stack-block 12–6
frames, as object environments 3–5
Fred
the Allegro editor, using 2–1
command tables, concepts and forms 9–10
Commands Tools menu item description 1–5
commands, useful for debugging,
C-m 11–1
C-x C-a 11–1
C-x C-d 11–1
C-x C-i 11–1
C-x C-m 11–1
C-x C-r 11–1
m-. 11–1
default package 2–1
editor commands, use in listener 1–1
help window, bringing up 2–3
keyboard commands, obtaining a window of 1–5
kill-ring, integrated with clipboard 1–2
programming 9–1
window
components of 9–1
functions, concepts and forms 9–6
initializing 9–6
printing hardcopy of 9–9
reverting 9–9
saving 9–9
updating 9–7
use for on-screen editing 9–1
fred-window, as an object class 9–1
fred-window-position, variable definition B–1
fred-windw-size, variable definition B–1
free variables, object bindings as 3–12
front-window,
function description 6–1
use in finding the front window 5–2
ftype declarations, how compiled A–7
function binding,
anywhere in the system, testing 3–16
current object, testing 3–16
hierarchy of current object, testing 3–16
inheritance rules 3–7
object,
accesssing the 3–15
assigning 3–15
creating 3–16
deleting 3–16
functions
compiled definitions of, controlling the retention of B–2
declarations, how compiled A–7
definitions,
controlling compilation of B–2
examining 1–5
in objects, finding 3–17
loading on call, as memory optimization technique 1–2, A–3
naming 3–15
object,
defining and naming 3–14
rules for inheritance of 3–5

owned by an object, iterative processing of all 3–20
preloading, use of `preload-all-functions` for 1–2, A–4
purging, controlling the 1–2, A–4
redefinition, controlling messages about B–2
swapping,
as component of Allegro memory management system A–3
controlling the purging of A–4
preloading A–4
`fwhere`, function description 3–17
garbage collection,
affects on
Lisp operation 1–2
loaded functions 1–2, A–3
Allegro's algorithm for A–4
invoking 1–2
as a non–interruptable task 1–2
not performed for regions C–14
triggered by heap resizing 12–6
GC, as cursor symbol during garbage collection 1–2
`gc`, as procedure which manually invokes garbage collection 1–2
General Computer's HyperCharger, as supported by Allegro ii–1
general trap calls,
`register-trap` as a 12–5
`stack-trap` as a 12–5
when to use 12–5
generic pointer, how different from a zone pointer 12–9
`%get-byte`, function description 12–7
`%get-full-long`, function description 12–7
`%get-long`, function description 12–7
`%get-ostype`, function description 12–8
`get-picture`, window function description C–17
`get-pixel`, window function description C–19
`get-polygon`, window function description C–18
`%get-ptr`, function description 12–7
`get-record-field`, function description 13–5
`%get-safe-ptr`, function description 12–7
`%get-signed-byte`, function description 12–7
`%get-signed-word`, function description 12–7
`%get-string`, function description 12–7
`get-string-from-user` function description 4–3
`%get-word`, function description 12–7
`_GetResource` stack trap, used in example in stack trap discussion 12–4
getting
argument list for a function 2–3
author of a file 10–11
byte–offset of a field within a record–type 13–4
cell
contents of table dialog–item 7–13
coordinates in a table dialog 7–14
character from a buffer 9–4
command documentation 9–11
current
clip–region C–3
editable text for a dialog 7–4
menubar 5–2
pen–mode C–5
pen–pattern C–6
pen–state C–6
default
button for a dialog 7–4
mark for a buffer 9–2
value of a field within a record–type 13–4
value record for a record type 13–4
description of a field of a particular record–type 13–4
dimensions of table dialog–items 7–13
documentation string for a function 2–4
end position of a line 9–4
event dispatching interval 8–4
field–type of a field within a record–type 13–4
field values with `rref` 13–6
file
creation date 10–10
modification date 10–10
font of a dialog–item 7–7
highest license number 3–19
horizonal
dimension of an array dialog 7–16
coordinate of a point 4–1
keyboard equivalent for a menu–item 5–6
kill–ring menu 2–5
layer number of a window 6–5
license number of an object 3–19
line number 9–4
Lisp expression from a buffer 9–5
list of
all installed menus 5–2
dialog–items 7–4
existing windows 6–1
field descriptions for a record–type 13–4
field names for a record–type 13–4
installed menu–items 5–4
selected cells in a table dialog–item 7–14
Macintosh file type of a file 10–11

menu–item
check character of a menu–item 5–6
font style of a menu–item 5–6
title of a menu–item 5–5
modification count of a buffer 9–3
mouse position 8–3
number of lines in a buffer 9–4
origin of a window C–3
pixel C–19
position of a
dialog–item 7–6
mark 9–3
pen C–4
window 6–4
property
from a buffer property list 9–4
list of a buffer 9–4
pushed radio–button of a cluster 7–5
record pointer within a handle 13–6
scroll position of a table dialog 7–14
sequence from a sequence dialog–item 7–15
size of
a buffer 9–3
a dialog–item 7–7
the pen C–4
table dialog–items 7–13
a window 6–4
start position of a line 9–4
string representation of a point 4–1
substring from a buffer 9–5
text of a dialog–item 7–7
title of a
menu 5–3
window 6–4
value of fields in records 13–5
vertical
coordinate of a point 4–1
dimension of an array dialog 7–16
visible dimensions of table dialog–items 7–13
volume reference number for a file 10–12
global
binding, variable, use with objects 3–2
environment, as outermost frame for an object 3–5
object, global environment as a 3–1
point,
converting a local point to a C–19
converting to a local pont C–19
variables, setting the value of 1–6
`global-to-local`, window function description C–19
grafport,
as first field of a window record 13–6
Macintosh standard record type, as Allegro pre–defined 13–3
rectangle calculations not dependent on C–8
window functions as Quickdraw functions that depend on C–1
created when windows are created C–1
graphic operations on
rectangles C–10
polygons C–18
regions C–16
use of points in 4–1
`*gray-pattern*`, as an Allegro pen pattern C–6
`h-specifier`, array–dialog–item variable description 7–16
`:h-specifier`, as keyword for array–dialog–item function description 7–16
`:handle`,
as a value returned by record–storage 13–4
use
in defining new record types 13–2
with `rref` 13–5
with `rset` 13–5
handle,
as return value from an address register 12–4
obtaining a record pointer within a 13–6
passing to the Macintosh operating system 12–9
`rlet` not legal in allocating records stored as 13–2
testing for 12–9
`handlep`, function description 12–9
handler
for mouse clicks 8–1
selecting an event 8–5
event 8–1
hardcopy,
device, setting up options for 1–4
of the active editor window, obtaining 1–4
hardware requrements, for running Allegro ii–1
`have`, function description 3–16
heap,
Lisp, use in Allegro memory management A–3
application,
alocating memory on 12–6
use in Macintosh memory management A–3
Macintosh,
accessing the pointer to the window record on 6–5
allocating space for a Pascal record on a 13–4
system, use in Macintosh memory management A–3
differences between Application and Lisp 12–1
resizing of 12–6

HeapZone memory blocks, testing for a pointer to 12–9
help, Fred commands, descriptions 2–3
help, using the Documents tools menu item to obtain 1–5
hfs-volume-p, function description 10–12
hiding
  the pen C–4
  windows 6–5
hierachy of current object, testing function and value bindings of 3–16
highest-license-number, function description 3–19
home directory, finding 10–7
hook,
  for events 8–4
  for handling cursors 8–7
horizontal
  character position, Fred window,obtaining 9–8
  coordinate of a point, obtaining the 4–1
  scroll–bar, table dialog–item, initializing 7–12
:host, as a keyword for make–pathname 10–4
host, as component of pathname 10–1
host-namestring, function description 10–5
i-beam-cursor, variable description 8–8
I/O, windows as primary method for screen–related 6–1
IEEE double float, as Allegro floating point data type A–1
ignore declarations, how compiled A–7
implementation, Allegro A–1
%inc-pointer, function description 12–9
incremental compiler, Allegro as an 1–1
incrementing pointers 12–9
index-to-cell, sequence–dialog–item function description 7–15
indexing into a sequence dialog 7–15
inheritance,
  as uni–directional only 3–4
  object function, rules for 3–5
  of objects, testing 3–17
  of variables and procedures, by objects 3–3
  overlapping vs non–overlapping 3–7
  role of usual in implementing 3–8
  single parent, example of 3–3
inherited behavior, modification by objects 3–6
init file,
  use in creating a default initial environment 1–1
  use in preloading swappable functions A–4
  removal suggested for memory optimization ii–2
init.lisp file, use in launching Allegro ii–2
init–list, use in initializing instances 3–9
init-list-default, macro description 3–14
initializing
  array dialog–items 7–16
  dialog–items 7–5
  dialog windows 7–3
  editable text dialog–items 7–9
  Fred windows 9–6
  instance variables 3–9, 3–14
  instances 3–9
  a menu 5–3
  menu–items 5–5
  objects 3–13
  radio button dialog–items 7–10
  a sequence dialog–item 7–14
  table dialog–items 7–12
  windows 6–1
inline declarations, how compiled A–7
Insert Killed String Edit menu item description 1–4
inserting
  a character from a buffer 9–4
  clipboard contents into the active window 1–4
  the current kill–ring string into the buffer 2–4
  a kill–ring string from a menu into the buffer 2–5
  a new line 2–4
  a set of double quotes 2–5
  group comment marks 2–5
  parentheses 2–5
  strings from kill–ring into editor buffer 1–4
insertion,
  affect of relative to cursor 2–2
  Fred commands, descriptions 2–4
  point, affect of RETURN key on 1–1
insertion–point, caret as 2–2
inset-rect, function description C–8
inset-region, function description C–15
insetting
  rectangles C–8
  regions C–15
Inside Macintosh,
  as a source of documentation for Allegro ii–1
  as background for understanding low–level system interface 12–1
Inspect Tools menu item description 1–5
inspecting
  Lisp objects 1–5
  the current expression 2–4
inspector, window–based, how to invoke 11–2
installation disks, contents of ii–2
installed
  menu, finding 5–2

menu, testing for 5–3
menu–items, obtaining a list of 5–4
installing
Allegro ii–2
a menubar 5–2
menus 5–3
using `menu-install` for 5–1
instance variables,
as state variables for an instance 3–9
initializating 3–9, 3–14
instances,
as example of a class 3–9
how different from classes 3–9
use of lexical bindings for communicating values between 3–11
interactive programming, Lisp listener as window designed for 1–1
`intersect-rect`, function description C–8
`intersect-region`, function description C–15
intersection,
of a region and a rectangle, testing for C–15
of two
rectangles, calculating C–8
regions, calculating C–15
`invert-arc`, window function description C–13
`invert-oval`, window function description C–11
`invert-polygon`, window function description C–19
`invert-rect`, window function description C–10
`invert-region`, window function description C–16
`invert-round-rect`, window function description C–12
inverting
an arc C–13
an oval C–11
a polygon C–19
a rectangle C–10
a region C–16
a rounded rectangle C–12
`:italic`,
as a font–style keyword 4–2
as a menu–item font style 5–6
items, dialog window, initializing 7–3
`&key` arguments, optimization of A–5
key
typed, handler for 8–2
up event, handler for 8–2
keyboard commands,
obtaining a window of editor 1–5
use of `*comtab*` to calculate editor 1–5
keyboard equivalent,
as component of menu–item 5–1
menu–item,
action taken for 5–1
obtaining 5–6
setting 5–6
menu equivalents,
command–a 1–4
command–c 1–4
command–e 1–4
command–f 1–4
command–n 1–3
command–o 1–3
command–p 1–4
command–s 1–3
command–v 1–4
command–x 1–4
command–z 1–4
keystroke
events, handling 9–10
name, translating into a code 9–11
`keystroke-code`, function description 9–11
`keystroke-function`, *fred–window* function description 9–11
`keystroke-name`, function description 9–11
keystrokes,
as an event 6–1
as event handled by top window 6–1
`kill-mark`, function description 9–3
`kill-picture`, window function description C–18
`kill-polygon`, window function description C–18
kill–ring,
adding lines to 2–5
adding selections to 2–5
copying the current selection onto 2–5
as extension to Macintosh clipboard 9–2
Fred, integrated with clipboard 1–2
inserting strings into editor buffer from 1–4
list of strings for B–4
menu, obtaining 2–5
as a multi–level clipboard 1–3
string, inserting into the buffer 2–4
killed strings, list of B–4
`*killed-strings*`,
as kill–ring location 9–2
variable definition B–4
killing
a mark 9–3
menus 5–3
pictures C–18
polygons C–18
`kindof`,
called by `oneof` 3–9, 3–13
examples of use in creating new objects 3–1
function description 3–13
use by `defobject` 3–13

launching Allegro ii–2
layer number, window, obtaining the 6–5
LEFTARROW, use to move cursor 2–4
length,
of a record type field, specifying 13–2
record–type, determining 13–4
let,
rlet uses the same general form as 13–7
statements, as source of lexical bindings 3–12
Levco Prodigy–4, as supported by Allegro ii–1
lexical
bindings,
as arguments to functions 3–12
let statements as source of 3–12
use in communicating values between instances 3–11
closure objects, as full (upward) funags A–5
scope, %stack-block bindings have 12–6
lexically apparent bindings, affect on incremental compilation 1–1
Library:records.lisp, as file which contains standard Macintosh record type definitions 13–3
license number,
highest, obtaining 3–19
object, obtaining 3–19
function description 3–19
license-to-object, function description 3–19
light-gray-pattern,
as an Allegro pen pattern C–6
variable definition B–4
line breaks, in listener, use of CONTROL–RETURN keys for inserting 1–1
line,
deleting 2–5
drawing C–7
inserting a new 2–4
moving the cursor
down one 2–4
to end of 2–4
to start of 2–4
up one 2–4
number, obtaining 9–4
obtaining the
end position of a 9–4
start position of a 9–4
reindenting 2–4
window function description C–7
line-to, window function description C–7
lines in a buffer, obtaining the number of 9–4
lines-in-buffer, function description 9–4
Lisp
data,
algorithm for representing 12–1
as ineligible for passing to a Macintosh trap 12–2
how different from Macintosh data 12–1
how represented 12–1
expression,
how to select 2–2
inspecting the current 2–4
location, finding 9–6
obtaining from a buffer 9–5
reindenting 2–4
Heap, as storage area for Lisp data 12–1
listener window,
affect of CONTROL–RETURN keys in 1–1
affect of ENTER key in the 1–1
affect of RETURN key in the 1–1
as part of Allegro programming environment 1–1
characteristics and use 1–1
suspend event, handler for 8–3
reactivat event, handler for 8–3
objects, examining the structure of 1–5
operations, aborting 1–2
Operations Fred commands, descriptions 2–5
package, contents of A–2
lisp-pathname, function definition 10–5
List Definitions in buffer tools menu item description 1–5
listener-window-position, variable definition B–1
listener-window-size, variable definition B–1
listener,
closing, affects of 1–1
use of command–l in activating 1–6
use of editor commands in 1–1
moving the current expression into 2–6
listener-comtab, variable description 9–11
listener-size, variable definition B–1
:load, compile file keyword, as Allegro extension A–8
Load eval menu item description 1–5
load-on-call,
as component of Allegro memory management system A–3
as function loading mechnism 1–2
load-verbose, variable definition B–2
loading files 1–5
controlling the message verbosity during B–2
local point,
converting a global point to a C–19
converting to a global pont C–19

local-to-global, window function
description C–19
lock-file, function description 10–11
locking files 10–11
logical pathnames,
concepts and forms 10–8
creating 10–8
expanding 10–8
logical-pathname-alist, variable
description 10–8
:long,
as argument for defpascal 12–10
as return value type for defpascal
12–10
as return–value–keyword for stack trap
macros 12–3
long word data, 32–bit, sharing with traps 12–2
long–words,
reading from memory 12–7
writing to memory 12–8
low–level system calls, warning on the use of
12–1
M, as symbol for meta in editing commands
2–3
M-.,
Fred command useful for debugging 11–1
Fred command that displays source code for a
symbol 2–3
M-#, Fred command that inserts #| |# 2–5
M-", Fred command that inserts a set of double
quotes 2–5
M-(, Fred command that inserts a set of
parentheses 2–5
M-<, Fred command that moves cursor to
beginning of buffer 2–4
M->, Fred command that moves cursor to end of
buffer 2–4
M-), Fred command that moves cursor to right
of next right parenthesis 2–4
M-b, Fred command that moves cursor back one
word 2–4
M-BACKSPACE, Fred command that deletes
backward delimiters 2–5
M-c, Fred command that initial caps current
word or selection 2–5
M-d, Fred command that deletes words to the
right of the cursor 2–5
M-f, Fred command that moves cursor forward
one word 2–4
M-l, Fred command that downcases current word
or selection 2–5
M-U, Fred command that upcases current word
or selection 2–5
M-v, Fred command that scrolls back one
screenful 2–4
M-w, Fred command that pushes current selection
onto kill ring 2–5
M-y, Fred command that yanks from a kill ring
menu 2–5
mac-default-directory, function
description 10–7
mac-directory, function description 10–6
mac-file-creator, function description
10–11
mac-file-type, function description 10–11
mac-filename, function description 10–6
mac-namestring, function description 10–6
mac-pathname, function description 10–6
mac-pathname, reading a string as a A–1
mac-volume, function description 10–6
Macintosh
character set, optional, accessing 2–5, A–1
data,
how different from Lisp data 12–1
when needed 12–1
data types,
patterns 12–1
regions 12–1
windows 12–1
editor mode,
turning on and off B–3
use of clover key as command key 2–2
use of shift clover key as control key
2–2
Finder version 5.5, as system software for
Allegro ii–1
heap, allocating space for a Pascal record on
a 13–4
II, affect on Allegro speed ii–1
memory management, division of memory
in A–3
Memory Manager, Allegro facilities for
working with 12–6
mode, typing keyboard equivalents in 5–1
models, which run Allegro ii–1
operating system, sharing data between
Allegro and the 12–1
pathnames,
concepts and forms 10–5
creating 10–6
pointers, as immediate data 12–1
ROM calls, use of Macintosh data for
communication with 12–1
standard record types, list of Allegro
pre–defined 13–3
style editing, using Fred for 2–1
System version 4.1, as system software for
Allegro ii–1
toolbox, creating procedures which can be
called by 12–10
traps,
legal data types for passing 12–2
pointers returned by, use as Pascal
records 13–1
register compared with stack 12–2

Macintosh–mode, shift–clover as control key for 1–2
macro definition, examining 1–5
macroexpanding the current expression 2–6
MacWrite, relation to the Allegro editor, Fred 2–1
make-bitmap, function description C–16
make-buffer, function description 9–3
make-comtab, function description 9–11
make-mac-pathname, function description 10–6
make-mark, function description 9–3
make-pathname, function description 10–4
make-point, function description 4–2
make-record,
  as memory allocator for long–lived rectangles C–2
  function description 13–4
  record–types which should not be created by 13–5
  use in creating static records 13–1
makunbound, function description 3–16
makunbound-all, function description 3–16
map-point, function description C–19
map-polygon, function description C–20
map-rect, function description C–20
map-region, function description C–20
mapping
  points C–19
  polygons C–20
  rectangles C–20
  regions C–20
mark,
  creating 9–3
  deleting 9–3
  function description 9–2
  moving 9–3
  obtaining the position of 9–3
  reversing the direction of 9–3
  setting the position of 9–3
  testing if backward 9–3
mark-backward-p, function description 9–3
mark-position, function description 9–3
markp, function description 9–2
marks,
  creating 9–2
  as a Fred data type 9–1
  as pointers into buffers 9–1
  testing for 9–2
MC68881 floating point processor, how used by Allegro A–1
memory,
  accessing 12–7
  allocating on the stack 12–6
  allocation for rectangles C–2
  for a record, use of dispose-record for deallocating 13–4
  management,
    Allegro configuration A–3
    facilities for 12–6
    implementation A–1
    warnings on handling 12–6
  Pascal record
    as a block of 13–1
    as a pointer into 13–1
  reclaiming for closed windows 6–3
  releasing from Macintosh Heap 12–6
  requirements, for running Allegro ii–1
  stack, use of %stack-block to allocate 12–6
menu,
  as class from which menu objects are created 5–1
  variable description 5–3
menu–based programming tools, as components of Allegro programming environment 1–1
menu-deinstall, menu function description 5–3
menu-disable, menu function description 5–4
menu-dispose, menu function description 5–3
menu-enable, menu function description 5–4
menu-enabled-p, menu function description 5–4
menu-install,
  as function which installs menus in menubars 5–1
  menu function description 5–3
menu-installed-p, menu function description 5–3
menu–item,
  adding to a menu 5–4
  changeability in an installed menu 5–1
  check character,
    obtaining the 5–6
    setting the 5–6
  components of 5–1
  consequences of selecting 5–1, 5–5
  disabling 5–5, 5–6
  enabled, testing for 5–6
  enabling 5–6
  finding 5–4
  font style,
    obtaining the 5–6
    setting the 5–6
  generating command–key equivalents 5–5
  how user selects 5–1
  initializing 5–5
  keyboard equivalent,
    obtaining 5–6
    setting 5–6
  menu function description 5–4

as object in Allegro 5–1
obtaining a list of installed 5–4
removing from a menu 5–4
separating into groups 5–1
style, menu-item function 5–6
title,
as component of menu-item 5–1
obtaining 5–5
setting 5–5
updating 5–6
menu-item, variable description 5–5
:menu-item-action, as keyword for exist menu-item function 5–5
menu-item-action,
as function which is called when a menu-item is selected 5–1
disabling of event interaction during 5–1
menu-item function 5–5
menu-item-check-mark, menu-item function 5–6
menu-item-disable, menu-item function 5–6
menu-item-enable, menu-item function 5–6
menu-item-enabled-p, menu-item function 5–6
:menu-item-title, as keyword for exist menu-item function 5–5
menu-item-title, menu-item function 5–5
menu-item-update,
as context-sensitive function 5–1
menu-item function 5–6
:menu-items, as keyword for exist menu function 5–3
:menu-title, as keyword for exist menu function 5–3
menu-title,
changability in an installed menu 5–1
menu function description 5–3
menu-update, menu function description 5–4
menubar,
Allegro,
description of components 1–3
Edit menu, component descriptions 1–4
Eval menu, component descriptions 1–4
File menu, component descriptions 1–3
Tools menu, component descriptions 1–5
Windows menu, component descriptions 1–6
as component of standard Macintosh user interface 5–1
concepts and forms 5–2
default, finding the 5–2
deinstalling a 5–2
function description 5–2
as list of menus 5–1
obtaining the current 5–2
options, item descriptions 1–3
quickdraw point, finding B–1
setting 5–2
menubar-bottom, variable definition B–1
menus,
adding menu-items to 5–4
concepts and forms 5–1
deinstalling 5–3
deleting 5–3
disabled, characteristics of 5–1
disabling 5–4
enabling 5–4
finding all 5–3
finding installed 5–2
function description 5–3
initializing 5–3
installing 5–3
as Macintosh style editing command access 2–1
modifying 5–4
as object in Allegro 5–1
obtaining a list of all installed 5–2
obtaining the title of 5–3
removing menu-items from 5–4
setting the title of 5–3
testing for the installation of 5–3
updating 5–4
testing for enabled 5–4
merge-pathname, function description 10–4
merging pathnames 10–4
message-dialog function description 4–3
messages, communicating to user with message-dialog 4–3
meta key, use of option key for 2–2
meta-point,
establishing the source file linkage for B–2
use in accessing source code for a function 2–3
using Edit Definition Tools menu item for 1–5
miscellaneous
Fred commands, descriptions 2–6
system parameters B–4
modal dialog,
activating a 7–2
characteristics of 7–1
use of :double-edge-box window-type in creating 7–1
modal-dialog,
function description 7–3
use in activating a modal dialog 7–2

mode
line, use in setting the package 2–1
pen,
obtaining the current C–5
setting the current C–5
modeless dialog,
characteristics of 7–1
as a normal window 7–2
when available 7–2
modification count, buffer, obtaining 9–3
modifying
menu–items 5–6
menus 5–4
modularity, as benefit of object–oriented programming 3–1
module-file-alist, use in finding a module A–3
module-file-alist, variable definition B–3
module-search-path, initial value of A–3
module-search-path, use in finding a module A–3
module-search-path, variable definition B–3
modules,
finding A–2
search path for locating B–3
system parameters B–3
mouse
clicks, handler for 8–1
down, testing for 8–3
position, obtaining 8–3
up event, handler for 8–2
mouse-down-p, function description 8–3
move, window function description C–7
move-file, function description 10–10
move-mark, function description 9–3
move-to, window function description C–7
movement Fred commands, descriptions 2–4
moving
the current expression into the Listener 2–6
the caret
back one character 2–4
back one expression 2–4
back one word 2–4
down one line 2–4
to end of buffer 2–4
to end of line 2–4
forward one character 2–4
forward one expression 2–4
forward one word 2–4
to start of buffer 2–4
to start of line 2–4
up one line 2–4
how to 2–2
a dialog–item 7–6
the difference of two regions C–15
files 10–10
the intersection of
two rectangles C–8
two regions C–15
a mark 9–3
polygons C–18
rectangles C–8
regions C–15
the union of
two rectangles C–9
two regions C–15
a window, how to 2–2
windows 6–4
multi–tasking system, pseudo, Allegro as a 1–2
MultiFinder, Allegro requirements under ii–2
multiple inheritance,
as a feature of Object Lisp 3–1
object, concepts and examples 3–7
my-dialog, dialog–item variable description 7–7
:name, as a keyword for make–pathname 10–4
name, of an object, finding 3–18
namestring, function description 10–5
naming
functions 3–15
objects 3–13
New (command–n) file menu item description 1–3
new file, creating an editor window for a 1–3
new–region, window function description C–14
_NewHandle trap, use in allocating memory on the Application Heap 12–6
_NewPtr/_DisposPtr trap pair, as semantically equivalent to %stack-block
_NewPtr trap,
use in allocating memory on the Application Heap 12–6
use in setting up zone pointers 12–9
next-license-to-object, function description 3–19
next-screen-context-lines, variable definition B–4
nfunction, special form description 3–15
nil alias, as immediate data 12–1
non–interruptable tasks,
event processing 1–2
garbage collection 1–2
non–overlapping multiple inheritance, concepts of 3–7
notinline declarations, how compiled A–7
:notPatBix, as a pen–mode C–5
:notPatCopy, as a pen–mode C–5
:notPatOr, as a pen–mode C–5
:notPatXor, as a pen–mode C–5

:novalue, as return-value-keyword for stack trap macros 12-3
null event, handler for 8-2
numbers, how stored in Allegro A-1
nx-fixnums-remain-fixnums, variable definition A-7, B-4
nx-inline-car-cdr, variable definition A-8, B-4
nx-open-code-in-line, variable definition A-7, B-4
nx-tailcalls,
  use in controlling tail recursion A-6
  variable definition B-3
nx-trust-declarations, variable definition A-7, B-4
object-ancestors, function description 3-18
object-children, instance variable description 3-18
object-license, as current object's indentifying number 3-14
object-license, function description 3-19
Object Lisp,
  as Allegro's object-oriented programming system 3-1
  concepts and forms 3-1
  extensions of Common Lisp functions,
    boundp 3-15
    fboundp 3-15
    fmakunbound 3-15
    makunbound 3-15
    set 3-15
    setq 3-15
    symbol-function 3-15
    symbol-value 3-15
    typep 3-15
  function descriptions 3-13
object-name, instance variable description 3-18
object-oriented programming, benefits and characteristics 3-1
object-parents, function description 3-18
objectp, function description 3-18
objects,
  accessing, function binding, 3-15
  bindings, as free references 3-12
  communication with, function descriptions 3-14
  creating 3-13
    function bindings in 3-16
  defining functions in 3-14
  deleting function bindings in 3-16
  environments, frames as 3-5
  finding, 3-17
    ancestors 3-18
    children 3-18
    function definitions in 3-17
    name 3-18
    parents 3-18
  global, global environment as 3-1
  hierarchy, rules for determining bindings in 3-5
  inheritance,
    multiple, concepts and examples 3-7
    overlapping, concepts of 3-7
    of variables and procedures by 3-3
  initializing 3-13
  inspecting 1-5
  internal procedures for, examples of 3-5
  iterative processing of all existing 3-19
  iterative processing of all functions owned by 3-20
  iterative processing of all variables owned by 3-20
  license number, obtaining 3-19
  manipulation, example of use of ask for 3-2
  managing
    bindings and definitions inside of, function descriptions 3-14
    function descriptions 3-17
  modification of inherited behavior by 3-6
  naming 3-13
  non-overlapping inheritance, concepts of 3-7
  printing 3-18
  oriented programming, tutorial on 3-1
  as programming units that combine procedures and data 3-1
  redefining 3-13
  root, global environment as the 3-1
  and scoping, concepts and examples 3-10
  setting
    the current 3-14
    values of 3-15
  testing
    function and value bindings of current 3-16
    function and value bindings of hierarchy of current 3-16
    inheritance of 3-17
  value, accessing 3-15
  variables,
    conflict with special variables 3-13
    creating bindings in 3-16
    deleting bindings in 3-16
    dynamic scope of 3-12
    examples of binding and assignment 3-2
    examples of evaluation 3-2
    and procedures in, local scoping of 3-1
offset-polygon, function description C-18
offset-rect, function description C-8
offset-region, function description C-15

oneof,
calling of exist by 3–9
function description 3–13
Open (command–o) File menu item description 1–3
Open Selected File File menu item descriptions 1–3
open–coding, of trap calls 12–5
open-region, window function description C–14
opening
an existing file 1–3
regions C–14
a selected file 1–3
operating system
traps, as register–based 12–2
types, writing to memory 12–8
optimization of code, facilities to aid the 12–1
optimize declarations, how compiled A–7
option key, as the meta key 2–2
option–d, as pathname escape character 10–3
option-key-p,
function description 8–4
use during execution of menu–item–action 5–1
origin,
setting C–3
window
function description C–3
obtaining C–3
:ostype,
as return–value–keyword for stack trap macros 12–3
as type–keyword for stack trap macros 12–3
:outline,
as a font–style keyword 4–2
as a menu–item font style 5–6
oval,
drawing a border inside an C–11
erasing an C–11
filling an C–11
inverting an C–11
overlapping multiple inheritance, concepts of 3–7
overriding default record storage, cautions on 13–6
ownp, function description 3–16
#p, as an Allegro reader macro character A–1
package, as default package for Fred 2–1
:package, as keyword for exist *fred–window* function 9–7
package,
associated with a Fred window,
obtaining 9–8
setting 9–8
contents of
CCL package A–2
LISP package A–2
USER package A–2
window, setting 2–1
Page Setup File menu item description 1–4
paint-arc, window function description C–13
paint-oval, window function description C–11
paint-polygon, window function description C–18
paint-rect, window function description C–10
paint-region, window function description C–16
paint-round-rect, window function description C–12
parent, single, example of inheritance from 3–3
parentheses, inserting 2–5
parents, of an object, finding the 3–18
Pascal
Quickdraw functionality, extension by Allegro C–1
records
formats, concepts and forms 13–1
passing around by Lisp 13–1
storing 13–1
use of pointers returned by Macintosh traps as 13–1
using 13–1
var
arguments, passing 12–10
parameters, call–by–reference arguments as 12–2
Paste (command–v) Edit menu item description 1–4
paste,
dialog function description 7–5
fred–window function description 9–10
pasting
text between windows 2–2
the contents of the clipboard into the active window 1–4
:patBix, as a pen–mode C–5
:patCopy,
as a pen–mode C–5
as the normal setting for pen–mode C–7
pathname-device, function description 10–5
pathname-directory, function description 10–5
pathname-host, function description 10–5
pathname-name, function description 10–5
pathname-type, function description 10–5
pathname-version, function description 10–5
pathnamep, function description 10–5

pathnames
Common Lisp, concepts 10–1
components 10–1
creating 10–4
escape character 10–3
finding the default 10–7
functions which construct 10–5
Lisp, testing for 10–5
logical,
creating 10–8
expanding 10–8
Macintosh,
concepts and forms 10–5
creating 10–6
reading a string as a A–1
merging 10–4
parsing examples 10–2
strings, parsing 10–1
using Lisp 10–4
:patOr, as a pen–mode C–5
pattern,
as component of pen state C–4
Macintosh
standard record type, as Allegro pre–defined 13–3
data type 12–1
pen,
how stored and accessed C–6
obtaining the current C–6
setting the current C–6
for use with Quickdraw calls B–4
:patXor, as a pen–mode C–5
pen,
attributes, diagram of C–4
as component of a window C–4
hiding C–4
how state affects drawing C–4
location, affect of pen–normal on C–7
moving C–7
obtaining the
current mode of C–5
current pattern of C–6
current state of C–6
position of the C–4
size of the C–4
patterns,
Allegro, list of C–6
how stored and accessed C–6
as records which must be explicitly disposed of C–6
routines, concepts and forms C–4
setting the
current mode of C–5
current pattern of C–6
current state of C–6
size of C–4
showing C–4
state, components of C–4
testing for visibility C–4
pen-hide, window function description C–4
pen-mode,
obtaining
as an integer C–7
the keyword for C–7
represented as an integer C–7
setting to normal C–7
window function description C–5
pen-modes, as pen–mode keyword list C–7
pen-modes, diagram of effects C–5
pen-normal, window function description C–7
pen-pattern,
setting to normal C–7
window function description C–6
pen-position, window function description C–4
pen-show, window function description C–4
pen-shown-p, window function description C–4
pen-size,
setting to normal C–7
window function description C–4
pen-state,
window function description C–6
as records which must be explicitly disposed of C–6
penstate, Macintosh standard record type, as Allegro pre–defined 13–3
physical control key, support for 1–2
pictures,
concepts and forms C–17
creating C–17
deleting C–18
drawing C–17
pixels,
obtaining C–19
bitmaps as rectangular arrays of C–16
how affected by pen–modes C–5
how different from points C–2
per inch, horizontal and vertical, finding B–1
pixels-per-inch-x, variable definition B–1
pixels-per-inch-y, variable definition B–1
:plain,
as a font–style keyword 4–2
as a menu–item font style 5–6
plane, window, initializing 6–2
point-h, function description 4–1
point-in-rect-p, function description C–9
point-in-region-p, function description C–15
point-string, function description 4–1

point-to-angle, function description C–9
point-to-cell, table–dialog–item function description 7–14
point-v, function description 4–1
:pointer,
as a value returned by record–storage 13–4
use in defining new record types 13–2
use with rref 13–5
use with rset 13–5
pointerp, function description 12–9
pointers,
coercing from a fixnum 12–9
decrementing 12–9
generic, how different from a zone pointer 12–9
incrementing 12–9
to the Macintosh Heap, as legal to pass to a Macintosh trap 12–2
to non–Lisp memory, testing for 12–9
to non–relocatable application HeapZone memory block, testing for 12–9
passing to the Macintosh operating system 12–9
reading from memory 12–7
returned by Macintosh traps, use as Pascal records 13–1
record, obtaining within a handle 13–6
to window record on the Macintosh heap, accessing the 6–5
writing to memory 12–8
zone, how different from a generic pointer 12–9
points,
creating 4–2
how components increase C–1
as data formats 4–1
how different from pixels C–2
how specified C–1
how to represent in function arguments 4–1
manipulation, efficiency of 4–1
mapping C–19
obtaining
a string representation of 4–1
the horizontal coordinate of a 4–1
the vertical coordinate of a 4–1
position, Fred window, obtaining 9–8
as representation for two–dimensional data 4–1
scaling C–19
stored
as 31–bit fixnums 4–1
as encoded fixnums C–2
transforming two fixnums into a A–1
use in describing the size of the pen C–4
points-to-rect, function description C–9
polygon,
concepts and forms C–18
creating C–18
deleting C–18
erasing a C–18
filling a C–18
inverting a C–19
graphics operations on C–18
mapping C–20
moving C–18
portrect,
affect on drawing C–1
window.portrect as abbreviation for 13–6
position
arguments, buffer, checking 9–2
dialog window, initializing 7–3
dialog–item,
initializing 7–5
obtaining the 7–6
setting the 7–6
pen, obtaining C–4
as a property of dialog–items 7–1
use in obtaining a pen–mode as an integer C–7
window,
initializing 6–1
obtaining the 6–4
preload-all-functions,
as part of environment control 1–2
function description A–4
preloading swappable functions A–4
pretty printing the current expression 2–6
Print (command–p) file menu item description 1–4
:print, compile file keyword, as Allegro extension A–8
Print Options Tools menu item
description 1–5
use in controlling the environment 1–2
print-length, use with print–record 13–5
print-level, use with print–record 13–5
print-record, function description 13–5
print-self, function description 3–18
printer paramenters, using Print Options Tools menu item to set 1–5
printing
the active window 1–4
a file 1–4
hardcopy of a Fred window 9–9
information about the current Fred buffer 2–3
objects 3–18
options, setting up 1–4
records 13–5
the screen 2–6
probe-file, function description 10–10
procedures,
as component of classes 3–9
inheritance by objects 3–3

object, scoping of 3–1
programming
environment,
Allegro, as subject of manual ii–1
description of Allegro 1–1
Fred, concepts and forms 9–1
property
-list, buffer,
obtaining 9–4
obtaining a property from 9–4
putting a property on 9–4
obtaining from a buffer property list 9–4
putting on a buffer property list 9–4
prototyping object hierarchies, how different from production setups 3–9
:ptr,
as argument for defpascal 12–10
as return value type for defpascal 12–10
as return–value–keyword for stack trap macros 12–3
as type–keyword for stack trap macros 12–3
purge-functions,
as part of environment control 1–2
function description A–4
purging,
of functions, controlling the 1–2, A–4
the buffer, closing the listener as means of 1–1
pushed-radio-button dialog function description 7–5
%put-byte, function description 12–8
%put-full-long, function description 12–8
%put-ostype, function description 12–8
%put-ptr, function description 12–8
%put-string, function description 12–8
%put-word, function description 12–8
putting a property on a buffer property list 9–4
Quckdraw arguments, why Allegro order can differ from Inside Macintosh C–2
QUED, relation to the Allegro editor, Fred 2–1
queueing programs during event handling 8–5
Quickdraw
code, available as an example file C–1
graphics, concepts and forms C–1
patterns for use with B–4
traps, calling directly 6–5
Quit file menu item description 1–4
quiting Allegro 1–4
radio button
dialog–items,
determining if pushed 7–10
determining the cluster for 7–10
initializing 7–10
pushing 7–10
testing for pushed 7–10
unpushing 7–10
obtaining the pushed 7–5
:radio-button-cluster, as a keyword for exist radio–button–dialog–item function 7–10
radio-button-cluster,
radio–button–dialog–item variable description 7–10
radio-button-dialog-item,
as a dialog–item subclass 7–5
radio button dialog–item class object 7–9
variable description 7–9
radio-button-push,
radio–button–dialog–item function description 7–10
:radio-button-pushed-p, as a keyword for exist radio–button–dialog–item function 7–10
radio-button-pushed-p,
radio–button–dialog–item function description 7–10
radio-button-unpush,
radio–button–dialog–item function description 7–10
radio–buttons, as dialog–item class 7–1
RAM cache, turning off suggested ii–2
reader macro
characters, Allegro A–1
#@, use in defining a point 4–1
#p, use for pathnames A–1
reading
bytes from memory 12–7
a file into a buffer 2–6, 9–6
long–words from memory 12–7
from memory, facilities for 12–7
operating system types from memory 12–8
pointers from memory 12–7
strings from memory 12–7
words from memory 12–7
real-font, function description 4–3
record-default,
function description 13–4
use in determining information about a record type 13–1
record-fields,
function description 13–4
use in determining information about a record type 13–1
record-info, function description 13–4
record-length,
function description 13–4
use in determining information about a record type 13–1
record-source-file, variable definition B–2
record-storage,
function description 13–4

use in determining information about a record type 13–1
`record-string`, use in obtaining a printed representation of a record 13–1
record types,
creating a default value record for 13–2
defining new 13–2
determining the default storage of 13–4
field
data type, specifying 13–2
description, specifying 13–2
length, specifying 13–2
name, specifying 13–2
functions which manipulate 13–4
length, determining 13–4
obtaining
the default value record for 13–4
a description of a field of 13–4
the byte–offset for a field within 13–4
the default value for a field within 13–4
the field descriptions for 13–4
the field names for 13–4
the field–type for a field within 13–4
option of the inspector, use in finding Macintosh record type definitions 13–3
Pascal, as guide to interpretation of 13–1
standard Macintosh, list of Allegro pre–defined 13–3
testing for a particular 13–4
`record-type-p`,
function description 13–4
use in testing for record types 13–1
records,
copying 13–5
created
by `rlet`, stored on stack 13–7
temporary 13–7
default value,
creating for a record type 13–2
obtaining for a record type 13–4
definition, as template that defines fields 13–1
fields,
accessing 13–5
setting the value of 13–5
memory for a , use of `dispose-record` for deallocating 13–4
pointer, obtaining within a handle 13–6
deleting 13–5
obtaining the value of fields in 13–5
Pascal,
as a block of memory 13–1
passing around by Lisp 13–1
record–type as guide to interpretation of 13–1
storing and using 13–1
use of pointers returned by Macintosh traps as 13–1
printing 13–5
setting the value of fields in 13–5
stored as handles, `rlet` not legal for allocating 13–2
structure of 13–1
summary of Allegro operations on 13–1
rect, Macintosh standard record type, as Allegro pre–defined 13–3
`rect-in-region-p`, function description C–15
rectangles,
allocation of memory for C–2
calculating C–8
an angle from a point and a C–9
functions, as global rather than object functions C–8
creating from two points C–9
drawing a border around C–10
erasing C–10
filling C–10
graphic operations on C–10
how specified C–2
how to specify without allocating memory C–2
insetting C–8
intersection of two, calculating and moving C–8
inverting C–10
mapping C–20
moving C–8
scrolling C–17
stored as eight–byte records C–2
testing for
the inclusion of a point C–9
the intersection of a region and a C–15
emptiness C–9
equality C–9
union of two, calculating and moving C–9
when record allocation is not needed C–2
rectangular
arrays of pixels, bitmaps as C–16
region, setting the clip–region to be a C–3
regions, creating C–14
redefining objects 3–13
`regbuf`,
Macintosh standard record type, as Allegro pre–defined 13–3
Pascal record type,
as target for `defpascal` argument 12–10
use of reset in setting 12–10
use of `rref` in accessing 12–10
regions,
calculating with C–15
closing C–15
as component of windows C–14

concepts and forms C–14
copying C–14
creating C–14
deleting C–14
difference of two, calculating and moving C–15
drawing a border inside a C–16
erasing a C–16
filling a C–16
graphics operations on C–16
how to select 2–2
insetting C–15
intersection of two, calculating and moving C–15
inverting a C–16
as a Macintosh data type 12–1
mapping C–20
moving C–15
opening C–14
rectangular,
creating C–14
setting the clip–region to be a C–3
testing for
the inclusion of a point C–15
the intersection of a rectangle and a C–15
emptiness C–16
equality C–15
union of two, calculating and moving C–15
register trap macros, general form, description 12–4
`register-trap`, macro description 12–5
register traps
as an Allegro facility for handling traps 12–2
compared with stack 12–2
errors, signalling 12–4
data types for 12–4
general format of 12–4
as a general trap call 12–5
registers,
address, data type for 12–4
data, data type for 12–4
reindenting
the current line 2–4
a Lisp expression 2–4
`remake-object`,
function description 3–13
use by defobject 3–13
`remove-dialog-items`, dialog function description 7–4
`remove-menu-items`, menu function description 5–4
`remove-self-from-dialog`, dialog–item function description 7–8
removing
dialog–items 7–4
dialog–items from a dialog 7–8
menu–items from a menu 5–4
`rename-file`, function description 10–9
renaming files 10–9
replacing
active window contents, using command–f for 1–4
a character in a buffer 9–4
the selected region of the active window with the contents of the clipboard 1–4
text from editable text dialog–items 7–5
representation of Lisp data 12–1
require, use in finding a module A–2
resize a window, how to 2–2
resources, Macintosh, accessing 12–4
retrieving last version of a file 1–3
RETURN key,
affect on insertion point in the Lisp listener window 1–1
affect on selected text 1–1
how different from ENTER key 1–1
use for copy–down and evaluation 1–1
use to insert new line 2–4
return value,
register trap, return–register–keyword as specifying the type of 12–4
stack trap, return–value–keyword as specifying the type of 12–3
`return-from-modal-dialog`,
macro description 7–3
use in returning from a modal dialog 7–2
`reverse-mark`, function description 9–3
reversing the direction of a mark 9–3
Revert File menu item description 1–3
reverting a
Fred window 9–9
window to last version saved 1–3
RIGHTARROW, use to move cursor 2–4
`rlet`,
function description 13–7
as memory allocator for short–lived rectangles C–2
not legal in allocating records stored as handles 13–2
use in allocating temporary records on a stack 13–1
ROM,
32–bit data fixnum requirement, working around 12–2
calls, use of Macintosh data for communication with 12–1
Macintosh, Allegro facilities for accessing 12–1
root object, global environment as the 3–1
rounded rectangle,
drawing a border inside a C–12
erasing a C–12

filling a C–12
inverting a C–12
rref,
as very efficient macro 13–6
macro description 13–5
preferred to get-record-field for obtaining field values 13–5
use in accessing
fields within records 13–1
Pascal regbuf record types 12–10
use with records created by toolbox traps 13–5
using in reading from memory 12–7
rset,
macro description 13–6
preferred to set-record-field for setting field values 13–5
use
in setting fields within records 13–1
in setting Pascal regbuf record types 12–10
with records created by toolbox traps 13–5
in writing to memory 12–7
run–time type checking, performed by system functions A–5
running Allegro ii–2
SANE, used by Allegro for floating point computation A–1
save area,
for copied regions, clipboard as 1–4
for deleted regions, clipboard as 1–4
Save
as file menu item description 1–3
(command–s) file menu item description 1–3
save-definitions, variable definition B–2
save-doc-strings, variable definition B–2
saving
the active window 1–3
the current buffer 2–6
a Fred window 9–9
the screen as a MacPaint file 2–6
scale-point, function description C–19
scaling points C–19
scope,
dynamic, of object variables 3–12
lexical, %stack-block bindings have 12–6
scoping,
object, concepts and examples 3–10
of object variables and procedures 3–1
:scratch-p, as keyword for exist *fred–window* function 9–6
screen,
height and width of, finding B–1
information system parameters B–1
printing 2–6
saving as a MacPaint file 2–6
scrolling
back one screen 2–4
forward one screen 2–4
screen-height, variable definition B–1
screen–related I/O, windows as primary method for 6–1
screen-width, variable definition B–1
scroll
a window, how to 2–2
position, table dialog, obtaining the 7–14
scroll-position, table–dialog–item function description 7–14
scroll-rect, window function description C–17
scroll-to-cell, table–dialog–item function description 7–14
scrolling
a table dialog 7–14
back one screen 2–4
forward one screen 2–4
rectangles C–17
search and replace dialog box, using command–f to access 1–4
Search (command–f) edit menu item description 1–4
search path,
for modules B–3
specifying 10–7
searching, using apropos as a tool for 1–5
Select All (command–a) edit menu item description 1–4
select-all,
as the function that selects the entire buffer 2–4
fred–window function description 9–10
selected regions, copying 1–4
selected regions, deleting 1–4
selected regions, replacing 1–4
selected text, affect of RETURN key on 1–1
selected-cells, table–dialog–item function description 7–14
selecting
cells in a table dialog–item 71–4
the current expression 2–6
the entire buffer 2–4
a Lisp expression, how to 2–2
menu–items, consequences of 5–1
a region 2–2
a window 2–1. 6–5
a word, 2–2
selection,
capitalizing the current 2–5
copying onto the kill–ring
deleting 2–5
downcasing the current 2–5

range,
as a component of a Fred window 9–1
Fred window, obtaining 9–8
Fred window, setting 9–8
type, table dialog–item, initializing 7–12
upcasing the current 2–5
selection-range, *fred–window* function description 9–8
:selection-type, as keyword for exist table–dialog–item function 7–13
self, function description 3–17
sequence
dialog, indexing into 7–15
dialog–item, initializing 7–14
sequence dialog–item,
obtaining 7–15
setting 7–15
sequence-dialog-item,
as a dialog–item subclass 7–5
variable description 7–14
:sequence-order, as keyword for sequence–dialog–item function 7–14
:sequence-wrap-length, as keyword for exist sequence–dialog–item function 7–15
set,
extended for Object Lisp 3–15
function description 3–15
set-cell-size, table–dialog–item function description 7–13
set-clip-region, window function description C–3
set-command-key, menu–item function 5–6
set-current-editable-text, dialog function description 7–4
set-cursor, function description 8–7
set-default-button, dialog function description 7–4
set-dialog-item-font, dialog–item function description 7–7
set-dialog-item-position, dialog–item function description 7–6
set-dialog-item-size, dialog–item function description 7–7
set-dialog-item-text, dialog–item function description 7–7
set-empty-region, window function description C–14
set-event-ticks, function description 8–5
set-file-create-date, function description 10–10
set-file-write-date, function description 10–10
set-h-specifier, array–dialog–item function description 7–16
set-mac-default-directory, function description 10–7
set-mac-file-creator, function description 10–11
set-mac-file-type, function description 10–11
set-mark, function description 9–3
set-menu-item-check-mark, menu–item function 5–6
set-menu-item-style, menu–item function 5–6
set-menu-item-title, menu–item function 5–5
set-menu-title, menu function description 5–3
set-menubar, function description 5–2
set-origin, window function description C–3
set-pen-mode, window function description C–5
set-pen-pattern, window function description C–6
set-pen-size, window function description C–4
set-pen-state, window function description C–6
set-record,
function description 13–5
use in setting multiple fields within a record 13–1
set-record-field, function description 13–5
set-rect-region, window function description C–14
set-region, use in changing coordinates of upper left hand corner of window C–1
set-selection-range, *fred–window* function description 9–8
set-table-dimensions, table–dialog–item function description 7–13
set-table-sequence, sequence–dialog–item function description 7–15
set-v-specifier, array–dialog–item function description 7–16
set-visible-dimensions, table–dialog–item function description 7–13
set-window-filename, *fred–window* function description 9–8
set-window-font,
use in setting the dialog font 7–2
window function description 6–4
set-window-layer, window function description 6–5

set-window-package,
  fred-window function description 9-9
  use in setting the package of a window 2-1
set-window-position, window function description 6-4
set-window-size, window function description 6-4
set-window-title, window function description 6-4
setq,
  extended for Object Lisp 3-15
  special form description 3-15
setting,
  a clip-region C-3
  clip-region to be a rectangular region C-3
  current
    pen-mode C-5
    pen-pattern C-6
    pen-state C-6
  dimensions of table dialog-items 7-13
  event dispatching interval 8-5
  file
    creation date 10-10
    modification date 10-10
  font of a dialog-item 7-7
  function bindings of objects 3-15
  horizonal dimension of an array dialog 7-16
  keyboard equivalent for a menu-item 5-6
  layer number of a window 6-5
  Macintosh file type 10-11
  menu-item
    check character 5-6
    font style 5-6
    title 5-5
  object values 3-15
  pen
    mode to normal C-7
    pattern to normal C-7
    size to normal C-7
  position of a
    dialog-item 7-6
    mark 9-3
    window 6-4
  printer parameters, using Print Options tools menu item for 1-5
  a sequence from a sequence dialog-item 7-15
  size of
    a dialog-item 7-7
    pen C-4
    table dialog-items 7-13
    a window 6-4
  text of a dialog-item 7-7
  title of a
    menu 5-3
    window 6-4
  value of
    fields in records 13-5
    global variables 1-6
    record fields 13-5
  vertical dimension of an array dialog 7-16
  visible dimensions of table dialog-items 7-13
  a window package 2-1
  up a menubar 5-2
setup-undo, *fred-window* function description 9-9
sfreply, Macintosh standard record type, as Allegro pre-defined 13-3
:shadow,
  as a font-style keyword 4-2
  as a menu-item font style 5-6
:shadow-edge-box,
  as a window-type keyword for exist dialog-window function 7-2
:shadow-edge-box, as a window-type keyword for exist window function 6-2
shadowing,
  of inherited bindings, example of 3-3
  of object bindings, by lexical bindings 3-11
sharing data between Allegro and the Macintosh operating system 12-1
shift-clover key,
  as command key for Emacs mode 1-2, 2-2
  as control key for Macintosh mode 1-2
shift-key-p,
  function description 8-4
  use during execution of menu-item-action 5-1
showing the pen C-4
:single-edge-box, as a window-type keyword for
  exist dialog-window function 7-2
  exist window function 6-2
size,
  buffer, obtaining 9-3
  as component of pen state C-4
  dialog window, initializing 7-3
  dialog-item,
    initializing 7-6
    obtaining the 7-7
    setting the 7-7
  pen,
    obtaining C-4
    setting C-4
  as a property of dialog-items 7-1
  table dialog-item,
    obtaining 7-13
    setting 7-13
  window,
    changing the 6-4
    initializing 6-1
    obtaining the 6-4

software configuration for Allegro ii–1
source code, for a function, accessing 2–3
special declarations, how compiled A–7
special variables, conflict with object variables 3–13
specialized table–dialog–items, concepts and forms 7–14
specifying a field
  data type for a record type 13–2
  description for a record type 13–2
  name for a record type 13–2
specifying the length of a record type field 13–2
:srcBic, as a transfer–mode keyword 4–2
:srcCopy, as a transfer–mode keyword 4–2
:srcOr, as a transfer–mode keyword 4–2
:srcPatBic, as a transfer–mode keyword 4–2
:srcPatCopy, as a transfer–mode keyword 4–2
:srcPatOr, as a transfer–mode keyword 4–2
:srcPatXor, as a transfer–mode keyword 4–2
:srcXor, as a transfer–mode keyword 4–2
stack trap macros, general form, description 12–3
stack,
  allocating memory on the 12–6
  as efficient temporary storage device 12–6
  backtrace,
    components of 11–2
    obtaining 1–5
  blocks, managing 12–6
  control, use in Allegro memory management A–3
  overflow, checked by compiled functions A–5
  space, use of tail–recursion to reduce A–6
  traps,
    compared with register traps 12–2
    data types for 12–3
    general format of 12–3
  use by
    rlet to store records 13–7
    Macintosh memory management A–3
stack, warning on use for temporary memory 12–6
%stack-block,
  special form description 12–7
  use in allocating storage on the stack 12–6
stack-trap,
  as a general trap call 12–5
  as an Allegro facility for handling traps 12–2
  macro description 12–5
standard evaluator, as an evaluation option A–4
standard-output, use in printing objects 3–18
start-picture, window function
  description C–17
start-polygon, window function
  description C–18
starting Allegro ii–2
state, pen,
  obtaining the current C–6
  setting the current C–6
static text, as dialog–item class 7–1, 7–8
static-text-dialog-item,
  as a dialog–item subclass 7–5
  as static text dialog–item class object 7–8
  dialog–item variable description 7–8
stepper, window–based, how to invoke 11–2
stopping Lisp operations, using clover–period (clover–.) for 1–2
:storage, use in
  overriding default storage type 13–2
  specifying the storage type for a record 13–4
storage, of a record–type, determining the default 13–4
storing Pascal records 13–1
stream-tyo, window function description C–7
streams,
  Fred windows as output 9–1
  use in printing objects 3–18
  windows as, affect on drawing text C–7
strings, 2–6
  finding in a buffer 9–5
  representation of a point, obtaining 4–1
  strings, from kill–ring, inserting into editor buffer 1–4
  passing to the Macintosh operating system 12–8
  reading from memory 12–7
  using
    Allegro turnkey dialog to obtain from user 4–3
    font–specs to determine the pixel width of 4–3
  writing to memory 12–8
string-width, function description 4–3
subscript-to-cell, array–dialog–item function description 7–16
subscripts, array dialog–items, initializing 7–16
substring, obtaining from a buffer 9–5
subtract-points, function description 4–2
swappable functions,
  as component of Allegro memory management system A–3
  controlling the purging of A–4
  preloading A–4
symbol-function,
  extended for Object Lisp 3–15
  function description 3–15

symbol-value,
  extended for Object Lisp 3–15
  function description 3–15
system
  HeapZone memory block, testing for a pointer to 12–9
  interface, low–level, concepts and forms 12–1
  parameters,
    Allegro Common Lisp menu B–2
    compiler B–3
    environment B–2
    miscellaneous B–4
    modules B–3
    screen information B–1
    variables which describe B–1
    window configuration B–1
t alias, as immediate data 12–1
TAB, use to reindent current line 2–4
:table-array, as keyword for exist array–dialog–item function 7–16
table-dialog-item,
  as a dialog–item subclass 7–5
  variable description 7–12
table dialog–items,
  concepts and forms 7–10
  initializing 7–12
  as a method for selecting items from a set 7–10
  as rectangles with a series of cells 7–11
:table-dimensions, as keyword for exist table–dialog–item function 7–12
table-dimensions, table–dialog–item function description 7–13
:table-hscrollp, as keyword for exist table–dialog–item function 7–12
:table-sequence, as keyword for sequence–dialog–item function 7–14
table-sequence, sequence–dialog–item function description 7–15
:table-subscript, as keyword for array–dialog–item function description 7–16
:table-vscrollp, as keyword for exist table–dialog–item function 7–12
tables, as dialog–item class 7–1
tail recursion, controlling B–3
tail–recursive, compiler as properly A–6
talkto,
  function description 3–14
  use contrasted with ask 3–14
temporary records, creating 13–7
terminating a modal dialog 7–2
terminating dialogs 4–3
testing
  a rectangle for the inclusion of a point C–9
  a region for the inclusion of a point C–15
  anywhere in the system, for function bindings in 3–16
  anywhere in the system, for value bindings in 3–16
  current object, for function bindings in 3–16
  current object, for value bindings in 3–16
  for
    a checked dialog–item 7–9
    a double click 8–3
    an enabled menu–item 5–6
    buffers 9–2
    enabled dialog–items 7–7
    enabled menus 5–4
    existence of a file 10–10
    existing fonts 4–3
    files being locked 10–11
    installation of a menu 5–3
    Lisp pathnames 10–5
    marks 9–2
    mouse down 8–3
    permitted returns in editable text 7–9
    selected cells in a table dialog–item 7–14
    the intersection of a region and a rectangle C–15
  function and value bindings anywhere in the system 3–16
  hierarchy of current object, for function bindings 3–16
  hierarchy of current object, for value bindings 3–16
  if a mark is backward 9–3
  inheritance, of objects 3–17
  objects 3–18
  rectangles for
    emptiness C–9
    equality C–9
  regions for
    emptiness C–16
    equality C–15
  windows for
    undo capability 6–6
    visibility 6–5
text,
  as a property of dialog–items 7–1
  current editable for a dialog,
    changing the 7–4
    obtaining the 7–4
  dialog, font of 7–2
  dialog–item,
    initializing 7–6
    obtaining the 7–7
    setting the 7–7

drawing C–7
affect of current font specification on C–7
drawing, affect of windows as streams on C–7
editable text dialog–item
class 7–1
copying 7–5
deleting 7–5
deleting 7–5
files,
compiling 1–5
loading 1–5
using the Documents tools menu item to obtain 1–5
static, as dialog–item class 7–1
windows, as subclass of windows 6–1
title,
accessing windows by 6–1
as component of menu–item 5–1
dialog window, initializing 7–3
menu–item,
obtaining 5–5
setting 5–5
of a menu,
obtaining 5–3
setting the 5–3
of undo menu–item, changing the 6–6
window,
changing the 6–4
initializing 6–1
obtaining the 6–4
toggle between full–screen and a smaller size 2–2
`:tool`, as a window type
which can have a close box 6–2
keyword for
`exist` dialog–window function 7–2
`exist` window function 6–2
toolbox,
Macintosh, creating procedures which can be called by 12–10
traps, as stack–based 12–2
Tools menu,
Allegro menubar, item descriptions 1–5
item descriptions 1–5
items,
Apropos 1–5
Backtrace 1–5
Documents 1–5
Edit Definition 1–5
Environment 1–6
Fred Commands 1–5
Inspect 1–5
List Definitions 1–5
Print Options 1–5
`*tools-menu*`, variable definition B–2
`*top-listener*`, variable definition B–1
tracing, how to invoke 11–3
`_TrackControl` trap, as example of use of `defpascal` 12–10
transfer–mode, as component of font–spec 4–2
transposing two characters 2–5
traps
Allegro facilities for handling 12–2
arguments, characteristics of 12–2
how to customize calls to 12–1
Macintosh, use of pointers returned by as Pascal records 13–1
not included in Allegro, how to use 12–1
Quickdraw, calling directly 6–5
register,
data types for 12–4
general format of 12–4
stack,
data types for 12–3
general format of 12–3
type–checking not performed on arguments to 12–2
tutorial on object oriented programming 3–1
`:type`, as a keyword for `make-pathname` 10–4
type
coercion, type–keyword as specifying argument 12–3
dialog window, initializing 7–3
window, initializing 6–2
type–checking, not performed on arguments to traps 12–2
typeface conventions, in the Lisp listener window 1–1
typep, function description 3–17
unboxed immediate data, as legal to pass to a Macintosh trap 12–2
unboxing
of fixnums 12–2
use in coercing a pointer from a fixnum 12–9
unchecking a dialog–item's check–box 7–9
`:underline`,
as a font–style keyword 4–2
as a menu–item font style 5–6
Undo (command–z) Edit menu item description 1–4
menu–item,
as trigger for undo window function 6–5
title, changing the 6–6
support, 9–9
concepts and forms 6–5
window function
description 6–5
`triggered by undo menu-item` 6–5
`*undo-menu-item*`,
undo menu–item as value of 6–6

variable definition B–2
union,
of two rectangles, calculating C–9
of two regions, calculating C–15
union-rect, function description C–9
union-region, function description C–15
unlock-file, function description 10–11
unlocking files 10–11
unwind-protect, use with unlocking of handles 12–9
UPARROW, use to move cursor 2–4
upcasing
characters 9–5
the current word or selection 2–5
update window event, handler for 8–2
update-cursor, function description 8–7
updating a Fred window 9–7
updating
the cursor shape 8–7
a menu 5–4
menu–items 5–6
user interface functions, file system 10–13
USER package, contents of A–2
user-homedir-pathname, function description 10–7
using Pascal records 13–1
usual,
example of use in modifying object functions 3–6
role in implementing inheritance 3–8
usual-exist, use by exist 3–13
v-specifier, array–dialog–item variable description 7–16
:v-specifier, as keyword for exist array–dialog–item function 7–16
value
bindings,
anywhere in the system, testing 3–16
current object, testing 3–16
hierarchy of current object, testing 3–16
in an object, accessing 3–15
object,
assigning 3–15
function bindings, assigning 3–15
var arguments, Pascal, passing 12–10
variables
assignment, in an object, examples of 3–2
binding,
in an object, examples of 3–2
inheritance rules 3–7
conflict between special and object 3–13
definitions, examining 1–5
global
binding of, use with objects 3–2
setting the value of 1–6
inheritance by objects 3–3
instance,
initialization of 3–9
as state variables for an instance 3–9
establishing default values 3–14
object,
creating bindings for 3–16
deleting bindings for 3–16
dynamic scope of 3–12
examples of evaluation of 3–2
scoping of 3–1
examples of multiple private versions 3–1
owned by an object, iterative processing of all 3–20
variant fields, as multiple field mapping mechanism 13–1
:verbose, compile file keyword, as Allegro extension A–8
verbose-eval-selection, variable definition B–2
:version, as a keyword for make–pathname 10–4
version, as component of pathname 10–2
vertical
character position, Fred window,obtaining 9–8
coordinate of a point, obtaining the 4–1
scroll–bar, table dialog–item, initializing 7–12
visibility, dialog window, initializing 7–3
visible
dimensions, table dialog–item,
initializing 7–12
obtaining 7–13
setting 7–13
making windows 6–5
:visible-dimensions, as keyword for exist table–dialog–item function 7–12
visible-dimensions, table–dialog–item function description 7–13
:void, as return value type for defpascal 12–10
volume-number, function description 10–12
warn, calls to, not tail-recursive A-6
warn-if-redefine, variable definition B-2
:warnings, compile-file keyword, as Allegro extension A-8
warnings
about use of heaps 12-1
on allocating memory on the stack 12-6
on memory management handling 12-6
on overriding default record storage 13-6
on the use of low-level system calls 12-1
watch-cursor, variable description 8-8
wfind, as function that brings up the search dialog box 2-6
where, function description 3-17

white-pattern,
as an Allegro pen pattern C-6
variable definition B-4
wildcards, using 10-7
window,
as a parent of dialog objects 7-1
as the superclass from which other windows inherit 6-1
variable description 6-1
window-activate-event-handler, window function description 8-2
window-based programming tools, as components of Allegro programming environment 1-1
window-buffer, *fred-window* function description 9-7
window-can-undo-p window function description 6-6
window-click-event-handler, window function description 8-1
window-close,
dialog function description 7-4
use in closing a dialog window 7-3
window function description 6-3
window-cursor, window variable description 8-7
window-cursor-mark, *fred-window* function description 9-7
window-deactivate, use when closing windows 6-3
window-deactivate-event-handler, window function description 8-2
window-disk-insert-event-handler, window function description 8-3
window-draw-contents, window function description 8-2
window-event, window function description 8-5
window-filename, *fred-window* function description 9-8
:window-font, as keyword for
exist dialog function 7-3
exist window function 6-1
window-font
init option, use in setting the dialog font 7-2
window function description 6-4
window-hardcopy, *fred-window* function description 9-9
window-hide, window function description 6-5
window-hpos, *fred-window* function description 9-8
window-key-event-handler, window function description 8-2
window-key-up-event-handler, window function description 8-2
:window-layer, as keyword for exist window function 6-2
window-layer, window function description 6-5
window-line-vpos, *fred-window* function description 9-8
window-mouse-position, window function description 8-3
window-mouse-up-event-handler, window function description 8-2
window-null-event-handler, window function description 8-2
window-package, *fred-window* function description 9-8
window-point-position, *fred-window* function description 9-8
:window-position, as keyword for
exist dialog function 7-3
exist window function 6-1
window-position,
use with dialogs 7-1
window function description 6-4
window-resume-event-handler, window function description 8-3
window-revert, *fred-window* function description 9-9
window-save,
as function that saves the top Fred window 2-6
fred-window function description 9-9
window-save-as,
as function that writes the top Fred window to a file 2-6
fred-window function description 9-9
window-select, window function description 6-5
window-select-event-handler, window function description 8-3
:window-show, as keyword for
exist dialog function 7-3
exist window function 6-1
window-show,
use with dialogs 7-1
window function description 6-5
window-shown-p, window function description 6-5
:window-size, as keyword for
exist dialog function 7-3
exist window function 6-1
window-size, window function description 6-4
window-start-mark, *fred-window* function description 9-7
window-suspend-event-handler, window function description 8-3
:window-title, as keyword for
exist dialog function 7-3
exist window function 6-1

window-title, window function description 6-4
:window-type, as keyword for
  exist dialog function 7-3
  exist window function 6-2
window-update, *fred-window* function description 9-7
window-update-cursor, window function description 8-7
window-update-event-handler, window function description 8-2
windows,
  activating 6-5
  accessing 6-1
    by title 6-1
    open 1-6
  Allegro, finding the default size and position of B-1
  buffer,
    evaluating and compiling the entire active 1-5
    finding a definition in the active editor 1-5
  close box as a non-modifialbe property of 6-2
  closed, testing for 6-5
  closing 6-3
    the active editor 1-3
  concepts and forms 6-1
  configuration, system parameters B-1
  copying selected regions of the active 1-4
  creation as trigger for grafport creation C-1
  deactivating 6-5
  deleting
    selected regions of the active 1-4
    without saving the selected region from the active 1-4
  dialog
    activating 7-3
    closing 7-3, 7-4
  drawing commands as object functions for C-1
  as example of record which should not be created with make-record 13-5
  evaluating the current selection in the active editor 1-4
  font-spec, changing the 6-4
  for a new file, creating an editor 1-3
  for an existing file, creating an editor 1-3
  Fred commands, description 2-6
  function description 6-1
  functions,
    dependence on grafports C-1
    Fred 9-6
  grafport as first field of 13-6
  hiding 6-5
  initializing 6-1
  inserting clipboard contents into the active 1-4
  layer number,
    obtaining the 6-5
    setting the 6-5
  list of types which can contain a close box 6-2
  locating a buffer for a 9-7
  as a Macintosh data type 12-1
  as Macintosh style editing display 2-1
  Macintosh standard record type, as Allegro pre-defined 13-3
  making visible 6-5
  menu,
    Allegro menubar, item descriptions 1-6
    use in detecting changed windows 2-2
  moving 6-4
  obtaining a list of existing 6-1
  position,
    as a component of a Fred window 9-1
    as component of pen state C-4
    obtaining the 6-4
    setting the 6-4
  as primary method for screen-related I/O 6-1
  printing the active 1-4
  record, on Macintosh heap, accessing pointer to 6-5
  replacing selections in the active 1-4
  reverting to last version saved 1-3
  regions as components of C-14
  saving the active 1-3
  searching in the active 1-4
  select event, handler for 8-3
  selecting 6-5
    the entire contents of the active 1-4
  size,
    changing the 6-4
    obtaining the 6-4
    setting the 6-4
  state, functions which manipulate C-3
  as streams, affect on drawing text C-7
  title,
    changing the 6-4
    obtaining the 6-4
    setting the 6-4
  type, as non-modifiable property of a window 6-2
  types, illustrations of 6-3
  testing for
    undo capability 6-6
    visibility 6-5
  window type as a non-modifialbe property of 6-2
  wptr as flag for closed 6-3
with-cursor, macro description 8-7
with-dereferenced-handles, macro description 12-9

use in obtaining a pointer within a handle 13-6
with-mark, macro description 9-3
with-pointer, macro description 12-9
with-port macro description 6-5
with-pstrs, macro description 12-8
without-interrupts, special form description 8-6
:word,
as argument for defpascal 12-10
as return value type for defpascal 12-10
as return-value-keyword for stack trap macros 12-3
word,
capitalizing the current 2-5, 9-5
deleting 2-5
downcasing the current 2-5
location, finding 9-6
moving the cursor
back one 2-4
forward one 2-4
long,
reading from memory 12-7
writing to memory 12-8
reading from memory 12-7
selecting 2-2
upcasing the current 2-5
writing to memory 12-8
working-directory, variable description 10-7
wptr,
as flag for closed windows 6-3
window variable description 6-5
writing
bytes to memory 12-8
the current buffer to a file 2-6
a file from a buffer 9-6
long-words to memory 12-8
to memory, facilities for 12-7
operating system types to memory 12-8
pointers to memory 12-8
strings to memory 12-8
words to memory 12-8
xor-region, function description C-15
y-or-n-dialog function description 4-3
yanking
a kill-ring string from a menu into the buffer 2-5
the current kill-ring string into the buffer 2-4
yes-no-cancel dialog, standard Macintosh, as Allegro turnkey dialog 4-3
zone pointer, how different from a generic pointer 12-9
zone-pointerp, function description 12-9