For
Macintosh Common Lisp
version 2

Macintosh Common Lisp Reference

030-1959-A
Developer Technical Publications
© Apple Computer, Inc. 1992

APPLE COMPUTER, INC.

© 1992, Apple Computer, Inc. All rights reserved.

No part of this publication or the software described in it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

Printed in the United States of America.

The Apple logo is a registered trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
408-996-1010

Apple, the Apple logo, APDA, AppleLink, A/UX, Chicago, LaserWriter, Macintosh, Monaco, MPW, MultiFinder, New York, and SANE are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Balloon Help, Finder, QuickDraw, and ToolServer are trademarks of Apple Computer, Inc.

Adobe Illustrator and PostScript are registered trademarks of Adobe Systems Incorporated.

AGFA is a trademark of Agfa-Gevaert

CompuServe is a registered trademark of CompuServe, Inc.

Helvetica, Palatino, and Times are registered trademarks of Linotype Company.

HyperCard is a registered trademark of Claris Corporation.

Internet is a trademark of Digital Equipment Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

MacPaint and MacWrite are registered trademarks of Claris Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Preface About This Book / xxi

About Macintosh Common Lisp / xxii
Macintosh tools / xxiii
Documentation / xxiii
Documentation conventions / xxv
- Courier font / xxv
- Italics / xxv
- Definition formats / xxvi
- Definition formats of CLOS generic functions / xxviii
- The generic function `initialize-instance` / xxviii
- Argument list punctuation / xxix
- Lisp syntax / xxx

Running Macintosh Common Lisp / xxxi
- Hardware requirements / xxxi
- System software / xxxi
- Installing Macintosh Common Lisp / xxxii
- Starting a Macintosh Common Lisp session / xxxii
- Ending a Macintosh Common Lisp session / xxxii

1 The Macintosh Common Lisp Environment / 1

Introducing the MCL environment / 3
The Lisp Listener, an interactive evaluation window / 3
- Variables associated with the Listener / 5

Evaluation / 6
Compilation / 7
The MCL editor, Fred / 7
- Features of the editor / 8
- Standard Macintosh commands and Fred commands / 8

Manipulating windows / 9

The programming procedure in Macintosh Common Lisp / 9
Editing files / 9
Evaluating files / 10
Compiling files / 10
Loading files / 11
Multitasking / 11
Automatic memory management (“garbage collection”) / 12
Ephemeral garbage collection / 12
Full garbage collection / 12
The MCL menus / 13
Apple menu / 14
File menu / 15
Edit menu / 16
Eval menu / 17
Tools menu / 18
Windows menu / 19
Customizing your work space / 20
Global variables in Print Options and Environment / 20
Print Options global variables / 21
Environment global variables / 21
The `init` file / 24

2 Editing in Macintosh Common Lisp / 27

The MCL editor / 29
The editing window / 30
Working with the editor / 31
Creating new windows and opening files / 31
Adding text to a file / 31
Saving buffers to files / 31
The minibuffer / 32
The kill ring and the Macintosh Clipboard / 32
Multiple fonts / 33
Packages / 34
Mode lines / 34
An `in-package` expression / 35
A `set-window-package` expression / 35
Finding a window’s package / 35
Fred parameters / 36
Normalizing `*next-screen-context-lines*` / 37
Editing in Macintosh style / 38

Editing in Emacs style / 39
The Control and Meta modifier keys / 39
Disabling dead keys / 40
Fred commands / 41
Help, documentation, and inspection functions / 42
Movement / 43
Selection / 46
Insertion / 49
Deletion / 52
Lisp operations / 54
Window and file operations / 56
Undo commands / 56
Numeric arguments / 57
Incremental searching in Fred / 57
Performing an incremental search / 58
Making additional searches / 58
Backing up with the Delete key / 58
Terminating an incremental search / 59
Doing another incremental search / 59
Special incremental search keystrokes / 60
Boyer-Moore search / 61

3 Points and Fonts / 63

Points / 64
How Macintosh Common Lisp encodes points / 64
How Macintosh Common Lisp version 2 stores points / 64
MCL functions relating to points / 65
Fonts / 69
Implementation of font specifications / 69
Implementation of font codes / 70
Functions related to font specifications / 71
Functions related to font codes / 75
System data / 82

4 Menus / 85

How menus are created / 87
A sample menu file / 87
The `menu-element` class / 88

The menu bar / 88
Menu bar forms / 88
Menu bar colors / 91
Menus / 93
MCL forms relating to menus / 93
MCL forms relating to elements in menus / 98
MCL forms relating to colors of menu elements / 101
Advanced menu features / 103
Menu items / 105
MCL forms relating to menu items / 106
MCL forms relating to menu item colors / 114
Window menu items / 115
Window menu item functions / 116
Window menu item class / 117
Updating the menu bar / 119
The Apple menu / 119
Example: A font menu / 120

5 Views and Windows / 121

The implementation of views and windows: The confusing part / 122
What simple views do / 122
What views do / 123
What windows do / 124
Class hierarchy of views / 124
Summary / 125
For more information / 126
MCL expressions relating to simple views and views / 126
Windows / 152
MCL functions for programming windows / 153
Advanced window features / 175
Supporting standard menu items / 181
Floating windows / 183

6 Dialog Items and Dialogs / 185

The dialog functionality in Macintosh Common Lisp / 187
- Dialog items / 187
- Dialog boxes / 187
- A simple way to design dialogs and program dialog items / 188
- Changes to dialogs in Macintosh Common Lisp version 2 / 188

Dialog items / 190

MCL forms relating to dialog items / 190

Advanced dialog item functions / 201

Specialized dialog items / 206
- Buttons / 206
- Default buttons / 207
- Static text / 209
- Editable text / 210
- Checkboxes / 216
- Radio buttons / 218
- Table dialog items / 221
- Pop-up menu dialog items / 234
- Scroll-bar dialog items / 236
- Sequence dialog items / 243

User-defined dialog items / 245

Dialogs / 246
- Modal dialogs / 247
- Modeless dialogs / 248

Simple turnkey dialog boxes / 248

MCL forms relating to dialogs / 256

7 Color / 261

Color encoding in Macintosh Common Lisp / 262

MCL expressions governing color / 263

Operations on color windows / 270
- Coloring user interface objects / 272
- Part keywords / 274
  - Menu bar / 274
  - Menus / 274
  - Menu items / 275
  - Windows / 275
  - Dialog items / 275
  - Table dialog items / 276

8 The Interface Toolkit / 277

The Interface Toolkit / 279
Loading the Interface Toolkit / 279
Editing menus with the Interface Toolkit / 280
Using the menu editing functionality / 280
Creating a new menu bar: Add New Menubar / 282
Getting back to the default menu bar: Rotate Menubars / 282
Deleting a menu bar: Delete Menubar / 283
Creating and editing menus: Add Menu / 283
Creating menu items / 283
Editing menu items / 284
Saving a menu bar / 285
Editing menu bar source code / 285
Editing dialogs with the Interface Toolkit / 286
Using the dialog-designing functionality / 287
Dialog-designing menu items / 287
Creating dialog boxes / 288
Adding dialog items / 290
Editing dialog items / 291

9 File System Interface / 295

Filenames, physical pathnames, logical pathnames, and namestrings / 297
Changes from earlier versions of Macintosh Common Lisp / 297
Printing and reading pathnames / 298
Pathname structure / 299
Macintosh physical pathnames / 300
Common Lisp logical pathnames / 300
Defining logical hosts / 301
Ambiguities in physical and logical pathnames / 301
More on namestrings and pathnames / 302
Creating and testing pathnames / 303
Parsing namestrings into pathnames / 305
The pathname escape character / 306
Loading files / 308
Macintosh default directories / 310
Structured directories / 313

Wildcards / 317
File and directory manipulation / 318
File operations / 322
Volume operations / 324
User interface / 327
Logical directory names / 330

10 Debugging and Error Handling / 333

Debugging tools in Macintosh Common Lisp / 334
Compiler options / 335
Fred debugging and informational commands / 336
Debugging functions / 340
Error handling / 348
Functions extending Common Lisp error handling / 348
Break loops and error handling / 349
Functions and variables for break loops and error handling / 353
Stack Backtrace / 355
Single-expression stepper / 358
Tracing / 359
MCL expressions associated with tracing / 363
Advising / 366
The Inspector / 369
Using the Inspector / 369
Inspector functions / 371
Other debugging macros / 372

11 Events / 375

Implementation of events in Macintosh Common Lisp / 376
How an event is handled / 376
MCL built-in event handlers / 377
Functions for redrawing windows / 387
Event information functions / 391
The event management system / 393
The cursor and the event system / 401
Event handlers for the Macintosh Clipboard / 405
MCL expressions relating to scrap handlers and scrap types / 406

12 Apple Events / 411

Implementation of Apple events / 412
Receiving Apple events / 412
- How to write an Apple event handler / 413
- Forms associated with Apple events / 414
- Installing Apple event handlers / 417
- Installing handlers for queued Apple event replies / 418
- Standard Apple event handlers / 420

Sending Apple events / 423

13 Streams / 425

Implementation of streams / 426
MCL expressions relating to streams / 426
Obsolete functions / 439

14 Programming the Editor / 441

How Macintosh Common Lisp handles text editing / 443
- Fred windows / 443
- Fred dialog items / 445
- Buffers and buffer marks / 445
- Copying and deletion mechanism: The kill ring / 446

MCL expressions relating to buffer marks / 447
- Using multiple fonts / 464
- Functions for manipulating fonts and font styles / 465

Fred functions / 470
The class `fred-mixin` / 470
Fred window and Fred dialog item functions / 471
- Creating a Fred window / 471
- Functions relating to Fred windows and Fred dialog items / 477

Functions implementing standard editing processes / 493
- Multiple-level Undo / 495
- Functions relating to Undo / 496

Working with the kill ring / 500
- Functions for working with the kill ring / 501
- Using the minibuffer / 502
- Functions for working with the minibuffer / 502

Defining Fred commands / 504

Fred command tables / 505
Keystroke codes and keystroke names / 505
Command tables / 507
Fred dispatch sequence / 507
MCL expressions associated with keystrokes / 507
MCL expressions relating to command tables / 512

15 Foreign Function Interface / 517

Using the Foreign Function Interface / 518
High-level Foreign Function Interface operations / 518
Argument specifications / 523
Result flags / 526
Short example / 527
Low-level functions / 528
Calling Macintosh Common Lisp from foreign functions / 532
Extended example / 533

16 Low-Level Operating System Interface / 535

Interfacing to Macintosh data structures / 537
Upgrading from previous versions of Macintosh Common Lisp / 538
Sharing data between Macintosh Common Lisp and the Macintosh Operating System / 539
Macptrs / 539
Memory management / 541
Stack blocks / 542
Accessing memory / 543
Miscellaneous routines / 554
Strings, pointers, and handles / 554
Pascal VAR arguments / 559
The Pascal null pointer / 559
Callbacks to Lisp from the ROM / 559

17 Higher-Level Operating System Interface / 563

Traps and records / 565
- Changes from previous versions of Macintosh Common Lisp / 565

Traps / 566

Loading trap definitions / 567

Defining a trap / 569
- Examples of calling traps / 570
- Determining record types and record field types / 571

Records / 574
- Installing record definitions / 575
- The structure of records / 575
- Defining record types / 576
- Variant fields / 579

Creating records / 580
- Creating temporary records with `rlet` / 580
- Creating records with indefinite extent / 582

Using records / 584

Getting information about records / 593

Trap calls using `stack-trap` and `register-trap` / 594
- Low-level stack trap calls / 595
- Low-level register trap calls / 597
- Lisp macros for calling Macintosh traps / 598

Notes on trap calls / 603
- 32-bit immediate quantities / 603
- Boolean values: Pascal true and false / 604

A Implementation Notes / 605

The Metaobject Protocol / 606
- Metaobject classes defined in Macintosh Common Lisp / 606
- Unsupported metaobject classes / 607
- Unsupported Introspective MOP functions / 608
- MCL functions relating to the Metaobject Protocol / 608

MCL class hierarchy / 621

Types and tag values / 621

Reader macros undefined in Common Lisp / 623

Numeric argument undefined in Common Lisp / 624

Numbers / 624

Characters / 626

Arrays / 626

Packages / 628
Additional printing variables / 629
Memory management / 630
Function swapping / 630
Garbage collection / 631
Ephemeral garbage collection / 632
Generations / 632
Memory management unit support / 632
Full garbage collection / 636
Evaluation / 637
Compilation / 638
Tail recursion elimination / 638
Self-referential calls / 638
Compiler policy objects / 638
Miscellaneous MCL expressions / 644

B Snapshots and Application Versions of Macintosh Common Lisp / 647

The snapshot utility / 648
Creating images / 648
Removing Macintosh pointers / 651
The top-level function / 653
Setting the top-level function / 653
Catching errors and aborts / 654
Using `*idle*` and `event-dispatch` / 655
Using `eval-enqueue` / 655

C The Common Lisp Object System / 657

Macintosh Common Lisp and the Common Lisp Object System standard / 659
How CLOS differs from Object Lisp / 659
Definitions / 660
Classes and their superclasses / 660
Slots / 661
Instances / 661
Generic functions and methods / 661

Classes and instances / 662
Creating a class with the macro `defclass` / 662
Creating an instance and giving its slots values / 664
Redefining a class / 665
Allocating the value of a slot in a class / 666
Classes as prototypes of other classes / 667
Methods / 667
Defining a method and creating a generic function / 668
Congruent lambda lists / 669
Documentation of methods / 669
Defining methods on instances / 670
Creating and using accessor methods / 670
Customizing initialization with `initialize-instance` / 672
Creating subclasses and specializing their methods / 673
Method combination / 674
The primary method / 674
The primary method and the class precedence list / 674
Examples of classes with multiple superclasses / 675
When there is a conflict: Choosing between methods / 676
Choosing between methods associated with direct and with more distant superclasses / 677
Creating auxiliary methods and using method qualifiers / 678
Mixin classes and auxiliary methods / 680
Extended example / 680

D Converting Your Files to CLOS / 685

General procedure for converting files to CLOS / 686
Turning `defobject` and `exist` pairs into `defclass` and `initialize-instance` pairs / 686
Turning occurrences of `defobfun` into `defmethod` / 687
Replacing free variable references / 688
Removing calls to `ask` / 689
Replacing `oneof` with `make-instance` and `kindof` with `defclass` / 689
Changing function names / 690

An extended example / 691
Calling packages and setting variables / 692
Turning calls to `defobject` and `exist` into `defclass` and `initialize-instance` / 694
Turning other calls to `defobfun` into `defmethod` / 698
Removing `ask` and replacing `oneof` by `make-instance` / 700
Making a `window` instance / 704

E QuickDraw Graphics / 705

QuickDraw in Macintosh Common Lisp / 707
Windows, GrafPorts, and PortRects / 707
Points and rectangles / 708
Window state functions / 710
Pen and line-drawing routines / 712
Drawing text / 720
Calculations with rectangles / 721
Graphics operations on rectangles / 726
Graphics operations on ovals / 730
Graphics operations on rounded rectangles / 733
Graphics operations on arcs / 737
Regions / 741
Calculations with regions / 744
Graphics operations on regions / 748
Bitmaps / 749
Pictures / 751
Polygons / 753
Miscellaneous procedures / 757

F For More Information / 763

The single most valuable source / 764
If you are learning Lisp / 764
If you are learning CLOS / 765
If you're learning about Macintosh programming / 766
If you're updating from a previous version of Macintosh Common Lisp / 766
If you'd like examples of MCL programming / 767

If you'd like to communicate with other MCL programmers / 767
Communicating through Internet / 767
Communicating through Usenet / 768
Communicating through AppleLink / 768
Communicating through CompuServe / 768
Communicating through other services / 768
Bugs and bug fixes / 768
Infrequent announcements of general interest / 769
If you're a developer: APDA / 769

Index / 771

Figures and tables

1 The Macintosh Common Lisp Environment / 1

Figure 1-1 The Lisp Listener / 3
Figure 1-2 The MCL default menu / 13
Figure 1-3 The String Search dialog box / 17

Table 1-1 File menu items / 15
Table 1-2 Edit menu items / 16
Table 1-3 Eval menu items / 17
Table 1-4 Tools menu items / 18
Table 1-5 Environment options / 21

2 Editing in Macintosh Common Lisp / 27

Figure 2-1 An editor window / 30

Table 2-1 Fred parameters / 36
Table 2-2 Fred commands for help, documentation, and inspection / 42
Table 2-3 Fred commands for movement / 44
Table 2-4 Fred commands for selection / 46
Table 2-5 Fred commands for insertion / 49
Table 2-6 Fred commands for deletion / 52
Table 2-7 Fred commands for Lisp operations / 54
Table 2-8 Fred commands for window and file operations / 56
Table 2-9 Fred commands for undoing commands / 56
Table 2-10 Fred commands for giving numeric arguments / 57
Table 2-11 Fred commands for searching / 60

4 Menus / 85

Table 4-1 Window menu items / 116

5 Views and Windows / 121

Figure 5-1 The class hierarchy of views from `simple-view` downward / 125

6 Dialog Items and Dialogs / 185

Figure 6-1 Examples of tables used in dialog boxes / 222
Figure 6-2 Cell positions represented as points / 223
Figure 6-3 A modal dialog (Print Options on the Tools menu) / 246
Figure 6-4 A modeless dialog (List Definitions on the Tools menu) / 247
Figure 6-5 A message dialog box / 250
Figure 6-6 A yes-or-no dialog box / 251
Figure 6-7 A `get-string-from-user` dialog box / 253
Figure 6-8 A `select-item-from-list` dialog box / 255

Table 6-1 Summary of changed dialog functions in Macintosh Common Lisp version 2 / 189

8 The Interface Toolkit / 277

Figure 8-1 The Interface Toolkit menu on the menu bar / 279
Figure 8-2 Choosing Edit Menubar from the Design menu / 280
Figure 8-3 The Menubar Editor window / 281
Figure 8-4 A Menu Editor window showing a menu with no items / 283
Figure 8-5 Editing items in the Menu Editor / 284
Figure 8-6 New Dialog dialog box / 289
Figure 8-7 Dragging an editable-text dialog item into an untitled dialog box / 291
Figure 8-8 Edit Dialog Items dialog box / 292

Table 8-1 Menubar Editor window options / 282
Table 8-2 Menu editing options / 284
Table 8-3 Menu item editing options / 285
Table 8-4 Menu items and corresponding MCL codes / 286
Table 8-5 Dialog design menu items / 287
Table 8-6 Seven types of dialog / 289
Table 8-7 Two attributes of dialog boxes / 290
Table 8-8 Editable options in dialog items / 292
Table 8-9 Editable options in subclasses of dialog items / 293

9 File System Interface / 295

Table 9-1 Some namestrings parsed into pathnames / 306
Table 9-2 Effect of escape characters / 307

10 Debugging and Error Handling / 333

Figure 10-1 MCL debugging tools / 334
Figure 10-2 Effects on the stack of `break`, `abort`, and `continue` / 350
Figure 10-3 Nesting of break loops / 352
Figure 10-4 Two ways to leave a break loop / 353
Figure 10-5 A Stack Backtrace dialong box / 356

Table 10-1 Compiler options / 336
Table 10-2 Fred debugging and informational commands / 338
Table 10-3 Constructs and their documentation types / 344
Table 10-4 Options in Inspector Central / 370

14 Programming the Editor / 439

Figure 14-1 The classes `fred-window`, `fred-mixin`, `window`, and `listener` / 444

Table 14-1 Modifier bits in the keystroke code / 506

15 Foreign Function Interface / 517

Table 15-1 Foreign type defaults / 525

17 Higher-Level Operating System Interface / 563

Table 17-1 Pascal types and their equivalent MCL types / 572
Table 17-2 Predefined record field types and their lengths / 578

A Implementation Notes / 605

Table A-1 Structure of metaobject classes defined in Macintosh Common Lisp version 2 / 606
Table A-2 Types of array element / 626
Table A-3 Theoretical limits on array length / 627
Table A-4 Additional printing variables / 628

C The Common Lisp Object System / 657

Figure C-1 The class `fourth-grader` / 663
Figure C-2 An instance of `fourth-grader` with a value in the `name` slot / 665

E QuickDraw Graphics / 705

Figure E-1 Location of point at upper-left corner of pixel / 708
Figure E-2 A PortRect / 709
Figure E-3 Multiple methods of passing rectangles / 711
Figure E-4 Attributes of a graphics pen / 712
Figure E-5 QuickDraw pen sizes / 714
Figure E-6 Pen pattern stored as a 64-bit block of memory / 716
Figure E-7 Effect of pen modes on pixels being drawn / 717
Figure E-8 Offset rectangle / 721
Figure E-9 Inset rectangle / 722
Figure E-10 Rectangle resulting from the intersection of two others / 723
Figure E-11 Smallest rectangle completely enclosing two others / 724
Figure E-12 Point to angle, calculated from two rectangles / 725
Figure E-13 Rectangle framed in the current pen / 727
Figure E-14 Effects of `paint-rect` and `invert-rect` / 729
Figure E-15 An oval within a rectangle / 730
Figure E-16 Rounded rectangle / 733
Figure E-17 Framing an arc / 738
Figure E-18 Regions / 741
Figure E-19 A rectangle scrolled down and to the right / 751
Figure E-20 A framed polygon / 755
Figure E-21 Effect of `map-point` / 759
Figure E-22 Effect of `map-rect` / 760

Preface

About This Book

Contents

About Macintosh Common Lisp / xxii
Macintosh tools / xxiii
Documentation / xxiii
Documentation conventions / xxv
- Courier font / xxv
- Italics / xxv
- Definition formats / xxvi
- Definition formats of CLOS generic functions / xxviii
- The generic function `initialize-instance` / xxviii
- Argument list punctuation / xxix
- Lisp syntax / xxx

Running Macintosh Common Lisp / xxxi
- Hardware requirements / xxxi
- System software / xxxi
- Installing Macintosh Common Lisp / xxxii
- Starting a Macintosh Common Lisp session / xxxii
- Ending a Macintosh Common Lisp session / xxxii

The Preface describes Macintosh Common Lisp's features for developing software tools and applications on the Macintosh. It describes the online and printed documentation, discusses MCL syntax and conventions, and describes how to install, run, and exit from Macintosh Common Lisp.

About Macintosh Common Lisp

Macintosh Common Lisp (MCL) provides a fluid and flexible environment for developing software tools and applications. Originally designed to handle the intricacies of artificial intelligence, Lisp is ideal for the complexities of programming the Macintosh computer.

Thirty years of evolution have made Lisp both efficient and rich in programmer-oriented features. Lisp compiles down to code that can be as fast as C code, and its object-oriented features are better integrated and allow dynamic creation and destruction of objects more naturally than C or Pascal.

Macintosh Common Lisp provides an interactive programming environment that saves time and improves debugging. You can compile, test, and debug functions individually, without completely recompiling and relinking an entire program. In addition, Lisp handles many details automatically. For example, Macintosh Common Lisp automatically reclaims unused memory, and if you are running with virtual memory, this task is usually done in background mode.

Macintosh Common Lisp also provides a full range of tools and programming aids. These include an editable Inspector, a source code stepper, a trace facility, a fast Lisp-oriented editor, and tools that give you quick access to the definitions of functions and variables.

Code written in other languages can be accessed through MCL's Foreign Function Interface.

Macintosh Common Lisp implements Common Lisp and the Common Lisp Object System as described in the second edition of *Common Lisp: The Language*, by Guy Steele, Jr., and others (Digital Press, 1990). It provides partial implementation of the Metaobject Protocol (MOP) as described in *The Art of the Metaobject Protocol*, by Gregor Kiczales and others (MIT Press, 1991). Future updates of Macintosh Common Lisp will provide expanded MOP support.

For details on Macintosh Common Lisp's implementation of Common Lisp and the MOP and the MCL approach to situations not clearly defined by Common Lisp, see Appendix A, "Implementation Notes."

In addition, Macintosh Common Lisp provides additional tools for programming and for interacting with the Macintosh interface.

Macintosh tools

Macintosh Common Lisp provides major portions of the Macintosh Toolbox as high level Lisp objects. This design makes it easy to write programs that use the resources of the Macintosh interface. With Macintosh Common Lisp, it takes only minutes to create customized menus and dialog boxes. Specialized types of windows that build on these standard ones can be created as new system objects. These tools make programming the Macintosh computer easy.

Macintosh Common Lisp provides complete low-level access to all Macintosh traps and data structures. In Macintosh Common Lisp version 2, these interfaces have been completely rewritten for ease of use and robustness.

Macintosh Common Lisp handles the four standard Apple events and provides facilities for creating and sending Apple events.

Documentation

The MCL documentation includes

- this manual, which provides complete documentation for the MCL programming environment and extensions
- *Getting Started With Macintosh Common Lisp* , which introduces MCL features to programmers unfamiliar with the language
- online documentation of all MCL and Common Lisp symbols, available through the Documentation menu item on the Tools menu (you can add documentation for your own symbols)
- help balloons for all menu items, autoloaded when you turn on Balloon Help
- extensive help and debugging features, supplied source code, and commented examples in both the Examples and Library folders

For documentation of the Common Lisp language and the Common Lisp Object System, you should also have access to a Common Lisp reference such as the second edition of *Common Lisp: The Language* .

This manual is intended for programmers who are familiar with both the Macintosh interface and Lisp. It doesn't assume experience programming the Macintosh computer, just experience using it.

If you're new to the Macintosh world, read the Macintosh owner's guide and try some simple Macintosh applications, such as MacWrite and MacPaint.

If you're new to Macintosh Common Lisp, read the manual *Getting Started With Macintosh Common Lisp*. It takes you through a short programming session, gives you tips and shortcuts, and shows you Macintosh Common Lisp's online documentation, help, and debugging features, as well as introducing you to some of the supplied source code.

If you need to learn Lisp, good Common Lisp tutorials are available in Timothy Koschmann's *A Common Lisp Companion*, in Winston and Horn's *Lisp*, third edition, in David Touretzky's *Common Lisp*, and in other introductory books on the language. For a list of these and other useful books, see Appendix F of this manual, "For More Information."

If you are new to the Common Lisp Object System, and especially if you are upgrading from version 1.3 or earlier of Macintosh Common Lisp, read Appendix C of this book, "The Common Lisp Object System," and Appendix D, "Converting Your Files to CLOS." The fullest description of CLOS appears in *Common Lisp: The Language*, Chapter 28; you should be familiar with that information. An extended CLOS tutorial is available in Sonya E. Keene's *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS* (Addison-Wesley, 1989), and Koschmann's *Common Lisp Companion* also covers CLOS in some detail. See Appendix F of this manual, "For More Information."

As you program, you should consult this reference, the online documentation, the supplied source code, and the examples provided in the MCL directory Examples. (The Library directory also contains useful code.) Macintosh Common Lisp provides continuing technical help for programmers via an online bulletin board. On this board, `info-mcl`, available from several sources, programmers can consult with one another and with MCL developers about solving applications problems. Many files of code are available on `info-mcl`, and you can contribute your own code to it. See Appendix F, "For More Information."

Because Macintosh Common Lisp adds a high-level interface to many Macintosh routines, you may not need to use *Inside Macintosh* as a reference. However, *Inside Macintosh* is indispensable for programming most Macintosh features. A full cross-reference to *Inside Macintosh* is available in the *Inside Macintosh X-Ref* (Addison-Wesley, 1991).

Documentation conventions

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

Courier font

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

Here is a sample interaction.

```
? (+ 10 10)                          <—This is what you type.
20                                   <—This is the program's response.
? (format nil "A string.")                    <—This is what you type.
"A string."                          <—This is the program's response.
```

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

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

Italics

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

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

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

Definition formats

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

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

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

pop-up-menu [*Class name*]

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

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

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

with-focused-view [*Macro*]

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

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

Arguments

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

Example

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

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

find-window [*Function*]

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

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

Arguments

title	A string specifying the title of the window to search for.
class	A class used to filter the result. (The `&optional` in the syntax means that this argument is optional; that is, if it is omitted, a specified default value is used.)

Definition formats of CLOS generic functions

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

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

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

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

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

Arguments

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

Example

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

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (set-view-position ed #@(20 20))
1310740
```

The generic function `initialize-instance`

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

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

initialize-instance [*Generic function*]

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

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

Arguments

dialog-item A dialog item.

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

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

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

Argument list punctuation

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

- Brackets [ ] indicate that anything they enclose is optional. This means that anything within them may appear once or not at all.
- Braces { } followed by an asterisk * mean that whatever they enclose may appear any number of times or not at all; everything within the braces is interpreted as a group.
- Braces { } followed by a plus sign + mean that whatever they enclose may appear multiple times but must appear at least once.
- A vertical bar | inside braces or brackets separates mutually exclusive choices. The group may be composed of a set from one side of the bar or from the other.
- Double brackets [[ ]] indicate that any number of the enclosed alternatives may appear, and in any order, but that each alternative may be used at most once unless followed by an asterisk.
- A downward arrow ↓ indicates a form of syntactic indirection that helps to make double-bracket notation more readable.

Lisp syntax

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

The following are some general characteristics of Lisp syntax:

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

  Nested lists are enclosed in nested parentheses:

  ```
  (like (these))
  ```

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

  The expression `'foo` means `(quote foo)` and the expression `'(cons 'a 'b)` means `(quote (cons (quote a) (quote b)))` .

- A semicolon signals a comment. The semicolon and all characters following it up to the end of the line are ignored. A newline signals the end of the comment:

  ```
  (Here is Lisp code) ;Here is a comment,
                      ;which continues here.
  (and here is Lisp code again)
  ```

- Quotation marks, also called double quotes, surround character strings:

  ```
  "like this"
  ```

- A backslash `\`, the escape character, causes the next character to be treated as a letter rather than syntactically. For example, `\{` indicates the character for a left brace.
- Vertical bars in pairs `| |` surround the name of a symbol with many special characters. Surrounding some characters with vertical bars is roughly equivalent to putting a backslash before each of the characters. (Remember that a single vertical bar separates mutually exclusive choices in the description of a special form or macro.)
- A number sign `#`, also called a hash mark, signals the beginning of a complicated syntactic structure. The next character designates the syntactic structure to follow. For example, `#b1001` means `1001` in binary notation; `#(foo bar baz)` denotes a vector of three elements, `foo`, `bar`, and `baz`; `#\A` denotes the character object `A`; `#P"foo:bah"` indicates the pathname `"foo:bah"` ; and `#'`*function* means `(function` *function*`)`.

- A grave accent ` (also called a backquote) is used together with commas. The backquote syntax represents a program that will construct a data structure; commas are used within backquote syntax.
- A colon is used to indicate the package of a symbol. For instance, `lisp:dialog-item-size` denotes the symbol `dialog-item-size` in the package named `lisp`.

Running Macintosh Common Lisp

This section explains the hardware and operating system requirements of Macintosh Common Lisp, the installation process, and the methods for running Macintosh Common Lisp.

Hardware requirements

Macintosh Common Lisp version 2 requires approximately 4.5 MB of disk space. (The release notes distributed with Macintosh Common Lisp give exact figures for both the basic system and options such as the Toolbox interfaces.)

Macintosh Common Lisp should be given a partition of at least 2 MB; 3 MB is recommended. Increasing partition size improves performance.

You may wish to turn off the RAM cache and remove memory-hungry system extension files before starting Macintosh Common Lisp.

Under the limit imposed by Macintosh System 6, Macintosh Common Lisp can access no more than 8 MB of RAM. This limitation does not exist in System 7.

System software

Macintosh Common Lisp version 2 has been designed to run with either version 6.0.4 and above or version 7.0 of the Macintosh system software. If you have versions older than 6.0.4, you should upgrade.

Installing Macintosh Common Lisp

To install Macintosh Common Lisp, see the release notes distributed with your Macintosh Common Lisp package.

Starting a Macintosh Common Lisp session

To run Macintosh Common Lisp, double-click the Macintosh Common Lisp icon. This action loads the kernel, initializes the Lisp system, and loads the file `init.lisp` or `init.fasl` (if one of these is present in the same directory as Macintosh Common Lisp). After this, you will see the MCL main menu, a Listener window, and a welcome message.

Ending a Macintosh Common Lisp session

To end an MCL session, choose Quit from the File menu or use the Quit function that follows. If you have any unsaved files, Macintosh Common Lisp will ask you if you want to save them before it quits.

`quit` [*Function*]

Syntax `quit`

Description The `quit` function exits from Macintosh Common Lisp.

Chapter 1 The Macintosh Common Lisp Environment

Contents

Introducing the MCL environment / 3
The Lisp Listener, an interactive evaluation window / 3
    Variables associated with the Listener / 5
Evaluation / 6
Compilation / 7
The MCL editor, Fred / 7
    Features of the editor / 8
    Standard Macintosh commands and Fred commands / 8
Manipulating windows / 9
The programming procedure in Macintosh Common Lisp / 9
    Editing files / 9
    Evaluating files / 10
    Compiling files / 10
    Loading files / 11
Multitasking / 11
Automatic memory management (“garbage collection”) / 12
    Ephemeral garbage collection / 12
    Full garbage collection / 12
The MCL menus / 13
    Apple menu / 14
    File menu / 15
    Edit menu / 16
    Eval menu / 17
    Tools menu / 18
    Windows menu / 19
Customizing your work space / 20
    Global variables in Print Options and Environment / 20
        Print Options global variables / 21
        Environment global variables / 21
    The `init` file / 24

This chapter describes the basic components of the Macintosh Common Lisp programming environment. It introduces the basics of the Lisp Listener, the editor, evaluation, compilation (which is handled incrementally), window manipulation, and the programming procedure in Macintosh Common Lisp. It briefly discusses multitasking and MCL memory management. It describes the MCL tools available from the menu bar. It also provides some suggestions about customizing your work space.

Introducing the MCL environment

Macintosh Common Lisp provides an integrated programming environment built around three components: the Lisp Listener, an Emacs-style editor, and a set of window- and menu-based programming tools.

In addition, Macintosh Common Lisp is fully compatible with the Macintosh interface; you can work as you would anywhere else on the Macintosh.

The sections that follow describe the Lisp environment of Macintosh Common Lisp.

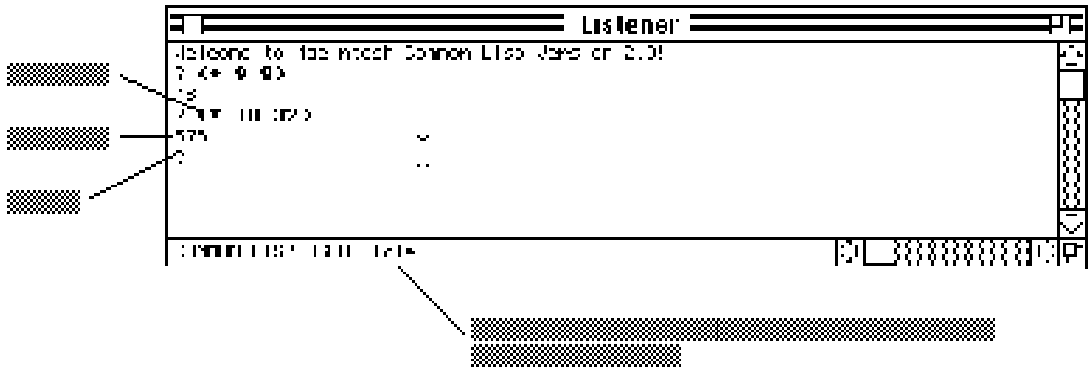
The Lisp Listener, an interactive evaluation window

One of Lisp's most important advantages is support for interactive programming. This means that you can type Lisp forms, and Lisp will evaluate them—do what they say—and print a result immediately.

The **Lisp Listener** (or just the Listener for short) is a special window designed for interactive Lisp programming (see Figure 1-1). Interaction between the programmer and Lisp takes place in this special window. You send forms to the Listener, or type them into the Listener directly, and Macintosh Common Lisp evaluates them and prints the result in the Listener.

- Text you enter into the Listener appears in boldface. Text that Macintosh Common Lisp generates does not.

- **Figure 1-1** The Lisp Listener



In general, interaction occurs at the bottom of the text displayed in the Listener. Scroll upward in the Listener to see the history of the interactive session.

In ordinary use, there is only one Listener window at a time. Like any other window, the Listener may be closed. If there is no Listener, any attempt to write to `*terminal-io*` creates a new Listener. When the Listener is closed, all its text is lost, but any symbol defined in the Listener remains defined for the duration of the session.

Here are some of the Listener's features:

- Pressing Return when the insertion point is not in the last line of text in the Listener causes the insertion point to move to the last line. If there is a selection, the selection is copied to the last line. If there is no selection, the text surrounding the insertion point moves down if it has been entered by the user, and is ready to evaluate.
- Pressing Enter on the numeric keypad is equivalent to pressing Return twice. It performs both copy-down and evaluation.
- Many keystroke commands are available using two modifier keys, Control and Meta. By default, Control modifiers are typed by pressing the Control key and Meta modifiers by pressing the Option key.
- Pressing Control-Return causes a carriage return to be inserted. Use Control-Return when you want to reformat text in the Listener without performing copy-down or evaluation.
- Pressing Control-G creates a new input line without evaluating the current input line. The canceled input line is not erased, and you can use it later.

```
? Here is an input line       ;now press Control-G
?                             ;you have a new input line
```

- Pressing Meta-G moves the previous input line to the bottom of the text in the Listener. Each time you press this key combination, a previous input line is moved to the bottom of the text in the Listener.
- Pressing Control-Meta-P in the Listener moves the cursor to the previous input line. (Press the Control, Meta, and P keys all at once to give this command.)
- Pressing Control-Meta-N in the Listener moves the cursor to the next input line.
- To get an online list of Listener editing commands, press Control–question mark when you are in the Listener, or choose Listener Commands from the Tools menu.
- Most editing commands and features work in the Listener as well. (They are discussed in the section "The MCL Editor, Fred" later in this chapter.)

Here are some other features of the Listener:

- The Listener's text may be saved. Select Save As from the File menu to save a copy of the text in the current Listener.

- You may copy and paste text between the Listener and other MCL windows. (But multiple-font selections pasted into the Listener do not take their font information with them, since the Listener enforces its own plain/bold protocol.)

Variables associated with the Listener

The following four variables govern the behavior of the Lisp Listener.

top-listener [*Variable*]

Description The `*top-listener*` variable contains the current Listener.

listener-default-font-spec [*Variable*]

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

listener-window-position [*Variable*]

Description The `*listener-window-position*` variable specifies a point indicating the position used when new Listener windows are created. The user may set this variable.

Example

Here is an example of setting this variable. (The `point-string` and `make-point` functions are documented in the "Points" section of Chapter 3, "Points and Fonts.")

```
? *listener-window-position*
19660850
? (point-string *listener-window-position*)
"#@(50 300)"
? (setf *listener-window-position* (make-point 20 300))
19660820
```

listener-window-size [*Variable*]

Description The *listener-window-size* variable specifies a point indicating the size used when new Listener windows are created. The user may set this variable.

Evaluation

Lisp expressions are easily evaluated from within the Listener or from elsewhere in Macintosh Common Lisp.

If the last line of text in the Listener is a complete expression, press Return to evaluate the expression. If the expression is not complete (for example, if there are unclosed parentheses or quotation marks), then a carriage return is inserted, just as in a normal editor window.

Lisp expressions can be evaluated from any editor window; it is not necessary to copy them to the Listener.

To evaluate an expression from an editor window:

1. **Select the expression or place the insertion point at the beginning or end of the expression.**

   If there is a selection, it can include more than one expression.

2. **Do any one of the following:**

   Press the Enter key on the numeric keypad.

   Press Command-E.

   Choose Eval Selection from the Eval menu.

Results appear in the Listener.

When Macintosh Common Lisp is performing evaluation, the About Macintosh Common Lisp menu item is preceded by a diamond and the Listener minibuffer shows the word “busy”.

Compilation

Macintosh Common Lisp uses an incremental compiler. When you evaluate an expression, Macintosh Common Lisp compiles it and runs the compiled code. Because the compiler is fast, there is no noticeable delay.

The MCL editor, Fred

Macintosh Common Lisp includes a powerful text editor called Fred (Fred Resembles Emacs Deliberately), which is an implementation of Emacs, the powerful programmable editor that is standard in most implementations of Lisp.

Fred includes the usual Macintosh editor features, such as multiple windows, mouse based cut and paste operations, and so on. If you are an experienced Macintosh user, you can get started with Fred simply by using the editing commands you already know.

Fred also includes a set of commands designed specifically for Lisp programming. These commands are described in detail in Chapter 2, “Editing in Macintosh Common Lisp,” and summarized below. They include commands for manipulating Lisp, dealing with windows, inspecting functions, and getting help, as well as many specialized editing commands. As you work with Macintosh Common Lisp, you should familiarize yourself with them.

Fred commands use two modifier keys, called Control and Meta.

- To type Control modifiers, use the Control key if your Macintosh keyboard has one. If your Macintosh does not have a Control key, use Command or Command Shift, depending on the value of the variable `*emacs-mode*` . See Chapter 2, “Editing in Macintosh Common Lisp.”
- To type Meta modifiers on the Macintosh keyboard, use the Option key.

Since Fred is written in Macintosh Common Lisp, it is completely programmable. Two changes users often make are rebinding the Meta key to Escape and adding a fonts menu. The file `escape-key.lisp` in the Examples folder binds the Macintosh Escape key to Meta, and the file `font-menus.lisp` in the same folder adds a set of font menus.

Features of the editor

Here are some of the features of the editor. They all work in the Listener window, as do almost all Fred commands.

- Placing the insertion point after a close parenthesis, or before an open parenthesis, causes the matching parenthesis to blink. For example, placing the insertion point after a close parenthesis causes the matching open parenthesis to blink. This feature is very helpful in balancing parentheses.
- Double-clicking after a close parenthesis, or before an open parenthesis, highlights to the matching parenthesis. For example, double-clicking before an open parenthesis highlights forward to the matching close parenthesis. This is another quick way to check the balance of your parentheses.
- Pressing Tab after a Return indents the new line appropriately.
- Pressing Control-Meta-Q reindents the current expression in a readable way.
- Pressing Meta–close parenthesis moves the cursor into position for typing the next expression.
- Pressing Enter on the numeric keypad evaluates the selection or the form within which the insertion point is located.

These features are available to all user-designed windows. You can use them, suppress them, or use Macintosh Common Lisp to customize any special behavior your application requires.

Fred, like the Listener, has online documentation. Choose Fred Commands from the Tools menu to see a window of all Fred commands.

Standard Macintosh commands and Fred commands

All standard Macintosh editing commands work in the Listener and in Fred windows. In most circumstances, Macintosh Common Lisp does not care whether you use Macintosh editing commands, Fred commands, or a combination. The significant exception is that Fred commands can work more powerfully than their Macintosh equivalents; for example, the Fred Copy command is more powerful than the Macintosh Clipboard, and so certain items copied in Fred cannot be retrieved from the Clipboard. That exception is discussed in more detail in Chapter 2, “Editing in Macintosh Common Lisp.”

Manipulating windows

When you edit Lisp code, Macintosh Common Lisp lets you work with multiple windows on screen. For the most part these are worked with, resized, and closed just like other Macintosh windows. Three features make it especially easy to move among Macintosh Common Lisp windows:

- Pressing Command-L makes the Listener the frontmost window.
- Pressing the Option key and clicking the title bar of any window except a modal dialog makes that window the rearmost window.
- Holding down the Option key when closing a window closes all the windows of that class.
- Holding down the Command key when closing a window hides the window. Its title appears in italics in the Windows menu.

The programming procedure in Macintosh Common Lisp

This section describes how to use the editor to write programs and how to compile and load files of Lisp source code.

Editing files

When you begin learning Lisp, you may simply want to type expressions into the Listener to see what happens. Soon, however, you want to start saving your work. You do this by editing definitions and other Lisp forms in an editor window, rather than in the Listener.

To open a file, choose the standard Macintosh Open command from the File menu.

The editor opens and saves files in text-only format, so that they can be shared with other Macintosh editors, such as QUED and MPW. If you use a word processor as your editor—for example, Microsoft Word or MacWrite—be sure to save the files in text only format *with line breaks* . This allows Macintosh Common Lisp to read them.

- *Note:* If you use another editor on a Fred file containing multiple fonts, the Fred font information is corrupted.

You can evaluate forms and compile definitions directly from editor windows; there is no need to copy them into the Listener.

- In the Listener, forms are automatically evaluated when they are complete and you press Return.
- In an editor window, place the cursor directly before or after a form, or select a form; then press Command-E, press Enter on the numeric keyboard, or choose Eval Selection from the Eval menu. The results of the evaluation are displayed in the Listener.

To save files, choose Save or Save As from the File menu and give your file a name.

The usual extension for source files is `.lisp`. The extension `.text` is used for text files that contain non-Lisp information, such as documentation, notes, and the like.

Evaluating files

To evaluate the contents of an entire file, open the file and choose Eval Buffer from the Eval menu. The effect is the same as typing every expression in the file into the Listener; all the definitions in the file are available to Macintosh Common Lisp for the duration of the session.

Compiling files

When you save Lisp code in editable files, it is saved as simple text. That is, the text of the definitions and other expressions is saved. The compiled forms of the expressions are not saved. This means that when you restart Lisp and open the file, you need to recompile the functions in the file before you can use them. Although the Macintosh Common Lisp compiler is fast, this can take some time, especially when you are working on a large project.

To avoid the need to recompile functions every time you restart Lisp, Macintosh Common Lisp provides a *file compiler*. The file compiler takes a source code file (that is, a text file of Lisp forms, including definitions), compiles all the forms in the file, and saves the compiled versions in another disk file. This file of compiled forms, called a **`fasl`** file, can then be quickly loaded into Lisp. (Loading is described in the next section.)

One way to compile files is by choosing Compile File from the Tools menu. When you choose this command, Macintosh Common Lisp displays a directory dialog box, allowing you to pick a file to compile. Only uncompiled files are shown. When you have selected a file, you are prompted for a name under which to save the file. The default is the name of the original text file, but with a `.fasl` extension.

You can also compile files with the Common Lisp function `compile-file`.

Loading files

Loading a file is like opening the file and evaluating all the forms in it. One difference is that you can load both source files and compiled files. Macintosh Common Lisp loads `fasl` files quickly because the loaded information is already compiled. Loading source files can take somewhat longer because the compiler is invoked.

You can load files by choosing Load from the Eval menu or by using the Common Lisp function `load`.

Multitasking

Macintosh Common Lisp has a pseudo-multitasking system that allows editing and other operations to occur while Lisp code is running (for instance, during evaluation and compilation). It also means that events are still processed when your main program is running.

You can stop Lisp operations—except those that cannot be interrupted—by pressing Command-period or by choosing Abort from the Eval menu. This returns control to the main Listener loop. You can suspend an operation by choosing Break from the Eval menu or by pressing Command-comma.

Certain MCL tasks (for instance, garbage collection, described in the next section) cannot be interrupted. During these operations no other tasks can be performed.

Choose Continue or press Command-slash to resume the operation. Continue is also available on the list generated by the Restarts menu item on the Eval menu.

Automatic memory management (“garbage collection”)

In some programming languages, memory management can be a problem. One of the advantages of Macintosh Common Lisp (and Lisp in general) is that you do not need to explicitly deallocate storage for variables or other data structures; Macintosh Common Lisp handles this automatically for you.

To implement memory management, Macintosh Common Lisp provides a **garbage collector,** a small routine that periodically recycles the memory from unneeded data structures.

Macintosh Common Lisp has two kinds of automatic memory management, called ephemeral garbage collection and full garbage collection.

Ephemeral garbage collection

In general, most heap-allocated objects become inaccessible soon after they are created. The ephemeral garbage collector exploits this by concentrating its efforts on reclaiming memory allocated to newly created objects.

The ephemeral garbage collector partitions the population of all dynamically allocated objects into three sets, called generations. Generations are divided roughly into ages by time of creation. Objects first go into the space allocated to the youngest generation. When that space fills up, Macintosh Common Lisp performs an ephemeral garbage collection on only that space, clearing it of all objects. Objects that cannot be reclaimed are promoted to the middle generation. When the middle generation fills up, Macintosh Common Lisp reclaims space within its partition, promoting surviving objects to the oldest generation.

Only when all three sets are full is a full garbage collection invoked.

Ephemeral garbage collection can be enabled by calling the function `egc`.

Full garbage collection

Full garbage collection reclaims memory throughout the system. During full garbage collection, Lisp operation pauses and the cursor turns into the letters `GC`. Garbage collection typically takes between 1 and 10 seconds, depending on the microprocessor and amount of RAM. During a full garbage collection no other tasks can be performed.

You can invoke garbage collection manually by calling the function gc.

For full details on both kinds of garbage collection, see “Garbage Collection” in Appendix A, “Implementation Notes.”

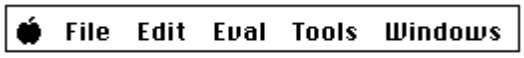
The MCL menus

This section summarizes the menus and menu items available from the MCL menu bar. Many of the individual menu commands are described in more detail in other sections of this manual. For example, the command New, which creates a new editor window, is described in Chapter 2, “Editing in Macintosh Common Lisp.”

The MCL menu bar is created completely by Lisp code, using the Common Lisp Object System. You can customize the menu bar and create alternative menu bars; this process is described in Chapter 4, “Menus,” and in Chapter 3 of *Getting Started With Macintosh Common Lisp* .

This section describes the menu bar that appears when you start up an unmodified version of Macintosh Common Lisp (see Figure 1-2). Keyboard equivalents for commands are noted after the command names.

Figure 1-2 The MCL default menu



The default menu bar contains six menu titles:

- Apple
- File
- Edit
- Eval
- Tools
- Windows

The first three menus are common to most Macintosh applications.

- The Apple menu has two sections. Choose the first command to see information about Macintosh Common Lisp. The rest of the menu provides access to desk accessories.
- Use the File menu to open and close editor windows, to print, and to quit Lisp.
- Use the Edit menu to manipulate text in the editor windows.

The last three menus are specific to Macintosh Common Lisp.

- The Eval menu provides tools for evaluating expressions, loading and compiling files, and interrupting and continuing program operation.
- The Tools menu gives access to a variety of programming aids. These are described in a later section.
- The Windows menu lists all the windows currently accessible in Macintosh Common Lisp.

The remainder of this section describes the menus in the menu bar.

Apple menu

One menu item on the Apple menu applies to Macintosh Common Lisp. This menu item, About Macintosh Common Lisp, displays a dialog box showing the version number of Macintosh Common Lisp.

The rest are systemwide menu items.

File menu

The File menu items, listed in Table 1-1, control files.

n **Table 1-1** File menu items

Menu item	Effect
New (-N)	Creates a new editor window, not initially associated with any file.
Open… (-O)	Allows you to select a text file and creates a new editor window for the file.
Open Selected File (-D)	Enabled only when there is a selection in the top editor buffer. Attempts to parse this selection as a pathname. If successful, creates an editor window for the pathname's file. The title of this menu item changes to reflect the contents of the current selection. For information on MCL pathname parsing, see the section "Pathname Structure" in Chapter 9, "File System Interface."
Close (-W)	Closes the current window. If the contents of the window have been edited since the last time it was saved, a dialog box asks you if you want to save or throw away the changes.
Save (-S)	Saves the contents of the active window into the file named in the title bar of the window. If the window is not associated with a disk file, Save As is invoked.
Save As...	Allows you to specify a directory and filename, and saves the contents of the active window to that directory / filename.
Save Copy As...	Allows you to specify a directory and filename, and saves a copy of the contents of the active window to that directory / filename.
Revert... (-R)	Reverts to the version of the window contents last saved to disk. Before the reversion occurs, you are asked to verify whether you really want to revert to the last version saved.
Page Setup...	Allows you to set printing options for the current printer.
Print... (-P)	Brings up the Print File dialog box.
Quit (-Q)	Closes all windows and exits the Lisp environment. If any window contains revisions that have not been saved to disk, you are given the option to save them. If you use the keyboard equivalent of this command, Macintosh Common Lisp asks you to confirm that you wish to exit

Edit menu

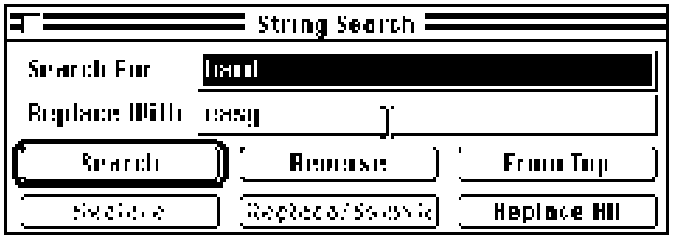
Table 1-2 shows the menu items that control editing in the Macintosh Common Lisp environment. The file `font-menus.lisp` in your MCL Examples folder supplements these standard commands with a set of font menus.

Table 1-2 Edit menu items

Menu item	Effect
Undo (-Z)	Undoes the last editor command, if possible. The Undo command applies to changes to the text, such as cutting, copying, pasting, and clearing, but does not undo movement of the insertion point. When the last command cannot be undone, Undo is disabled. When it is enabled, it usually indicates what it can undo by changing its name.
Undo More	Undoes in order editor commands before the last editor command, if possible. The same restrictions apply as with Undo.
Cut (-X)	Deletes the selected text and places it in the Clipboard and on the kill ring.
Copy (-C)	Copies the selected text to the Clipboard and to the kill ring.
Paste (-V)	Replaces the selected text with the text from the Clipboard. If there is no current selection, the text is simply inserted at the cursor location.
Clear	Deletes the current selection. The deleted text is not copied to the Clipboard. (However, like all deleted text, it *is* pushed onto the MCL kill ring.)
Select All (-A)	Selects the entire contents of the buffer associated with the active window.
Search... (-F)	Brings up the String Search dialog box with several options, including search-and-replace and replace-all. When a search causes the window to scroll (that is, when the found text isn't currently visible in the window), the found text is placed near the top of the window. The number of lines between the top of the window and the found text is determined by the global variable `*next-screen-context-lines*` .

Figure 1-3 illustrates how the Search command (Table 1-2) works.

n **Figure 1-3** The String Search dialog box



Eval menu

The Eval menu contains eight menu items, described in Table 1-3.

n **Table 1-3** Eval menu items

Menu item	Effect
Eval Selection (-E)	Evaluates the current selection in the top editor window. If there is no selection and the insertion point is next to a parenthesis, the expression bounded by the parentheses is evaluated.
Eval Buffer (-H)	Evaluates the entire contents of the top editor buffer. This is equivalent to Select All followed by Eval Selection.
Load... (-Y)	Presents the standard file dialog box and allows you to select a file to be loaded into Macintosh Common Lisp. You may load both `.lisp` and `.fasl` files (respectively, uncompiled and compiled Lisp files).
Compile File...	Allows you to select a file for compilation. You are asked to specify both the source and destination files, using the standard file dialog box.
Abort (-period)	Cancels the current computation and returns to the read-eval print loop. If Macintosh Common Lisp was in a break loop, it leaves the loop. A canceled computation cannot be resumed.

(continued)

n **Table 1-3** Eval menu items (continued)

Menu item	Effect
Break (-comma)	Suspends the current computation and enters a break loop. The state of the machine can be examined in the break loop (see Chapter 10, "Debugging and Error Handling"). You can resume the computation by choosing Continue or by calling the function `continue` .
Continue (-/)	Continues the last operation halted by a break or by a continuable error. This command is disabled when it is not applicable.
Restarts (-\)	Provides a list of possible ways to restart the current operation. This command is disabled when it is not applicable.

Tools menu

The tools menu contains eleven items, described in Table 1-4.

n **Table 1-4** Tools menu items

Menu item	Effect
Apropos	Brings up the Apropos dialog box, which performs `apropos` on a string entered by the user. The results are displayed in a table in the dialog box.
Documentation...	Gets a symbol name from the user and displays the documentation string for the symbol (if a documentation string is available). Documentation strings are available if the symbol was defined when the value of `*save-doc-strings*` was true and if the symbol definition contains a documentation string. Documentation strings are also available for all the external symbols in the `COMMON-LISP` and `CCL` packages if the MCL Help file is present in the folder containing the MCL application.
Edit Definition...	Allows the user to enter a symbol name and attempts to find the source code for the definition of the symbol. The source code file will be known if the symbol was defined when the value of the variable `*record-source-file*` was true.

(continued)

n **Table 1-4** Tools menu items (continued)

Menu item	Effect
Inspect	Brings up the Inspector Central window. The Inspector is described in Chapter 10, “Debugging and Error Handling.”
List Definitions	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. The dialog box contains a filter that can select only definitions containing a particular string. When the active window is not an editor window, this menu item is dimmed.
Search Files	Brings up a dialog box for searching a set of files for a given string. The user can use wildcard characters to specify the set of files to search. Macintosh Common Lisp supports Common Lisp extended wildcards, which are documented in Steele, pages 623–627, and also has its own wildcards; see the section “Wildcards” in Chapter 9, “File System Interface”
Backtrace (-B)	This command is enabled only when Macintosh Common Lisp is in a break loop. Choosing it brings up a stack backtrace. (For details, see Chapter 10, “Debugging and Error Handling.”)
Fred Commands	Brings up a window listing all the commands in Fred, the Macintosh Common Lisp editor. You can search through this list using the Fred search command, Control-S, and can copy text from it to another Fred window.
Listener Commands	Brings up a window listing all Listener commands that differ from Fred commands.
Print Options...	Brings up a dialog box that allows the user to set the values of several Lisp printer parameters.
Environment...	Brings up a dialog box that allows the user to set the values of several global variables affecting the Macintosh Common Lisp environment.

Windows menu

The titles of all visible windows appear as items in this menu. The menu is ordered by window layer. (That is, the front window—the window currently in use—is the first menu item and the back window is the last menu item.) To activate a window, select its menu item.

Editor buffers that have been changed but not saved are marked with a cross next to their names. The name of the currently selected window is dimmed and cannot be selected.

If you're not in the Listener window, you can select it by pressing Command-L.

To move the currently selected window to the back, hold down the Option key, then click the window's title bar.

To hide a window, click its close box while holding down the Command key. Its title will appear in italics in the Windows menu.

Customizing your work space

Macintosh Common Lisp allows you to customize your environment in several ways.

- Several global variables control aspects of the programming environment. You can set many of these in the dialog boxes available through the Print Options and Environment commands in the Tools menu. You can predefine all global variables in your `init` file (the file `init.lisp`).
- It is easy to customize your menu bar to include operations you perform frequently, or commands that are part of an application environment. The book *Getting Started With Macintosh Common Lisp* gives an extended example of creating commands and adding them to your menu bar; see the section "Making a Menu" in Chapter 3, "Prototyping." You can set your menu bar to the state you prefer in your `init` file.
- Finally, you can save snapshots of a particular MCL system state, including all of the above and all the states from files you have loaded. This environment loads more quickly than loading all the source and `fasl` files that were used to create it, and gives you the benefits of a completely known Lisp environment. For details see Appendix B, "Snapshots and Application Versions of Macintosh Common Lisp."

Global variables in Print Options and Environment

Macintosh Common Lisp provides a number of global variables defining the behavior of your system. You can customize the most frequently used ones by choosing Print Options and Environment from the Tools menu.

If you always want to boot Macintosh Common Lisp with these variables set to specific values, you should set them in your `init` file using `setq`.

Print Options global variables

The options in the Print Options dialog box implement Common Lisp printing functions as described in Chapter 22, "Input/Output," of *Common Lisp: The Language*. (See in particular Section 22.1.6.) For example, clicking `*PRINT-PRETTY*` in the Print Options dialog box sets the value of the Common Lisp `*print-pretty*` global variable to true.

Environment global variables

The options in the Environment dialog box set some frequently used MCL variables. Table 1-5 lists these variables and what they do. A checked variable has the value true; an unchecked one, the value `nil`.

Table 1-5 Environment options

Variable	Purpose
`*arglist-on-space*`	Displays the argument list of a function when user types space following open parenthesis and function name. (Does not parse; displays arglist of *any* symbol-name that follows an open parenthesis.) *Default is true;* displays argument list for functions defined with `*save-definitions*` true. If `nil`, does not display argument lists.
`*autoload-lisp-package*`	Provides backward compatibility with earlier versions of MCL. (It is usually preferable to upgrade your code.) *Default is* `nil`; `:lisp` package must be loaded by hand. If true, `:lisp` package is automatically loaded when required.
`*autoload-traps*`	Loads file containing definition of a referenced trap. *Default is true;* the file is loaded automatically. If `nil`, the file must be loaded by hand.

(continued)

n **Table 1-5** Environment options (continued)

Variable	Purpose
`*backtrace-on-break*`	Displays Stack Backtrace window when Macintosh Common Lisp enters a break loop. *Default is* `nil`; Backtrace must be selected on the Tools menu. If true, the Stack Backtrace window displays automatically.
`*break-on-errors*`	Enters a break loop on errors. *Default is true;* enters a break loop when an error is signaled. If `nil`, errors return to the read-eval-print loop.
`*break-on-warnings*`	Enters a break loop on warnings. *Default is* `nil`; warnings do not interrupt program flow. If true, enters a break loop when a warning is signaled.
`*emacs-mode*`	On Macintosh computers with no Control key, determines which key combination specifies MCL Control key. *Default is* `nil`; Command is Macintosh Command key and Command-Shift is MCL Control key. If true, Command is MCL Control key; Command-Shift is Macintosh Command key.
`*fasl-save-local-symbols*`	Provides a default value for the `:save-local-symbols` keyword argument to `compile-file`. *Default is* `nil`; local symbols are not saved in the compiled file. If true, local symbols are saved in the compiled file and are available when the file is loaded.
`*load-verbose*`	Determines whether Macintosh Common Lisp reports as it loads each file. *Default is* `nil`; only result of loading file is displayed. If true, Macintosh Common Lisp prints a message to `*standard-output*` as it loads a file.

(continued)

n **Table 1-5** Environment options (continued)

Variable	Purpose
`*paste-with-styles*`	Affects all commands that cause text to be pasted into a window. *Default is true;* style information is retained when text is copied and pasted. If `nil`, style information is discarded.
`*record-source-file*`	Determines whether compiler records source file of definitions. The definition contains a pointer to the source file. You can retrieve the definition by pressing Meta-period when the insertion point is next to the symbol name. *Default is true;* compiler records source file of all definitions, and Meta-period can retrieve source code. If `nil`, no record is kept and Meta-period cannot retrieve source code.
`*save-definitions*`	Determines whether lambda expressions are retained when functions are compiled. If lambda expressions are retained, then functions can be uncompiled and `step` will be useful. *Default is* `nil`; lambda expressions are not retained; `step` is not useful for compiled functions. If true, lambda expressions are retained and `step` is useful for compiled functions.
`*save-doc-strings*`	Determines whether documentation strings are retained. *Default is true;* documentation strings are retained. If `nil`, documentation strings are discarded.
`*save-fred-window-positions*`	Affects whether window size, position, and current selection of Fred windows are retained. *Default is true;* information is retained. If `nil`, information is discarded.
`*save-local-symbols*`	Determines whether names of arguments and local variables are saved when functions are compiled. *Default is true;* information is retained. If `nil`, information is discarded.

(continued)

n **Table 1-5** Environment options (continued)

Variable	Purpose
`*verbose-eval-selection*`	Determines whether results of intermediate evaluations are displayed when text is selected in Fred windows and evaluated. *Default is* `nil`; only final result of form is displayed. If true, result of each top-level form is displayed.
`*warn-if-redefine*`	Helps prevent accidental redefinition of a function defined somewhere else. *Default is true;* compiler issues a warning whenever a function, macro, or variable is redefined from a new file. If `nil`, compiler does not issue warnings when user functions are redefined (but does when built-in functions are redefined).

The `init` file

When you double-click the Macintosh Common Lisp icon, Macintosh Common Lisp attempts to load a file called `init.lisp` or `init.fasl` from its home folder (that is, the folder containing Macintosh Common Lisp). The file with the `.fasl` extension is a compiled file; the other is a source code file. If both files are found, the more recent one is loaded.

You can use the `init` file to create a default initial environment, for example, to set the state of various system parameters, load utility files, and so on.

You may want to customize some additional global variables in your `init` file. A possible variable is described next.

print-abbreviate-quote [*Variable*]

Description The *print-abbreviate-quote* variable determines whether lists whose first element is the symbol quote or the symbol function are printed specially.

If the value is true (the default), the lists are printed specially.

If the value is nil, they are not.

You can also set up Macintosh Common Lisp to install new menus, set up new logical hosts, and load and open files.

Here are examples of the kinds of statement found in a typical init file:

```
(setq *print-abbreviate-quote* t)
(setq *arglist-on-space* nil)
(setq *clear-mini-buffer* nil)
(setf (logical-pathname-translations "new-home")
        `(("**;*.*" ,(merge-pathnames ":**:*.*"
                      (mac-default-directory)))))
(load "MCL:Applications Development:my-menubar.lisp")
(open "MCL:Examples:online_index.text")
```

When you start Macintosh Common Lisp by double-clicking a file, the file is loaded but the init file is not.

If you find you are developing a very complex init file, you should investigate creating a heap image instead. See Appendix B, “Snapshots and Application Versions of Macintosh Common Lisp.”

Chapter 2 Editing in Macintosh Common Lisp

Contents

The MCL editor / 29
The editing window / 30
Working with the editor / 31
- Creating new windows and opening files / 31
- Adding text to a file / 31
- Saving buffers to files / 31
- The minibuffer / 32
- The kill ring and the Macintosh Clipboard / 32
- Multiple fonts / 33
- Packages / 34
  - Mode lines / 34
  - An `in-package` expression / 35
  - A `set-window-package` expression / 35
  - Finding a window's package / 35
- Fred parameters / 36
  - Normalizing `*next-screen-context-lines*` / 37

Editing in Macintosh style / 38
Editing in Emacs style / 39
- The Control and Meta modifier keys / 39
- Disabling dead keys / 40

Fred commands / 41
- Help, documentation, and inspection functions / 42
- Movement / 43
- Selection / 46
- Insertion / 49
- Deletion / 52
- Lisp operations / 54
- Window and file operations / 56
- Undo commands / 56
- Numeric arguments / 57

Incremental searching in Fred / 57
Performing an incremental search / 58
Making additional searches / 58
Backing up with the Delete key / 58
Terminating an incremental search / 59
Doing another incremental search / 59
Special incremental search keystrokes / 60
Boyer-Moore search / 61

This chapter discusses the MCL editor, Fred (Fred Resembles Emacs Deliberately), which combines the standard Macintosh multiple-window text editor with Emacs, the fully programmable editor that is a feature of most Lisp implementations.

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

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

The MCL editor

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

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

- Fred includes many specialized Lisp manipulation commands. For example, you can select complete or partial symbolic expressions, move from level to level of a symbolic expression, reindent them, get their documentation and argument list, and inspect them, all with simple keyboard commands.
  - Lisp expressions can be evaluated from Fred windows by pressing Enter (that is, the Enter key on the numeric keypad, not the Return key) with the cursor position at either end of a top-level expression. You can also highlight an expression and press Command-E.
  - Placing the insertion point after a close parenthesis, or before an open parenthesis, causes the matching parenthesis to blink. For example, placing the insertion point after a close parenthesis causes the matching open parenthesis to blink. This feature is very helpful in balancing parentheses.
  - Double-clicking after a close parenthesis, or before an open parenthesis, highlights to the matching parenthesis. For example, double-clicking before an open parenthesis highlights forward to the matching close parenthesis. This is another quick way to check the balance of your parentheses.
  - Pressing Tab after a Return indents the new line appropriately.
  - Pressing Control-Meta-Q reindents the current expression in a readable way.
  - Pressing Meta–close parenthesis moves the cursor into position for typing the next expression.
  - Other Fred commands get information on the argument list of a function or its documentation, inspect it, and edit its source file. Most of these are available both on the Tools menu and as keyboard commands.

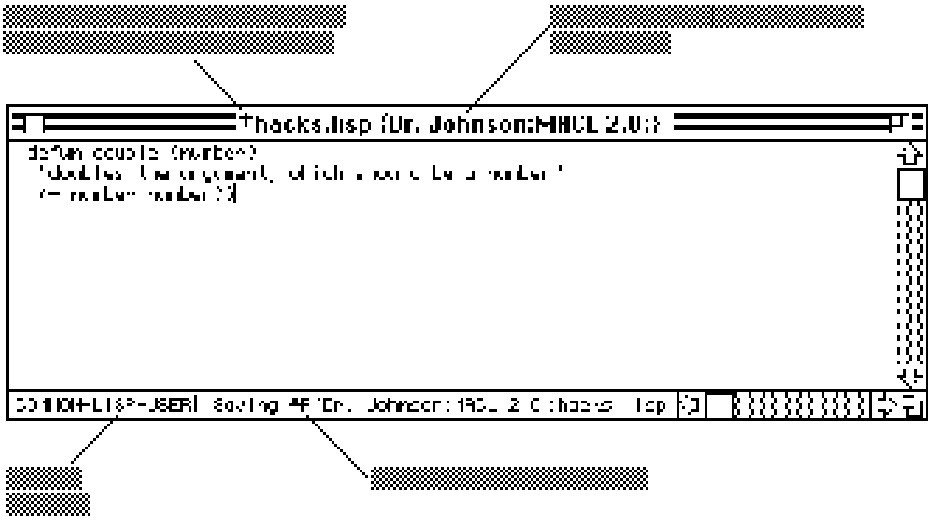
- Fred has online documentation of its own commands. Choose Fred Commands from the Tools menu to see a window of all Fred commands.
- Since Fred is written in Macintosh Common Lisp, it is completely programmable. Two changes users often make are binding Meta to the Escape key and adding a fonts menu. The file `escape-key.lisp` in the MCL Examples folder binds the Macintosh Escape key to Meta, and the file `font-menus.lisp` in the same folder adds a set of font menus.

The editing window

Figure 2-1 describes the parts of an editor window. At the top, in the title bar, is the pathname of the file contained in the window. The larger editing area contains text, which is in a buffer (a copy of the contents of a file). In this case, all of the window's contents are visible; however, the window always contains the whole buffer, whether or not all the contents are currently visible.

At the bottom of the window, the minibuffer displays the name of the window's package and other information.

- **Figure 2-1** An editor window



Working with the editor

This section gives general information on using the editor.

Creating new windows and opening files

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

Adding text to a file

Fred works with a mouse and the keyboard, just like other Macintosh text editors such as MacWrite. However, it understands Lisp and Lisp formatting better than those text editors. For example:

- Placing the insertion point after a close parenthesis causes the matching open parenthesis to blink.
- Double-clicking after a close parenthesis highlights backward to the matching open parenthesis.
- Highlighting one or more expressions and pressing Command-E, or choosing Eval Selection from the Eval menu, evaluates the selection.
- Pressing Tab after a Return indents the line appropriately.
- Pressing Control-Meta-Q reindents the current expression in a readable way.
- Pressing Meta–close parenthesis moves the cursor into position for typing the next expression.

See the sections “Editing in Emacs Style” and “Fred Commands” later in this chapter.

Saving buffers to files

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

A small cross to the left of the filename in the title bar of a Fred window indicates that the contents of the window have been altered since the buffer was last saved. (See Figure 2-1.)

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

The minibuffer

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

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

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

The kill ring and the Macintosh Clipboard

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

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

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

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

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

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

Multiple fonts

Fred has a limited multiple-font capability. Runs of characters may be in different fonts, and each window has a current insertion font. The insertion font is the font used when new characters are typed in the window.

Fred window fonts can be set programmatically, as described in Chapter 14, “Programming the Editor.” They can also be set through menus; a file in your MCL Examples folder,`font-menus.lisp` , shows how to add a set of font/size/style commands to the Edit menu.

When you set the font of a Fred window (either programmatically or through a menu), the resulting behavior depends on whether or not text has been selected. If text is selected, all the text in the selection is displayed in the new font, and the current insertion font does not change. If no text is selected, the current insertion font changes to the new font.

Fred does not automatically reset the insertion font when you move the insertion point. If the insertion font is 9-point Monaco, typed characters will appear in that font even if you place the insertion point between two characters displayed in Times bold. You must always set the insertion font explicitly by setting the window font when no text is selected.

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

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

Packages

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

A new, empty Fred window has no associated package.

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

Mode lines

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

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

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

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

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

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

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

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

An `in-package` expression

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

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

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

- *Note:* The search for the `in-package` form does not account for the read-time conditionals `#+` and `#-`.

A `set-window-package` expression

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

Finding a window's package

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

Fred parameters

The parameters in Table 2-1 can be used to control some of the behavior of Fred.

n **Table 2-1** Fred parameters

Variable	Purpose
`*arglist-on-space*`	Displays in the minibuffer the argument list of a function when a user types a space following an open parenthesis and function name. (Does not parse; displays argument list of *any* symbol name that follows an open parenthesis.) *Default is true;* displays argument list for functions. If `nil`, does not display.
`*clear-mini-buffer*`	Specifies whether to clear the minibuffer after each Fred command. If you operate with `*arglist-on-space*` true, you may wish to set this to `nil` so that argument lists persist long enough to use. *Default is true;* text is cleared from minibuffer after any Fred command is run. If `nil`, text is cleared from minibuffer only when being replaced by other text.
`*emacs-mode*`	For Macintosh keyboards with no Control key, determines which key combination specifies MCL Control key. *Default is* `nil`; Command is Macintosh Command key and Command-Shift is MCL Control key. If true, Command is MCL Control key; Command-Shift is Macintosh Command key.
`*fred-default-font-spec*`	Specifies which font is used when new Fred windows are opened. The initial value is `("Monaco" 9 :PLAIN)`.
`*mini-buffer-font-spec*`	Specifies the font used in minibuffers. The default value is `("Monaco" 9)`. Note that the size of minibuffers does not increase even when a large point size is used.

(continued)

n **Table 2-1** Fred parameters (continued)

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

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

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

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

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

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

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

Example

This function could be defined as follows.

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

Editing in Macintosh style

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

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

You can use almost any combination of MCL editing commands and Macintosh commands; Macintosh Common Lisp does not usually care how you combine them. The exception is that the MCL kill ring is more powerful than the Macintosh Clipboard, so the kill ring may overwrite the Clipboard information when you do not initially expect it.

Editing in Emacs style

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

The Control and Meta modifier keys

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

- The Emacs Control modifier is accessible through the Macintosh Control key or through the Command key (on Macintosh keyboards that don't have a Control key). In all MCL documentation, whichever key you are using to indicate Control is referred to as the Control key.
  - If the value of `*emacs-mode*` is `nil` and your Macintosh does not have a Control key, press Command-Shift to indicate the Control key and Command to choose a menu item.
  - If the value of `*emacs-mode*` is true and your Macintosh does not have a Control key, press the Command key to indicate the Control key and Command-Shift to choose a menu item.

  To issue a Control command, hold down the Control key or the Command key while you press the letter of the command. For example, to enter Control-X, hold down the Control key and press X. To enter Control-X Control-S (the Emacs Save command), hold down the Control key and press X, then continue to hold down the Control key and press S. To enter Control-X H (the Emacs Select Entire Buffer command), hold down the Control key and press X, then release the Control key and press H.
- The Emacs Meta modifier is accessed through the Macintosh Option key. You can rebind it to the Escape key or another key; see the note in this section. In all MCL documentation, the key you are using to indicate Meta is referred to as the Meta key.

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

u *Note:* If you would prefer to use the Escape key as a Meta key, load the file `escape-key.lisp` in the Examples folder. After you do, the Escape key works like the usual Lisp Meta key; press and release it before you press the command letter. The Option key remains a Meta key and works as it did before.

You do not need the Macintosh Option character set in ordinary MCL programming. You can insert a Macintosh Option character into Macintosh Common Lisp by quoting it; press Control-Q, then the character. For instance, you can insert the bullet sign, normally the Option-8 keystroke, by pressing Control-Q, then Option-8.

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

Disabling dead keys

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

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

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

  If you install a keyboard layout that does not support dead keys, you can insert a dead-key keystroke in Macintosh Common Lisp by quoting it. For example, you can generate the character ã by pressing Control-Q Control-N A.

n You can use the Escape key as a Meta key, as described in the previous section. If you do this regularly, load `escape-key.lisp` as part of your `init` file.

- Previously the dead-key difficulty was got around by using the functions `(set-dead-keys)` and `(get-dead-keys)`, which are now deprecated. They still work, but will be removed in a future release because they edit the 'KCHR' resource globally. (That is, they make dead keys undead for *all* Macintosh applications, not just Macintosh Common Lisp—clearly unfriendly behavior.)

  If you are still using these two functions, this is how they work:

  - `(set-dead-keys nil)` disables dead keys.
  - `(set-dead-keys (get-dead-keys))` reenables them. (The function `get-dead-keys` always returns the same value, a list of dead keys. You should not change this value.)

  If you use these functions, you must remember to reenable dead keys before you quit Macintosh Common Lisp.

Fred commands

The following Fred commands are defined in the initial MCL environment. Files in the Examples folder include additional Fred commands, and you can also write your own (as described in the section “Defining Fred Commands” in Chapter 14, “Programming the Editor”). Many commands are case insensitive; that is, you can press either Control-D or Control-Shift-D.

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

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

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

s **Important** If your keyboard does not have the Control key, indicate Control by using either Command (if the value of `*emacs-mode*` is true) or Command-Shift (if the value of `*emacs-mode*` is `nil`). Remember that Emacs Meta is equivalent to the Macintosh Option key unless you have loaded `escape-key.lisp` . s

Help, documentation, and inspection functions

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

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

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

(continued)

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

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

Movement

During editing, use the functions and keystrokes in Table 2-3 to move the insertion point. Most of these movement commands can be modified by the Shift key to establish or extend a selection; see Table 2-4.

n **Table 2-3** Fred commands for movement

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

(continued)

n **Table 2-3** Fred commands for movemen(continued)

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

Selection

The keystrokes in Table 2-4 are used to select text. You can modify most motion commands with the Shift key to select the region between the original insertion point and the new insertion point.

In addition, you can use the mouse to select text. If the value of the variable `*multi-click-count*` is non`nil` (the default), then Macintosh Common Lisp counts multiple clicks:

- Two clicks select a word.
- Three clicks select a line.
- Four clicks select the entire buffer.

Table 2-4 Fred commands for selection

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

(continued)

n **Table 2-4** Fred commands for selection (continued)

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

(continued)

n **Table 2-4** Fred commands for selection (continued)

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

Insertion

The keystrokes in Table 2-5 are used to insert text and space.

n **Table 2-5** Fred commands for insertion

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

(continued)

n **Table 2-5** Fred commands for insertion (continued)

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

(continued)

Table 2-5 Fred commands for insertion (continued)

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

Deletion

The keystrokes and functions in Table 2-6 are used to delete text and spaces.

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

n **Table 2-6** Fred commands for deletion

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

(continued)

n **Table 2-6** Fred commands for deletion (continued)

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

Lisp operations

The functions and keystrokes in Table 2-7 perform Lisp operations on the current expression.

Table 2-7 Fred commands for Lisp operations

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

(continued)

n **Table 2-7** Fred commands for Lisp operations (continued)

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

Window and file operations

The functions and keystrokes in Table 2-8 are used to save and select text manipulated in windows.

n **Table 2-8** Fred commands for window and file operations

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

Undo commands

The Undo command undoes the effect of previous commands. Functions and keystrokes associated with Undo are listed in Table 2-9. Successive insertions or deletions, or multiple replacements via the Search dialog, are considered a single command.

Each window has its own Undo history list.

n **Table 2-9** Fred commands for undoing commands

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

Numeric arguments

The keystrokes in Table 2-10 multiply the effect of any command to which they can be applied. (They can always be applied to motion and selection commands.)

n **Table 2-10** Fred commands for giving numeric arguments

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

Incremental searching in Fred

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

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

Performing an incremental search

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

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

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

Making additional searches

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

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

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

Backing up with the Delete key

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

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

If the last keystroke added a block of characters to the search string, pressing Delete removes the entire block. (See Control-Q, Control-W, and Control-Y in "Special Incremental Search Keystrokes" later in this chapter.)

Terminating an incremental search

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

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

Doing another incremental search

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

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

Special incremental search keystrokes

These keystrokes in Table 2-11 have special meanings in the context of an incremental search.

n **Table 2-11** Fred commands for searching

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

Boyer-Moore search

Macintosh Common Lisp now optionally supports the Boyer-Moore search algorithm, which is faster than the built-in algorithm. If you prefer to use this, load the file `boyer-moore.lisp` in your Examples folder.

Chapter 3 **Points and Fonts**

Contents

Points / 64
- How Macintosh Common Lisp encodes points / 64
- How Macintosh Common Lisp version 2 stores points / 64

MCL functions relating to points / 65

Fonts / 69
- Implementation of font specifications / 69
- Implementation of font codes / 70
- Functions related to font specifications / 71
- Functions related to font codes / 75

System data / 82

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

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

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

Points

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

How Macintosh Common Lisp encodes points

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

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

Points are always returned as a single encoded integer.

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

How Macintosh Common Lisp version 2 stores points

In Macintosh Common Lisp version 2, the integer that encodes the x and y coordinates of a point is automatically converted to a bignum if a fixnum cannot accommodate it. (For definitions of bignum and fixnum, see *Common Lisp: The Language*.)

In version 1.3, this integer was always a fixnum. There was no provision for overflowing into a bignum; if the integer became too big, it was simply truncated, leading to obscure bugs. Because points were always fixnums, they could be compared with `eq`.

In version 2, the range of fixnums is smaller than it was in version 1.3. In Macintosh Common Lisp version 1.3, `most-positive-fixnum` was 2*30 – 1; in version 2, it is 2*28 – 1. Except in cases where efficiency is paramount and the range is guaranteed to be below 4096, you should always assume that graphics points may be bignums. Because of this, `eq` can't safely be used to compare points; you must use `eql` as follows:

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

MCL functions relating to points

The following functions relate to points.

point-string [*Function*]

Syntax `point-string` *point*

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

Argument *point* A point.

Example

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

point-h

[*Function*]

Syntax `point-h` *point*

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

Argument *point* A point.

Example

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

point-v

[*Function*]

Syntax `point-v` *point*

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

Argument *point* A point.

Example

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

point<=

[*Function*]

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

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

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

make-point

[*Function*]

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

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

Arguments

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

Examples

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

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

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

add-points

[*Function*]

Syntax `add-points` *point1 point2*

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

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

Arguments

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

Example

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

subtract-points

[*Function*]

Syntax `subtract-points` *point1 point2*

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

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

Arguments

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

Example

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

Fonts

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

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

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

Font codes are described fully in *Inside Macintosh*.

Implementation of font specifications

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

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

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

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

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

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

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

The following are examples of legal font specifications:

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

Implementation of font codes

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

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

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

The font-face layout looks like this:

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

The mode-size layout looks like this:

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

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

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

Functions related to font specifications

The following functions implement and govern font specifications

real-font

[*Function*]

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

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

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

Argument *font-spec* A font specification.

font-spec

[*Function*]

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

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

Arguments

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

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

Example

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

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

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

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

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

string-width [*Function*]

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

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

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

Arguments

font-spec	A font specification.
string	A string.

Example

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

grafport-write-string [*Macro*]

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

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

Arguments

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

Example

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

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

font-info [*Function*]

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

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

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

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

Argument *font-spec* A font specification.

Example

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

view-font

[*Generic function*]

Syntax

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

Description

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

In the Listener, view-font removes boldface text, then calls the method ofwindow.

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

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

Arguments

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

set-view-font

[*Generic function*]

Syntax

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

Description

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

Arguments

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

Functions related to font codes

The following functions implement and govern font codes.

font-codes [*Function*]

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

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

Arguments

font-spec A font specification.

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

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

Examples

Here is an example of getting and reading font codes.

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

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

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

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

font-codes-info

[*Function*]

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

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

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

Arguments

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

Example

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

---

view-font-codes

[*Generic function*]

Syntax

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

Description

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

Arguments

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

Example

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

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

Syntax

set-view-font-codes (*view* simple-view) *ff ms* &optional *ff-mask ms-mask*

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

set-view-font-codes (*window* window) *ff ms* &optional *ff-mask ms-mask*

Description

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

Arguments

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

Example

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

grafport-font-codes

[*Function*]

Syntax `grafport-font-codes`

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

set-grafport-font-codes

[*Function*]

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

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

Arguments

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

wptr-font-codes [*Function*]

Syntax `wptr-font-codes` *wptr*

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

Argument *wptr* A window pointer.

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

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

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

Arguments

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

merge-font-codes [*Function*]

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

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

Arguments

old-ff	The old font/face code, expressed as a fixnum. A font/face code stores the encoded name of the font and its face (plain, bold, italic, and so on). If there is no *old-ff*, the value of *old-ff* is set to 0.
old-ms	The old mode/size code, expressed as a fixnum. A mode/size code indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size. If there is no *old-ms*, the value of *old-ms* is set to 0.

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

Examples

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

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

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

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

Here is a more condensed version of the same merging.

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

System data

The following symbols are bound to useful Macintosh system data.

font-list [*Variable*]

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

pen-modes [*Variable*]

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

Example

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

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

style-alist [*Variable*]

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

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

Example

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

white-pattern [*Variable*]
black-pattern [*Variable*]
gray-pattern [*Variable*]
light-gray-pattern [*Variable*]
dark-gray-pattern [*Variable*]

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

screen-width [*Variable*]
screen-height [*Variable*]

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

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

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

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

menubar-bottom [*Variable*]

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

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

Chapter 4 **Menus**

Contents

How menus are created / 87
A sample menu file / 87
The `menu-element` class / 88
The menu bar / 88
    Menu bar forms / 88
    Menu bar colors / 91
Menus / 93
    MCL forms relating to menus / 93
    MCL forms relating to elements in menus / 98
    MCL forms relating to colors of menu elements / 101
    Advanced menu features / 103
Menu items / 105
    MCL forms relating to menu items / 106
    MCL forms relating to menu item colors / 114
Window menu items / 115
    Window menu item functions / 116
    Window menu item class / 117
Updating the menu bar / 119
The Apple menu / 119
Example: A font menu / 120

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

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

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

A simple MCL application for editing menus, the Interface Toolkit, is documented in Chapter 8 of this manual and in Chapter 3, "Prototyping," of *Getting Started With Macintosh Common Lisp*. You can use the Interface Toolkit without knowing any of the information here. Before reading this chapter, especially if you are new to Macintosh Common Lisp, you might try out the Interface Toolkit to see how MCL menus work.

How menus are created

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

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

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

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

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

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

A sample menu file

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

The `menu-element` class

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

`menu-element` [*Class name*]

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

The menu bar

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

At any time, only one menu bar can be displayed. Other menu bars can be defined, however, and you can rotate among them.

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

Menu bar forms

The following MCL forms control menu bars.

`menubar` [*Class name*]

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

menubar

[*Variable*]

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

menubar

[*Function*]

Syntax `menubar`

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

Example

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

set-menubar

[*Function*]

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

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

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

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

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

Example

```
? (setq foo (menubar))
(#<Apple-Menu " ">  ;No Apple character in this font.
 #<Menu "File">
 #<Menu "Edit">
 #<Menu "Eval">
 #<Menu "Tools">
 #<Menu "Windows">)
;Assume a menu, MY-FROGS-MENU, whose title is "Tree Frogs":
? (set-menubar (list (car foo) my-frogs-menu))
(#<Apple-Menu " ">
 #<Menu "Tree Frogs">)
? (menubar)
(#<Apple-Menu " ">
 #<Menu "Tree Frogs">)
```

---

default-menubar

[*Variable*]

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

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

Example

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

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

find-menu

[*Function*]

Syntax `find-menu` *string*

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

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

Example

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

Menu bar colors

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

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

part-color

[*Generic function*]

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

Description The `part-color` generic function returns the color of *part*, a part of the menu bar. See Chapter 7, “Color,” for a description of color encoding.

Arguments

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

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

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

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

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

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

Example

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

set-part-color

[*Generic function*]

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

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

Arguments

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

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

color The new color, encoded as an integer. (See Chapter 7, "Color.")

Example

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

part-color-list [*Generic function*]

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

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

Argument *menubar* The current menu bar, the only instance of the class `menubar` .

Example

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

Menus

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

MCL forms relating to menus

The following MCL forms control menus.

menu [*Class name*]

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

initialize-instance [*Generic function*]

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

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

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

Arguments

menu A menu.

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

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

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

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

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

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

Example

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

menu-title

[*Generic function*]

Syntax menu-title (*menu* menu)

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

Argument

menu	A menu.

set-menu-title

[*Generic function*]

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

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

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

Arguments

menu	A menu.
new-title	A string.

Example

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

menu-install [*Generic function*]

Syntax menu-install (*menu* menu)

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

Argument *menu* A menu.

Example

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

menu-deinstall [*Generic function*]

Syntax menu-deinstall (*menu* menu)

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

You may reinstall the menu at a later time.

Argument *menu* A menu.

Example

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

menu-installed-p [*Generic function*]

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

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

Argument *menu* A menu.

Example

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

menu-disable

[*Generic function*]

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

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

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

Argument *menu* A menu.

menu-enable

[*Generic function*]

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

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

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

Argument *menu* A menu.

menu-enabled-p

[*Generic function*]

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

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

Argument *menu* A menu.

menu-style [*Generic function*]

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

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

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

Argument *menu* A menu.

menu-update-function [*Generic function*]

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

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

Argument *menu* A menu.

MCL forms relating to elements in menus

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

add-menu-items [*Generic function*]

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

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

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

Example

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

---

remove-menu-items

[*Generic function*]

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

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

Arguments

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

Example

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

menu-items [*Generic function*]

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

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

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

Arguments

menu A menu.

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

Example

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

find-menu-item [*Generic function*]

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

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

Arguments

menu A menu.

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

Example

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

help-spec

[*Generic function*]

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

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

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

MCL forms relating to colors of menu elements

Like the menu bar, menus and parts of menus can be colored.

part-color

[*Generic function*]

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

Description The `part-color` generic function returns the color of *part* , a part of the menu. See Chapter 7, “Color,” for a description of color encoding.

Arguments

menu A menu.

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

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

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

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

Example

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

set-part-color

[*Generic function*]

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

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

Arguments

menu	A menu.
part	A keyword specifying which part of the menu should be set. The keywords are the same as for `part-color`.
color	The new color, encoded as an integer. (See Chapter 7, "Color.")

Example

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

part-color-list

[*Generic function*]

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

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

Argument

menu	A menu.

Example

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

Advanced menu features

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

menu-update

[*Generic function*]

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

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

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

Argument *menu* A menu.

menu-handle

[*Generic function*]

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

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

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

Argument *menu* A menu.

Example

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

menu-id

[*Generic function*]

Syntax menu-id (*menu* menu)

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

Argument *menu* A menu.

Example

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

menu-id-object-alist

[*Variable*]

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

menubar-frozen

[*Variable*]

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

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

If the value of this variable is nil, the menu bar will be redrawn.

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

draw-menubar-if [*Function*]

Syntax `draw-menubar-if`

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

Menu items

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

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

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

You can specialize this behavior for your own menu items.

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

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

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

The `menu-item-action-function` method is executed at interrupt level, and further event processing is disabled while it is executed. Therefore, if a menu item initiates a lengthy process, the process shouldn't be executed directly as a `menu-item-action` ; instead, it should be inserted into the normal read-eval-print loop using the function `eval-enqueue` . For a complete description of `eval-enqueue` , see Chapter 11, "Events."

MCL forms relating to menu items

The following MCL forms are provided for programming menu items.

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

`menu-item` [*Class name*]

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

`initialize-instance` [*Generic function*]

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

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

Arguments

menu-item A menu item.

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

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

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

`:command-key`

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

`:menu-item-action`

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

`:disabled`

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

`:menu-item-colors`

A property list of part keywords and colors. See the `set-part-color` method for menu items, described in the section "MCL Forms Relating to Menu Item Colors" later in this chapter.

`:menu-item-checked`

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

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

`:update-function`

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

`:help-spec`

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

Example

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

menu-item-action [*Generic function*]

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

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

Argument *menu-item* A menu item.

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

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

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

Argument *menu-item* A menu item.

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

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

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

Arguments *menu-item* A menu item.

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

menu-item-title

[*Generic function*]

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

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

Argument *menu-item* A menu item.

set-menu-item-title

[*Generic function*]

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

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

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

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

Example

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

menu-item-disable

[*Generic function*]

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

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

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

menu-item-enable [*Generic function*]

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

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

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

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

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

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

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

command-key [*Generic function*]

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

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

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

set-command-key [*Generic function*]

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

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

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

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

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

Example

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

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

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

menu-item-check-mark

[*Generic function*]

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

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

Argument *menu-item* A menu item.

set-menu-item-check-mark

[*Generic function*]

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

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

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

Arguments *menu-item* A menu item.

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

Example

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

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

menu-item-style

[*Generic function*]

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

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

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

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

Example

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

set-menu-item-style

[*Generic function*]

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

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

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

Arguments

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

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

Example

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

menu-item-update [*Generic function*]

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

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

Argument *menu-item* A menu item.

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

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

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

Argument *menu-item* A menu item.

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

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

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

Arguments *menu-item* A menu item.

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

Example

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

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

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

MCL forms relating to menu item colors

The following functions control the coloring of the menu items.

`part-color` [*Generic function*]

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

Description The `part-color` generic function returns the color of the part of the menu item specified by *part*. See Chapter 7, "Color," for a description of color encoding.

Arguments

menu-item A menu item.

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

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

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

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

set-part-color [*Generic function*]

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

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

Arguments

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

part-color-list [*Generic function*]

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

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

Argument

menu-item	A menu item.

Window menu items

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

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

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

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

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

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

Window menu item functions

The menu items and their corresponding functions are given in Table 4-1.

Table 4-1 Window menu items

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

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

Some of these functions are internal to Macintosh Common Lisp.

Window menu item class

The following definitions control the behavior of window menu items.

window-menu-item [*Class name*]

Description This is the class of window menu items.

initialize-instance [*Generic function*]

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

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

Arguments

window-menu-item
A window menu item.

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

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

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

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

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

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

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

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

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

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

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

Updating the menu bar

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

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

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

The Apple menu

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

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

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

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

The Apple menu remains installed as you work on it.

Example–A font menu

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

Chapter 5 **Views and Windows**

Contents

The implementation of views and windows: The confusing part / 122
What simple views do / 122
What views do / 123
What windows do / 124
Class hierarchy of views / 124
Summary / 125
For more information / 126
MCL expressions relating to simple views and views / 126
Windows / 152
MCL functions for programming windows / 153
Advanced window features / 175
Supporting standard menu items / 181
Floating windows / 183

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

The relationship of dialogs and dialog items to views and windows is described in this chapter. They are defined in Chapter 6, “Dialog Items and Dialogs.”

The implementation of views and windows–The confusing part

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

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

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

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

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

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

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

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

What simple views do

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

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

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

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

(Because dialog items have many specialized subclasses and methods, they are described in a separate place, Chapter 6, “Dialog Items and Dialogs.”)

What views do

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

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

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

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

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

What windows do

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

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

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

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

- Fred windows, used by Fred, the editor. These windows have functionality for editing text.
- Floating windows (whose class is `windoid`), a special class of window that always appears in front of other windows. Floating windows are typically used for creating tool palettes.
- Dialogs, in which you display information and initiate action in structured ways. In Macintosh Common Lisp version 2, dialog items may appear in any view or subclass of view, not only in dialog boxes. The `dialog` class is preserved for compatibility with earlier versions of Macintosh Common Lisp, but it doesn't exist in any functional sense.

Class hierarchy of views

Figure 5-1 shows the class hierarchy of views from `simple-view` downward.

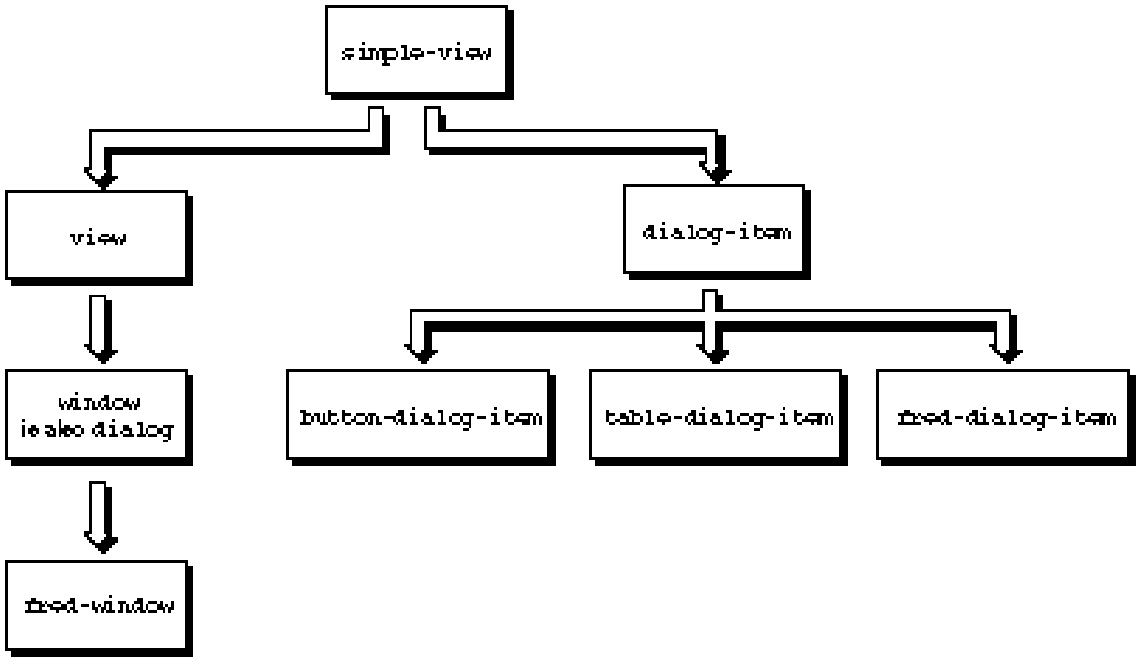
- The class `simple-view` is the parent of both `view` and `dialog-item`.
- The class `view` is the subclass of `simple-view` that defines the behavior of all views with subviews.
- The class `window` is a subclass of `view`, and `fred-window` and `dialog` (among others) are subclasses of `window`.

  The class `dialog` is simply `window` with slightly different default initial arguments, and dialog items do appear inside it exclusively; the class `window` and its subclasses are usable as dialog boxes.

- The class `dialog-item` is a subclass of `simple-view` because dialog items do not possess subviews.

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

- **Figure 5-1** The class hierarchy of views from `simple-view` downward



Summary

To summarize:

- Simple views have no subviews.
- Views have subviews.
- Views define graphics operations within other views.
- Windows define screen operations.
- Dialogs are windows with slightly different default values, good for dialog boxes.

- Fred windows have special methods to deal with, among other things, the display and editing of Lisp code and text.
- Dialog items are simple views since they have no subviews. They may appear in any view or window. The class `dialog-item` is never instantiated; only its subclasses have instances.

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

For more information

Dialog items and dialogs are described in Chapter 6, “Dialog Items and Dialogs.”

For information on the size, resolution, and other physical characteristics of the display, see Chapter 3, “Points and Fonts.”

Information on using color is given in Chapter 7, “Color.”

The event-related behavior of windows and views is described in Chapter 11, “Events.”

Information on drawing in views with QuickDraw is given in Appendix E, “QuickDraw Graphics.”

MCL expressions relating to simple views and views

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

`simple-view` [*Class name*]

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

initialize-instance [*Generic function*]

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

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

Arguments

view A simple view.

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

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

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

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

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

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

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

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

view

[*Class name*]

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

initialize-instance

[*Generic function*]

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

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

Arguments

view A view.

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

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

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

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

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

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

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

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

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

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

Example

Here is an example of a view being instantiated.

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

current-view [*Variable*]

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

mouse-view [*Variable*]

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

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

with-focused-view [*Macro*]

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

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

Arguments

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

Example

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

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

focus-view [*Generic function*]

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

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

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

Arguments

view	A view installed in a window, or `nil`. If `nil`, the current GrafPort is set to an invisible GrafPort.
font-view	A view or `nil`. If `nil`, the font is unchanged. If non-`nil`, the `view-font-codes` of *font-view* are installed after the rest of the focusing is completed. The default is `nil`. (See the section "Implementation of Font Codes" in Chapter 3, "Points and Fonts," for information on font codes.)

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

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

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

Arguments

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

Example

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

```
(defvar *w* (make-instance 'window))
(defvar *view* (make-instance 'view
                 :view-container *w*
                 :view-font '("Times" 12)
                 :view-size (view-size *w*)
                 :view-position #@(0 0)))
(with-pstrs ((s "Hello there."))
  (terpri *view*)
  (with-focused-view *view*
    ; This string will draw in the default font
    (#_DrawString s))
  (terpri *view*)
  (with-font-focused-view *view*
    ; This string will draw in times 12 font.
    (#_DrawString s)))
```

view-container [*Generic function*]

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

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

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

set-view-container [*Generic function*]

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

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

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

new-container The new container of the view.

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

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

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

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

Arguments

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

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

Syntax

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

Description

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

Argument

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

`subviews` [*Generic function*]

Syntax

`subviews` (*view* `view`) `&optional` *subview-type*

Description

The `subviews` generic function returns the subviews of *view*. If *subview-type* is present, only subviews matching that type are returned.

Arguments

view	A view.
subview-type	A Common Lisp type specifier.

`view-subviews` [*Generic function*]

Syntax

`view-subviews` (*view* `view`)

Description

The `view-subviews` generic function returns a vector containing all of the view's subviews. This vector should never be changed directly. It is updated automatically by calls to `set-view-container`.

Argument

view	A view.

do-subviews

[*Macro*]

Syntax `do-subviews (`*subview-var view* `[`*subview-type* `]) {`*form*`}*`

Description For each subview of *view* of the given *subview-type* , the macro `do-subviews` executes *form* with *subview-var* bound to the subview.

Arguments

subview-var	A variable.
view	A view.
subview-type	A Common Lisp type specifier.
form	Zero or more MCL forms.

Example

Here is how `do-subviews` might be used to define a method on `map-subviews` for `view`.

```
? (defmethod map-subviews ((view view) function
                           &optional subview-type)
  (if subview-type
    (do-subviews (subview view subview-type)
      (funcall function subview))
    (do-subviews (subview view)
      (funcall function subview))))
#<STANDARD-METHOD MAP-SUBVIEWS (VIEW T)>
```

map-subviews

[*Generic function*]

Syntax `map-subviews (`*view* `view)` *function* `&optional` *subview-type*

Description For each subview of *view* of the given *subview-type* , the generic function `map-subviews` calls *function* with the subview as its single argument.

Arguments

view	A view.
function	A function.
subview-type	A Common Lisp type specifier.

Example

Here is how `map-subviews` might be used to define a method on `subviews` for `view`.

```
? (defmethod subviews ((view view) &optional subview-type)
            (let ((result nil))
              (flet ((f (subview) (push subview result)))
                (declare (dynamic-extent #'f))
                (map-subviews view #'f subview-type))
              (nreverse result)))
#<STANDARD-METHOD SUBVIEWS (VIEW)>
```

view-named

[*Generic function*]

Syntax `view-named` *name* (*view* `view`)

Description The `view-named` generic function returns the first subview of *view* whose nickname is *name*. The subviews are searched in the order in which they were added to *view*.

Arguments

name	Any object, but usually a symbol. Nicknames are compared using `eq`.
view	A view.

Example

Here is an example of using `view-named` to find a button nicknamed `pearlie` in the dialog `dialog1`.

```
? (view-named 'pearlie dialog1)
#<RADIO-BUTTON-DIALOG-ITEM #x374BA9>
```

find-named-sibling

[*Generic function*]

Syntax `find-named-sibling` (*view* `simple-view`) *name*

Description The `find-named-sibling` generic function performs a search in *view* 's container and returns the first item in the container whose nickname is *name*. For example, given a dialog item *view* , it performs a search in the view that is *view* 's container to find another item with the nickname *name*. The items are searched in the order in which they were added to *view* 's container.

Arguments

view	A simple view.
name	Any object, but usually a symbol. Nicknames are compared using `eq`.

Example

The generic function `find-named-sibling` might be implemented as follows.

```
? (defmethod find-named-sibling ((view simple-view) name)
    (let ((container (view-container view)))
      (and container (view-named name container))))
```

---

add-subviews

[*Generic function*]

Syntax `add-subviews` (*view* `view`) `&rest` *subviews*

Description The `add-subviews` generic function sets the container of each of *subviews* to *view* .

If any of the subviews are already owned by *view* , `add-subviews` does nothing.

Arguments

view	A view.
subviews	A view or simple view, but not a window. *subviews* must be able to be contained within *view* .

Examples

This function could be defined as follows:

```
? (defmethod add-subviews ((view view) &rest subviews)
    (dolist (su subviews)
      (set-view-container su view)))
#<STANDARD-METHOD ADD-SUBVIEWS (VIEW)>
```

The following code adds a checkbox to a window, then checks to see whether it's there:

```
? (setf bim (make-instance 'window))
#<WINDOW "Untitled" #x4E42A9>
? (setf boxy (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4E5249>
? (add-subviews bim boxy)
NIL
? (subviews bim)
(#<CHECK-BOX-DIALOG-ITEM #x4E5249>)
```

remove-subviews

[*Generic function*]

Syntax `remove-subviews` (*view* `view`) `&rest` *subviews*

Description The `remove-subviews` generic function removes each of *subviews* from *view* .

If a subview is not in *view* , an error is signaled.

Arguments

view	A view.
subviews	A view or simple view, but not a window; *subviews* must be able to be contained within *view* .

find-clicked-subview

[*Generic function*]

Syntax

`find-clicked-subview` (*view* `simple-view`) *where*
`find-clicked-subview` (*view* `view`) *where*
`find-clicked-subview` (*view* `null`) *where*

Description The `find-clicked-subview` generic function returns the subview of *view* that contains the point *where* in its click region. The method for `null` searches all windows for a subview containing *where* in its click region.

This function is similar to `find-view-containing-point` , but `find-clicked-subview` calls `point-in-click-region-p` , and `find-view-containing-point` calls `view-contains-point-p` . The default method of `point-in-click-region-p` for views or simple views simply calls `view-contains-point-p` , but users can write methods to make views invisible to mouse clicks.

Arguments

view	A view or subview.
where	A point in the local coordinate system of the view's container.

view-corners

[*Generic function*]

Syntax `view-corners` (*view* `simple-view`)
`view-corners` (*window* `window`)

Description The `view-corners` method for `simple-view` returns two points, the upper-left and lower-right corners of *view* . The method for `window` returns the view size.

Arguments

view	A simple view or subclass of `simple-view` .
window	A window.

Example

```
? (view-corners (make-instance 'view
                  :view-position #@(10 20)
                  :view-size #@(30 40)))
1310730
3932200
? (point-string 1310730)
"#@(10 20)"
? (point-string 3932200)
"#@(40 60)"
```

invalidate-corners

[*Generic function*]

Syntax `invalidate-corners` (*view* `simple-view`) *topleft bottomright* `&optional` *erase-p*

Description The `invalidate-corners` generic function calls the Macintosh trap `#_InvalRgn` on the rectangle formed by *topleft* and *bottomright* in *view* .

Arguments

view	A simple view.
topleft	The upper-left corner of the rectangle to invalidate.
bottomright	The lower-right corner of the rectangle to invalidate.
erase-p	A value indicating whether or not to add the invalidated rectangle to the erase region of *view* 's window. The default is `nil`.

invalidate-view

[*Generic function*]

Syntax

`invalidate-view` (*view* `simple-view`) `&optional` *erase-p*
`invalidate-view` (*view* `view`) `&optional` *erase-p*

Description

The `invalidate-view` generic function invalidates *view* by running `invalidate-corners` on the region bounded by its `view-corners`.

Arguments

view	A view or simple view.
erase-p	A value indicating whether or not to add the invalidated region to the erase region of *view*'s window. The default is `nil`.

Example

For examples of the use of `invalidate-view`, see in your MCL Examples folder the files `view-example.lisp` and `text-edit-dialog-item.lisp`.

invalidate-region

[*Generic function*]

Syntax

`invalidate-region` (*view* `simple-view`) *region* `&optional` *erase-p*

Description

The `invalidate-region` generic function focuses on the view and calls `#_InvalRgn`. If the value of *erase-p* is true, the function adds this region to *view*'s window erase region; the next time `window-update-event-handler` runs, it will be erased. If *erase-p* is `nil` and the window was created with the `:erase-anonymous-invalidations` initarg set to true (the default), the function adds this region to the window's explicit invalidation region; `window-update-event-handler` will not erase it.

The function `invalidate-region` is called by `invalidate-view` and `invalidate-corners`, and indirectly by `set-view-position`, `set-view-size`, and `set-view-container`.

Arguments

view	A simple view.
region	The region to invalidate. The region must be a Macintosh region handle, that is, the result of `(#_NewRgn)`.
erase-p	A value indicating whether or not to add the invalidated view to the erase region of *view*'s window. The default is `nil`.

validate-corners [*Generic function*]

Syntax

`validate-corners` (*view* `simple-view`) *topleft bottomright*
`validate-corners` (*view* `view`) *topleft bottomright*

Description

The `validate-corners` generic function erases the previous contents of the rectangle formed by *topleft* and *bottomright* and calls `#_ValidRgn` on the rectangle. It also removes the rectangle from the erase region of *view*'s window

Arguments

view	A view or simple view.
topleft	The upper-left corner of the view to invalidate.
bottomright	The lower-right corner of the view to invalidate.

validate-view [*Generic function*]

Syntax

`validate-view` (*view* `simple-view`)
`validate-view` (*view* `view`)

Description

The `validate-view` generic function validates *view* by running `validate-corners` on the region bounded by its `view-corners`.

Argument

view	A view or simple view.

validate-region

[*Generic function*]

Syntax `validate-region` (*view* `simple-view`) *region*

Description The `validate-region` generic function focuses on the view and calls `#_ValidRgn` , removing the region from *view* 's window erase region and explicit invalidation region.

Arguments

view	A simple view.
region	A region. The region must be a Macintosh region handle, that is, the result of `(#_NewRgn)` .

wptr

[*Generic function*]

Syntax `wptr` (*view* `simple-view`)

Description The `wptr` generic function holds the pointer to a window record on the Macintosh heap. This record can be examined or the pointer passed to Macintosh traps that require a window pointer.

This generic function returns a window pointer if the view is contained in a window, or `nil` if the view is not contained in a window.

All views contained in a given window have the same `wptr` .

Argument

view	A simple view or subclass of `simple-view` .

Examples

Both a view and its subview have the same `wptr` .

```
? (setf bim (make-instance 'window))
#<WINDOW "Untitled" #x4E42A9>
? (setf boxy (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4E5249>
? (add-subviews bim boxy)
NIL
? (wptr boxy)
#<A Mac Zone Pointer Size 156 #x2C35B4>
? (wptr bim)
#<A Mac Zone Pointer Size 156 #x2C35B4>
```

You can test if a view's window has been closed by checking whether the value of its `wptr` slot is `nil`.

```
? (window-close bim)
NIL
? (wptr bim)
NIL
? (wptr boxy)
NIL
```

view-window [*Generic function*]

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

Description The `view-window` generic function returns the window containing *view* , or `nil` if the view is not contained in a window. If *view* is a window, `view-window` returns the window.

Argument *view* A simple view or subclass of `simple-view` .

Example

This code checks to determine that a simple view (a checkbox dialog item) is contained in a window:

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (setf win (make-instance 'window))
#<WINDOW "Untitled" #x4CFBE9>
? (add-subviews win checkbox)
NIL
? (view-window checkbox)
#<WINDOW "Untitled" #x4CFBE9>
```

view-position [*Generic function*]

Syntax `view-position (`*view* `simple-view)`

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

Argument *view* A view or simple view.

Example

This code returns the position of `checkbox`, a checkbox dialog item:

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (view-position checkbox)
262148
```

set-view-position

[*Generic function*]

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

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

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

Arguments

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

Example

This code sets the position of `checkbox`, a checkbox dialog item:

```
? (setf checkbox (make-instance 'check-box-dialog-item))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
? (set-view-position checkbox #@(20 20))
1310740
? (point-string 1310740)
"#@(20 20)"
```

view-default-position

[*Generic function*]

Syntax `view-default-position (`*view* `simple-view)`

Description The method of `view-default-position` for `simple-view` returns `#@(0 0)`. This function is called to determine the default value of the `:view-position` initarg of *view*.

Argument

view	A simple view or subclass of `simple-view`.

view-size [*Generic function*]

Syntax `view-size (`*view* `simple-view)`

Description The `view-size` generic function returns the size of the view.

Argument *view* A simple view or subclass of `simple-view`.

Example

This code returns the size of `checkbox`, a checkbox dialog item:

```
? (view-size checkbox)
1048596
```

set-view-size [*Generic function*]

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

Description The `set-view-size` generic function sets the size of the view.

Arguments

view	A simple view or subclass of `simple-view`.
h	The width of the new size, or the complete size (encoded as an integer) if *v* is `nil` or not supplied.
v	The height of the new size, or `nil` if the complete size is given by *h*.

view-default-size [*Generic function*]

Syntax `view-default-size (`*view* `simple-view)`

Description The method of `view-default-size` for `simple-view` returns `#@(100 100)`. This function is called to determine the default value of the `:view-size` initarg of *view*.

Argument *view* A simple view or subclass of `simple-view`.

view-scroll-position [*Generic function*]

Syntax `view-scroll-position (`*view* `simple-view)`

Description The `view-scroll-position` generic function returns the current scroll position of the view, which is the coordinate of the upper-left corner of the view. This position corresponds to the origin of a Macintosh GrafPort.

Argument *view* A simple view or subclass of `simple-view`.

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

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

Description The generic function `set-view-scroll-position` sets the position of the view's scroll position. It is usually called in response to a mouse click in a scroll bar. The function returns `nil`.

Arguments

view A simple view or subclass of `simple-view`.

h The horizontal coordinate of the new scroll position, or the complete scroll position (encoded as a point) if *v* is `nil` or not supplied.

v The vertical coordinate of the new scroll position, or `nil` if the complete scroll position is given by *h*.

scroll-visibly An argument specifying whether the scrolling is done immediately. If true, the function calls `#_ScrollRect` to do the scrolling immediately. Otherwise, the function invalidates the view so that it is redrawn the next time `window-update-event-handler` is called.

Example

```
? (setq foo (make-instance 'fred-window))
#<FRED-WINDOW "New" #x438D21>
? (view-scroll-position foo)
0
? (set-view-scroll-position foo 20 20)
NIL
```

view-nick-name

[*Generic function*]

Syntax

`view-nick-name (`*view* `simple-view)`
`view-nick-name (`*view* `view)`

Description

The `view-nick-name` generic function returns the nickname of the view. The nickname is used in conjunction with `view-named`.

Argument

view A view or simple view.

set-view-nick-name

[*Generic function*]

Syntax

`set-view-nick-name (`*view* `view)` *new-name*

Description

The `set-view-nick-name` generic function sets the nickname of the view to *new-name* and returns *new-name*.

Arguments

view A view or simple view.

new-name A name, usually a symbol or string.

find-view-containing-point

[*Generic function*]

Syntax

`find-view-containing-point (`*view* `view)` *h* `&optional` *v direct-subviews-only*
`find-view-containing-point (`*view* `null)` *h* `&optional` *v direct-subviews-only*

Description

The generic function `find-view-containing-point` returns the view containing the point specified by *h* and *v*. This may be the view or one of its subviews.

The null method searches all windows for a view that contains the point. The `null` class and its use are documented in *Common Lisp: The Language*, pages 780–783.

Arguments

view	A view.
h	The horizontal coordinate of the point, or the complete point if *v* is not supplied.
v	The vertical coordinate of the point.
direct-subviews-only	If *direct-subviews-only* is `nil` (the default), the most specific view is returned; subviews are searched for subviews, and so on. If true, then only the view or one of its direct subviews is returned.

Examples

This code determines the subview of the window `win` that contains the point `#@(21 21)`.

```
? (find-view-containing-point win #@(21 21))
#<CHECK-BOX-DIALOG-ITEM #x4CF721>
```

The following code returns the view that contains the mouse, when you don't know which window it's over:

```
(find-view-containing-point nil (view-mouse-position nil))
```

view-contains-point-p

[*Generic function*]

Syntax

`view-contains-point-p` (*view* `simple-view`) *where*
`view-contains-point-p` (*window* `window`) *where*

Description

The generic function `view-contains-point-p` returns `t` if *view* contains *where;* otherwise it returns `nil`. The method for `simple-view` takes *where* in the coordinates of the parent view; the method for `window` uses its own coordinates.

Arguments

view	A simple view or view.
window	A window.
where	The cursor position in the local coordinate system of the view's container when the mouse is clicked. If *view* is a window, the cursor position in the window's coordinate system.

point-in-click-region-p [*Generic function*]

Syntax `point-in-click-region-p (`*view* `simple-view)` *where*

Description The generic function `point-in-click-region-p` is called by `view-click-event-handler` to determine whether *where* is in *view* . The default method calls `view-contains-point-p` .

Arguments

view A simple view or view.

where For a view, the cursor position of the view in the local coordinate system when the mouse is clicked. For a simple view, the cursor position of the simple view in the local coordinate system of the view's container when the mouse is clicked.

view-activate-event-handler [*Generic function*]

Syntax

`view-activate-event-handler (`*view* `simple-view)`
`view-activate-event-handler (`*view* `view)`

Description The generic function `view-activate-event-handler` is called by the event system when the window containing the view is made active.

The definition for `simple-view` does nothing. The definition for `view` calls `view-activate-event-handler` on each subview. Specialize this generic function if your view needs to indicate visually that it is active.

Argument

view A simple view or view.

view-deactivate-event-handler [*Generic function*]

Syntax

`view-deactivate-event-handler (`*view* `simple-view)`
`view-deactivate-event-handler (`*view* `view)`

Description The generic function `view-deactivate-event-handler` is called by the event system to deactivate a view. It is called when the window containing the view is active and a different window is made active.

The definition for `simple-view` does nothing. The definition for `view` calls `view-deactivate-event-handler` on each subview. Specialize this generic function if your view needs to indicate visually that it has been deactivated.

Argument *view* A simple view or view.

`view-click-event-handler` [*Generic function*]

Syntax `view-click-event-handler` (*view* `simple-view`) *where*
`view-click-event-handler` (*view* `view`) *where*

Description The generic function `view-click-event-handler` is called by the event system when a mouse click occurs. The `simple-view` method does nothing. The `view` method calls `view-convert-coordinates-and-click` on the first subview for which `point-in-click-region-p` returns `t`.

The function `view-click-event-handler` scans subviews in the opposite order as does `view-draw-contents`. The first view added is the first one drawn but the last one to be queried during clicking.

If you define any `view-click-event-handler` methods for `window`, they must call `call-next-method`.

Arguments *view* A simple view or view.

where For a view, the mouse click position (the position when the mouse is clicked) of the view in the local coordinate system. For a simple view, the mouse click position of the simple view in the local coordinate system of the view's container.

Example

This function might be defined as follows, except that it does not do any consing:

```
? (defmethod view-click-event-handler-1 ((view view) where)
    (dolist (subview (nreverse (subviews view)) view)
      (if (point-in-click-region-p subview where)
        (return
          (view-convert-coordinates-and-click
              subview where view)))))
#<STANDARD-METHOD VIEW-CLICK-EVENT-HANDLER-1 (VIEW T)>
```

For further examples, see the files `grapher.lisp`, `shapes-code.lisp`, `thermometer.lisp`, and `view-example.lisp` in your MCL Examples folder.

view-convert-coordinates-and-click [*Generic function*]

Syntax

`view-convert-coordinates-and-click (`*view* `simple-view)` *where container*

`view-convert-coordinates-and-click (`*view* `view)` *where container*

Description

The generic function `view-convert-coordinates-and-click` runs `view-click-event-handler` on the cursor position within the view's container.

Arguments

view	A simple view or view.
where	For a view, the mouse click position (the position when the mouse is clicked) of the view in the local coordinate system. For a simple view, the mouse click position of the simple view in the local coordinate system of the view's container.
container	The view's container.

view-draw-contents [*Generic function*]

Syntax

`view-draw-contents (`*view* `simple-view)`
`view-draw-contents (`*view* `view)`

Description

The generic function `view-draw-contents` is called by the event system whenever a view needs to redraw any portion of its contents.

The default `simple-view` method does nothing. It should be shadowed by views that need to redraw their contents. The default `view` method calls `view-focus-and-draw-contents` on each of the view's subviews.

When `view-draw-contents` is called by the event system, the view's clip region is set so that drawing occurs only in the portions that need to be updated. This normally includes areas that have been covered by other windows and then uncovered.

Argument

view	A simple view or view.

view-focus-and-draw-contents [*Generic function*]

Syntax

```
view-focus-and-draw-contents (view simple-view)
  &optional visrgn cliprgn
view-focus-and-draw-contents (view view) &optional
  visrgn cliprgn
```

Description The generic function `view-focus-and-draw-contents` is used whenever a view needs to be focused on before any portion of its contents is redrawn. The method for `view` focuses on the view, then calls `view-draw-contents` if the *visrgn* and *cliprgn* region records overlap. The method for `simple-view` focuses on the view's container, then calls `view-draw-contents` .

Arguments

view	A simple view or view.
visrgn, *cliprgn*	Region records from the view's `wptr`.

Example

The method of `view-focus-and-draw-contents` for `simple-view` shows the use of the region record arguments.

```
(defmethod view-focus-and-draw-contents
                ((view simple-view) &optional visrgn cliprgn)
  (with-focused-view (view-container view)
    (when (regions-overlap-p visrgn cliprgn)
      (view-draw-contents view))))
```

The function `regions-overlap-p` takes two arguments, which must be pointers to Macintosh regions as returned by `(#_NewRgn)` . It returns true if they have a nonempty intersection and `nil` if they do not.

convert-coordinates [*Function*]

Syntax `convert-coordinates` *point source-view destination-view*

Description The `convert-coordinates` function converts *point* from the coordinate system of *source-view* to the coordinate system of *destination-view* .

The source view and destination view should be in the same view hierarchy (that is, they should have a common container, or one should be contained in the other).

Arguments

point	A point, encoded as an integer.
source-view	A view in whose coordinate system *point* is given.
destination-view	A view in whose coordinate system the return point is given.

Example

Here is a way of defining `view-convert-coordinates-and-click` by means of `convert-coordinates`.

```
? (defmethod view-convert-coordinates-and-click
     ((view view) where container)
   (view-click-event-handler view
     (convert-coordinates where container view)))
#<STANDARD-METHOD VIEW-CONVERT-COORDINATES-AND-CLICK (VIEW T T)>
```

Windows

Windows are a subclass of `view`. Their behavior is specialized on that of `view`, and they inherit slots from `view`. Windows may contain subviews, but a window cannot be a subview. (If they could, windows would attempt to display inside windows, and that is wrong: windows display views.)

Windows are used to display information on the screen. Because windows are views, graphics operations can also be performed on them. For many applications, the distinction between a window and a view is insignificant and you don't need to worry about views at all. You can simply work with windows, using both window and view operations.

The base class of windows is `window`. The features of `window` are common to all windows.

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

- `fred-window`, a subclass of windows used for text editing. The functionality of Fred windows is discussed in Chapter 14, "Programming the Editor."
- `windoid`, the class of floating windows. Floating windows always appear in front of other windows. You generally use them to create tool palettes. They are described in the section "Floating Windows" later in this chapter.

- `dialog`. The `dialog` class exists for convenience. It is a subclass of the `window` class and is identical except that its default window type is `:document` instead of `:document-with-zoom`, its default title is `"Untitled Dialog"` instead of `"Untitled"`, its default size is `#@(300 200)` instead of `*window-default-size*`, and its default position is `'(:top 100)` instead of `*window-default-position*`.

  You do not need to use the `dialog` class. You can use any window to create a dialog box, and dialog items can appear in any window.

  Dialogs are described in Chapter 6, “Dialog Items and Dialogs.”

MCL functions for programming windows

The following MCL functions are used for creating, reporting on, and modifying windows.

`window` [*Class name*]

Description The class `window` is the class of windows, built on `view`.

`initialize-instance` [*Generic function*]

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

Description The `initialize-instance` primary method for `window` initializes a window so that it can be used. (You make an instance with `make-instance`, which calls `initialize-instance`.)

Arguments

window A window.

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

`:view-position`
A point, keyword, or list giving the initial position of the window. The default is the result of calling `view-default-position` on the window. For a description of the list form of `view-position`, see the generic function `set-view-position` later in this section.

`:view-size`
A point giving the initial size of the window. The default is the result of calling `view-default-size` on the window.

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

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

`:view-subviews`
A list of initial subviews for the window.

`:window-title`
A string specifying the title of the window. The default is `"Untitled"`.

`:window-show`
If this argument is true (the default), a window is shown when it is created. If `nil`, the window is created invisibly. See `window-show` and `window-hide`.

`:view-font`
The font specification used by the window. The default is the result of calling `view-default-font` on the window.

`:window-layer`
: An integer describing the layer in which the new window will be created. By default this is `0` (the front window). For details, see `set-window-layer`, later in this section.

`:color-p`
: If `nil` (the default), the window is a normal window created by the `#_newWindow` trap. If non-`nil`, the window is a color window, created by the `#_newCWindow` trap.

`:window-type`
: A keyword describing the type of window to be created. The default is `:document-with-zoom`. This argument should be one of the following keywords:

```
:document
:document-with-grow
:document-with-zoom
:double-edge-box
:single-edge-box
:shadow-edge-box
:tool
```

`:procid`
: A number indicating the window definition ID (procID) of the window to be created. This is an alternative to specifying `:window-type`, for programmers who want to use window definitions with nonstandard IDs.

`:window-do-first-click`
: A Boolean value determining whether the click that selects a window is also passed to `window-click-event-handler`. The default value is `nil`.

  The click that selects an application in Multifinder is not passed to the application unless either the window clicked on is not the front window or the Get Front Clicks bit is set in the application's size resource.

`:close-box-p`
: A Boolean value determining whether the window will have a close box. Close boxes aren't available on all windows.

`:wptr` For use by advanced programmers, an argument used as a pointer to a window record on the Macintosh heap. Instead of creating a new window, `initialize-instance` builds a window object around the window specified by `:wptr`. This is useful when you want to create the window yourself and integrate it with the MCL window object system.

`:erase-anonymous-invalidations`
An argument determining behavior when *window* is refreshed. If the value of this initialization argument is true (the default), any parts of the invalid region of *window* that were not added by `invalidate-region` are erased when *window* is refreshed. If this value is `nil`, no extra erasing is done. Since erasing draws the background color and background pattern, and since anonymous invalidation usually happens only because a formerly covered part of the window is exposed, you usually should use the default. (The function `invalidate-region` is called by `invalidate-view` and `invalidate-corners`, and indirectly by `set-view-position`, `set-view-size`, and `set-view-container`.) If your code invalidates parts of a window without calling `invalidate-region`, for example, by calling `#_InvalRgn`, you may notice flickering on redraw if you use the default value of `:erase-anonymous-invalidations`.

Example

Here is an example of instantiating a window.

```
? (setq baz (make-instance 'window
              :window-title "Bazwin"
              :view-position #@(200 300)
              :window-type :tool
              :color-p t))
#<WINDOW "Bazwin" #x5DB8C9>
```

windows

[*Function*]

Syntax `windows &key :class :include-invisibles :include-windoids`

Description The `windows` function returns a list of existing windows that are instances of `:class`. The list is ordered from front to back.

Arguments

`:class` A class used to filter output. Only windows that match the value of `:class` are included in the returned list. The default is `window`, which includes all windows.

`:include-invisibles`
If the value of this variable is true, invisible windows are included in the list. If false (the default), invisible windows are not included.

`:include-windoids`
If the value of this variable is true, floating windows (the class `windoid`) are included in the list. If false (the default), floating windows are not included. Floating windows are also included if the value of the `:class` argument is `windoid`.

Examples

Here are some examples of the use of `windows`.

```
? (windows)
(#<LISTENER "Listener" #x49EB31>
 #<APROPOS-DIALOG "Apropos" #x532EF1>
 #<FRED-WINDOW "New" #x51CC61>)
? (windows :class 'fred-window)
(#<LISTENER "Listener" #x49EB31>
 #<FRED-WINDOW "New" #x51CC61>)
? (windows :class 'apropos-dialog)
(#<APROPOS-DIALOG "Apropos" #x532EF1>)
```

front-window

[*Function*]

Syntax `front-window &key :class :include-invisibles :include-windoids`

Description The `front-window` function returns the frontmost window satisfying the arguments. If no windows satisfy the tests, `nil` is returned.

Arguments

`:class` A class used to filter output. The frontmost window that is an instance of the value of `:class` is returned. The default is `window`, which includes all windows.

`:include-invisibles`
If the value of this variable is true, the frontmost window, visible or invisible, is returned. If false (the default), the frontmost visible window is returned.

`:include-windoids`
If the value of this variable is true, the frontmost window or floating window is returned. If false (the default), the frontmost window that is not a floating window is returned.

Example

```
? (front-window)
#<LISTENER "Listener" #x5204C9>
```

---

target

[*Function*]

Syntax `target`

Description The `target` function returns the second window on the list of windows; it is equivalent to `(second (windows))` .

Example

```
? (windows)
(#<LISTENER "Listener" #x49EB31>
 #<APROPOS-DIALOG "Apropos" #x532EF1>
 #<FRED-WINDOW "New" #x51CC61>)
? (target)
 #<APROPOS-DIALOG "Apropos" #x532EF1>
```

map-windows [*Function*]

Syntax `map-windows` *function* `&key :class :include-invisibles :include-windoids`

Description The `map-windows` function calls *function*, a function of one argument, on each window that satisfies the keywords.

Arguments

function A function of one argument.

`:class` A class used to filter output. The function *function* is called only on windows that match the value of `:class`. The default is `window`, which includes all windows.

`:include-invisibles` If the value of this variable is true, *function* is applied to both visible and invisible windows that are instances of `:class`. If the value is false, *function* is applied only to visible windows.

`:include-windoids` If the value of this variable is true, *function* is applied to floating windows. If the value is false, it is not.

Example

The following code provides a simple way to implement `front-window` using `map-windows`:

```
? (defun simple-front-window
          ()
   (let ((f #'(lambda (w)
                (return-from simple-front-window w))))
   (declare (dynamic-extent f))
   (map-windows f)))
SIMPLE-FRONT-WINDOW
```

find-window

[*Function*]

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

Description The `find-window` function returns the frontmost window of the class *class* for which a prefix of the window's title is `string-equal` to *title* . If no window has *title* as its title, `nil` is returned. (The cross that appears in the title bar of modified Fred windows is ignored when comparing the title.)

Arguments

title	A string specifying the title of the window to search for.
class	A class used to filter the result. The frontmost window that inherits from *class* is returned. The default is `window` .

Example

```
? (find-window "Listener")
#<LISTENER "Listener" #x5204C9>
? (find-window 'listener)
#<LISTENER "Listener" #x5204C9>
? (find-window "lis")
#<LISTENER "Listener" #x5204C9>
? (find-window "ist")
NIL
```

window-close

[*Generic function*]

Syntax `window-close` (*window* `window`)

Description The `window-close` generic function closes the window. The associated Macintosh data structures will be deallocated the next time the garbage collector runs. This operation is the inverse of `initialize-instance` . When a window is closed, its state is lost and cannot be recovered.

The MCL event system calls `window-close` when the user clicks a window's close box or chooses Close from the File menu.

Argument

window	A window.

Example

You can tell if a window has been closed by determining whether `wptr` called on the window returns `nil`.

```
? (setq baz (make-instance 'window
                           :window-title "bazwin"))
#<WINDOW "bazwin" #x6143D1>
? (window-title baz)
"Bazwin"
? (wptr baz)
#<A Mac Zone Pointer Size 156 #x715930>
? (window-close baz)
NIL
? (window-title baz)
"<No title>"             ;the window's state is lost
? (wptr baz)
NIL
```

view-position

[*Generic function*]

Syntax `view-position` (*window* `window`)

Description The `view-position` generic function returns the position of the upper-left corner of the window as a point.

Argument *window* A window.

set-view-position

[*Generic function*]

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

Description The `set-view-position` generic function moves the window and returns the new position of the upper-left corner, expressed as a point.

For windows with title bars, such as document windows and tool windows, the position is not the upper-left corner of the title bar but the upper-left corner of the content area of the window.

Arguments

window A window.

h The horizontal coordinate of the new position, or the complete position.

This may also be a keyword or list specifying how to center the window.

To center a window, specify the new position as the keyword `:centered`. If the position is `:centered`, the window will be centered vertically and horizontally.

The position may also be a list of the form (*reference offset*), where *reference* is one of the keywords `:top`, `:left`, `:bottom`, or `:right`, and *offset* is a number.

- If *reference* is `:top`, the top of the window is offset *offset* number of pixels from the top of the screen, and the window is centered horizontally.
- If *reference* is `:bottom`, the bottom of the window is offset *offset* number of pixels from the bottom of the screen, and the window is centered horizontally.
- If *reference* is `:left`, the left side of the window is offset *offset* number of pixels from the left of the screen, and the window is centered vertically.
- If *reference* is `:right`, the right side of the window is offset *offset* number of pixels from the right of the screen, and the window is centered vertically.

v The vertical coordinate of the new position, or `nil` if the complete position is given by *h*.

Examples

```
? (setq bim (make-instance 'window
                  :view-position #@(50 50)))
#<WINDOW "Untitled" #x506829>
? (point-string (view-position bim))
"#@(50 50)"
? (set-view-position bim #@(100 100))
6553700
? (point-string (view-position bim))
"#@(100 100)"
```

Here is an example of the use of `:centered`.

```
? (setq bim (make-instance 'window
                  :view-position :centered))
#<WINDOW "Untitled" #x509F59>
```

view-size

[*Generic function*]

Syntax `view-size` (*window* `window`)

Description The `view-size` generic function returns returns the size of the window as a point.

Argument *window* A window.

set-view-size

[*Generic function*]

Syntax `set-view-size` (*window* `window`) *h* `&optional` *v*

Description The `set-view-size` generic function sets the size of the window.

The upper-left corner of the window is anchored, and the lower right corner moves according to the new size. If both *h* and *v* are given, they should be the new horizontal and vertical dimensions of the window. If the value of *v* is `nil` or not supplied, *h* is taken to be an encoded point holding both dimensions.

The new size is returned, expressed as a point.

Arguments

window	A window.
h	The new width of the window, or both the width and height (encoded as an integer point) if the value of *v* is `nil`.
v	The new height of the window, or `nil` if the height and width are both given by *h*.

window-size-parts [*Generic function*]

Syntax `window-size-parts` (*window* `window`)
`window-size-parts :before` (*window* `window`)

Description The `window-size-parts` generic function can be specialized to resize the subviews of a window whenever the size of the window is changed. This function is called directly or indirectly by the methods specialized on *window* for the generic functions `initialize-instance` , `set-view-size` , `window-zoom-event-handler` , and `window-grow-event-handler` .

The primary method for `window` does nothing. The `:before` method for `window` ensures that the `view-clip-region` and `view-origin` of each of the window's subviews are recomputed the next time they are needed. The method for `fred-window` resizes the horizontal and vertical scroll bars as well as the main text area of the window.

Argument *window* A window or Fred window.

window-default-position [*Variable*]

Description The default position of a newly opened window. The initial value is `#@(6 44)` .

window-default-size [*Variable*]

Description The default size of a newly opened window. The initial value is `#@(502 150)` .

view-default-position [*Generic function*]

Syntax `view-default-position` (*window* `window`)

Description When a window is created, the `view-default-position` generic function is called if no position is explicitly specified either as the `:view-position` initialization argument to `make-instance` or as a default initialization argument in the class definition. The value returned is used as the initial position of the window. It must be a valid position specifier, either a point or a centering specifier as documented under `set-view-position` . The system-supplied method specialized on `window` returns the value of `*window-default-position*` .

Argument *window* A window.

view-default-size [*Generic function*]

Syntax `view-default-size` (*window* `window`)

Description When a window is created, the `view-default-size` generic function is called if no size is explicitly specified either as the `:view-size` initialization argument to `make-instance` or as a default initialization argument in the class definition. The value returned is used as the initial size of the window. It must be a point. The system-supplied method specialized on `window` returns the value of `*window-default-size*` .

Argument *window* A window.

window-title [*Generic function*]

Syntax `window-title` (*window* `window`)
`window-title` (*window* `fred-window`)

Description The `window-title` generic function returns the window title as a string. It ignores the crosses in the title bars of modified Fred windows.

Argument *window* A window.

set-window-title

[*Generic function*]

Syntax

`set-window-title` (*window* `window`) *new-title*
`set-window-title` (*window* `fred-window`) *new-title*

Description

The `set-window-title` generic function sets the window title to *new-title* . It ignores the crosses in the title bars of modified Fred windows.

Arguments

window A window.

new-title A string to be used as the new title.

view-font

[*Generic function*]

Syntax

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

Description

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

For the Listener, `view-font` changes `:bold` to `:plain` in the result of `call-next-method` .

For Fred windows, `view-font` returns three values: the current font for newly inserted characters; the font of the first character after the insertion point, or of the first character in the selection if there is a selection; and a Boolean value specifying whether all the selected text is in the same font as the current font.

Argument

window A window, Fred window, or Listener window.

view-default-font

[*Generic function*]

Syntax

```
view-default-font (window window)
view-default-font (view simple-view)
view-default-font (window listener)
```

Description

If a `:view-font` initialization argument is not specified when a view is created, the generic function `view-default-font` is called to determine its font.

The `window` method on `view-default-font` returns the value of `*fred-default-font-spec*`. The `listener` method returns the value of `*listener-default-font-spec*`. The initial value of both these variables is `("Monaco" 9 :PLAIN)`. The `simple-view` method returns `nil`, meaning that the view inherits its font from its container.

Every window has a font spec associated with it, even if the window never uses fonts.

Arguments

window	A window.
view	A simple view.

set-view-font

[*Generic function*]

Syntax

```
set-view-font (window window) font-spec
set-view-font (window fred-window) font-spec
set-view-font (window listener) font-spec
```

Description

The `set-view-font` generic function sets the font spec of *window* to *font-spec*.

Arguments

window	A window.
font-spec	A font specifier. If *font-spec* doesn't specify all four components of a font spec, the missing components are taken from the window's current font. (See Chapter 3, "Points and Fonts," for a complete description of font specs.)

Example

Here is an example of setting a window font.

```
? (setf freddy (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4A20A1>
?(view-font freddy)
("Monaco" 9 :SRCOR :PLAIN)
NIL
NIL
? (set-view-font freddy '(:bold 14))
(:BOLD 14)
? (view-font freddy)
("Monaco" 14 :SRCOR :BOLD)
NIL
NIL
```

For another example of the use of `set-view-font`, see the file `font-menus.lisp` in your MCL Examples folder.

---

view-font-codes

[*Generic function*]

Syntax

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

Description

The `view-font-codes` generic function returns two values, the font face code and mode-size code for *view*'s font. (Font codes are a more efficient way of encoding font specs; they are described in *Inside Macintosh*.)

Arguments

view	A simple view.
window	A window.

Example

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

set-view-font-codes

[*Generic function*]

Syntax

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

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

Description

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

For full details of font codes, see *Inside Macintosh*.

Arguments

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

Example

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

part-color

[*Generic function*]

Syntax part-color (*window* window) *part*

Description The part-color generic function returns the color of the part of the window indicated by *part*.

Arguments

window A window.

part A keyword specifying which part of the window should be set. The five possible keywords have the following meanings:

- :content — The frames of :double-edge-box windows; unused in other windows.
- :frame — The outline of the window and the title bar of :tool windows.
- :text — The title of :document windows.
- :hilite — The lines in the title bar of :document windows.
- :title-bar — The background of the title bar in :document windows and the title in :tool windows.

set-part-color

[*Generic function*]

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

Description The `set-part-color` generic function sets the part of the window indicated by *part* to *color* and returns *color*, encoded as an integer. If *color* is `nil`, the default color is restored.

Arguments

window	A window.
part	A keyword specifying a part of the window whose color should be returned. The same are allowed as for `part-color`.
color	A color, either symbolic or encoded as an integer.

Example

```
? (setf fred (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4B4C79>
? (set-part-color fred :content *red-color*)
14485510
? (set-part-color fred :frame 2078484)
2078484
```

part-color-list

[*Generic function*]

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

Description The `part-color-list` generic function returns a property list of keywords and colors for all the colored components of the window. The same keywords apply as for `part-color`. Components whose color has not been set are not included.

Argument

window	A window.

Example

```
? (part-color-list fred)
(:FRAME 2078484 :CONTENT 14485510)
```

window-show [*Generic function*]

Syntax window-show (*window* window)

Description The window-show generic function makes a window visible on the screen (assuming the window is not at an off-screen position).

Argument *window* A window.

window-hide [*Generic function*]

Syntax window-hide (*window* window)

Description The window-hide generic function makes a window invisible on the screen.

Argument *window* A window.

window-shown-p [*Generic function*]

Syntax window-shown-p (*window* window)

Description The window-shown-p generic function returns true if the window is visible, and false if it is hidden.

Argument *window* A window.

window-ensure-on-screen [*Generic function*]

Syntax window-ensure-on-screen (*window* window) &optional *default-position default-size*

Description The generic functionwindow-ensure-on-screen ensures that the window is entirely visible on one or more of the Macintosh screens. It may overlap two screens, but if it is not entirely visible, as determined bywindow-on-screen-p , it is moved to the position *default-position* . If it is still not entirely visible, its size is changed to *default-size.*

This function is useful when window positions are saved and restored on Macintosh computers with different screen configurations.

If you hold down the shift key while selecting a window from the Windows menu, `window-ensure-on-screen` is called on it.

Arguments

window A window.

default-position The position to which the window is moved if it needs to be. The default *default-position* is the value of `*window-default-position*` .

default-size The default size of the window. The default *default-size* is the value of `*window-default-size*` .

window-on-screen-p

[*Generic function*]

Syntax `window-on-screen-p` (*window* `window`)

Description The `window-on-screen-p` generic function returns `t` if all of *window* is on the screen, `nil` otherwise.

Argument

window A window.

window-active-p

[*Generic function*]

Syntax `window-active-p` (*window* `window`)

Description The `window-active-p` generic function returns `t` if *window* is the active window, `nil` otherwise.

Except when Macintosh Common Lisp is not the active application, it returns `t` for all floating windows and for the frontmost non-floating visible window.

Argument

window A window.

window-layer

[*Generic function*]

Syntax `window-layer` (*window* `window`) `&optional` *include-invisibles*

Description The `window-layer` generic function returns the number of windows in front of *window* . Floating windows are counted.

Arguments

window A window.

include-invisibles A Boolean value specifying whether or not to include invisible windows in the count. The default value is `nil`, indicating that `window-layer` counts only visible windows.

set-window-layer

[*Generic function*]

Syntax `set-window-layer` (*window* `window`) *new-layer* `&optional` *include-invisibles*
`set-window-layer` (*window* `windoid`) *new-layer* `&optional` *include-invisibles*

Description The `set-window-layer` generic function changes the layer of the window to *new-layer* . Floating windows are counted.

To make a window the frontmost window that is not a floating window, set its layer to `*windoid-count*` .

You can use `set-window-layer` to move a regular window in front of a floating window. Once other events occur, however, the floating window moves back to the front.

Arguments

window A window.

windoid A floating window.

new-layer A nonnegative integer indicating how many windows should be in front of *window* . If *new-layer* is equal to or greater than the number of windows on screen, *window* is moved all the way to the back. If the value of *new-layer* is `0`, *window* is moved to the front.

include-invisibles

A variable specifying whether the layering should take invisible windows into account. If the value of *include-invisibles* is false (the default), invisible windows are ignored. If it is true, invisible windows are counted.

window-select

[*Generic function*]

Syntax

```
window-select (window window)
window-select (window windoid)
window-select (window null)
```

Description The `window-select` generic function brings a window to the front, activates it, and shows it if it is hidden. The previously active window is deactivated.

Argument *window* A window.

Advanced window features

The following operations are useful for advanced programmers working with windows.

window-zoom-position

[*Generic function*]

Syntax

```
window-zoom-position (window window)
```

Description The `window-zoom-position` generic function returns the zoom position of a window, that is, its position after the user clicks the zoom box. This value is either the last value given to `set-window-zoom-position` for *window* or the value returned by calling `window-default-zoom-position` on *window*.

Argument *window* A window.

window-default-zoom-position [*Generic function*]

Syntax `window-default-zoom-position (`*window* `window)`

Description The `window-default-zoom-position` generic function determines the default zoom position of a window, that is, its new position after the user clicks the zoom box.

Argument *window* A window.

Example

See the example under the definition of `set-window-zoom-position`.

window-default-zoom-position [*Variable*]

Description The `*window-default-zoom-position*` variable stores the default zoom position of a window, that is, its new position after the user clicks the zoom box.

This variable and `*window-default-zoom-size*` are initialized at startup to make a zoomed window fill the screen containing the menu bar with a 3-pixel border all around.

set-window-zoom-position [*Generic function*]

Syntax `set-window-zoom-position (`*window* `window)` *h* `&optional` *v*

Description The `set-window-zoom-position` generic function sets the zoom position of a window, that is, its new position after the user clicks the zoom box, and returns the new position, encoded as an integer.

Arguments

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

Example

Here is an example of setting the zoom position of a class of windows and of an instance.

```
? (defclass my-window-class (window) ())
#<STANDARD-CLASS MY-WINDOW-CLASS>
? (defmethod window-default-zoom-position
      ((w my-window-class))
           #@(10 50))
#<STANDARD-METHOD WINDOW-DEFAULT-ZOOM-POSITION (MY-WINDOW-CLASS)>
? (defvar *w* (make-instance 'my-window-class))
*W*
? (set-window-zoom-position *w* #@(20 60))
3932180
```

window-zoom-size

[*Generic function*]

Syntax `window-zoom-size` (*window* `window`)

Description The `window-zoom-size` generic function returns the zoom size of a window, that is, its size after the user clicks the zoom box. This value is either the last value given to `set-window-zoom-size` for *window* or the value returned by calling`window-default-zoom-size` on*window*.

Argument *window* A window.

window-default-zoom-size

[*Generic function*]

Syntax `window-default-zoom-size` (*window* `window`)

Description The generic function`window-default-zoom-size` determines the default zoom size of a window, that is, its new size after the user clicks the zoom box. The provided method returns the value of `*window-default-zoom-size*` .

Argument *window* A window.

window-default-zoom-size [*Variable*]

Description The *window-default-zoom-size* variable stores the default zoom size of a window, that is, its new size after the user clicks the zoom box.

This variable and *window-default-zoom-position* are initialized at startup to make a zoomed window fill the screen containing the menu bar with a 3-pixel border all around.

set-window-zoom-size [*Generic function*]

Syntax set-window-zoom-size (*window* window) *h* &optional *v*

Description The generic function set-window-zoom-size sets the zoom size of a window, that is, its new size after the user clicks the zoom box, and returns the new size, encoded as an integer.

Arguments

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

window-grow-rect [*Generic function*]

Syntax window-grow-rect (*window* window)

Description The window-grow-rect generic function returns a rectangle record whose upper-left and lower-right components determine the minimum and maximum sizes to which *window* can be resized with the mouse.

The window can still assume other sizes when the user clicks the zoom box, and other sizes can be set with the function set-view-size.

Argument

window	A window.

Example

```
(let ((r (window-grow-rect (target))))
  (format t "(~S, ~S)  (~S, ~S)"
          (pref r :rect.top)
          (pref r :rect.left)
          (pref r :rect.bottom)
          (pref r :rect.right)))
```

window-drag-rect

[*Generic function*]

Syntax `window-drag-rect` (*window* `window`)

Description The `window-drag-rect` generic function returns a rectangle record whose value constrains how *window* can be moved with the mouse. Whenever the pointer moves outside this rectangle, the gray window outline disappears, indicating that the user is out of bounds.

Argument *window* A window.

view-cursor

[*Generic function*]

Syntax `view-cursor` (*view* `simple-view`) *point*
`view-cursor` (*window* `window`) *point*

Description The `view-cursor` generic function returns the cursor shape to display when the mouse is at *point* , a point in *view* . It is called by `window-update-cursor` as part of the default `window-null-event-handler` .

Specialize the `view-cursor` generic function to change your view's cursor to one of the following predefined cursors or to a user-defined cursor.

`*arrow-cursor*`
The standard north-northwest arrow cursor.

`*i-beam-cursor*`
The I-beam shape used when the cursor is over an area of editable text.

`*watch-cursor*`
The watch-face shape shown during time-consuming operations, when event processing is disabled.

Arguments

view	A simple view.
window	A window.
point	The position of the cursor, expressed as a point.

window-cursor

[*Generic function*]

Syntax `window-cursor` (*window* `window`)

Description The `window-cursor` generic function returns the current cursor of *window* . The system-supplied `view-cursor` method for `window` calls `window-cursor` to determine the cursor of *window*.

Argument

window	A window.

window-object

[*Function*]

Syntax `window-object` *wptr*

Description The `window-object` function returns the window object pointed at by *wptr*.

Argument

wptr	A macptr to a window record..

Example

```
? (window-object (wptr (target)))
#<FRED-WINDOW "New" #x454909>
```

with-port

[*Macro*]

Syntax `with-port` *grafport* {*form* }*

Description The `with-port` macro executes *form* with *grafport* as the current GrafPort. Upon exit, the previously current GrafPort is restored. The *form* is executed within the special form `without-interrupts` .

This macro is a very low-level way of binding the QuickDraw GrafPort. It is not recommended for general use; instead use `with-focused-view` .

Arguments

grafport	A GrafPort, usually the `wptr` of a window.
form	Zero or more Lisp forms to be executed with the GrafPort set to *grafport* . These usually include direct calls to QuickDraw routines.

Supporting standard menu items

Many of the menu items in the default MCL menu bar operate on the top window. These menu items are instances of the class `window-menu-item` . (See Chapter 4, "Menus.") These commands can work in any window if the class of the window has an appropriate method.

The menu items and their corresponding functions are as follows:

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

If the class of the active window has a method definition for one of these functions, then the corresponding menu item is enabled. If the user chooses the menu item, the function is called on the active window. Enabling of items on the Edit menu is controlled by the generic function `window-can-do-operation` , described later in this section.

window-needs-saving-p [*Generic function*]

Syntax `window-needs-saving-p` (*window* `window`)

Description The `window-needs-saving-p` generic function determines whether the Save menu item in the File menu should be enabled for windows that have a definition of `window-save`.

The Save menu item is enabled if the class of the active window has a method definition for `window-save`, unless the window has a method definition for `window-needs-saving-p` and a call to `window-needs-saving-p` returns `nil`. If the window has a method definition for `window-needs-saving-p`, then Save is enabled only if a call to `window-needs-saving-p` returns true.

Argument *window* A window.

window-can-do-operation [*Generic function*]

Syntax
`window-can-do-operation` (*view* `fred-mixin`) *operation*
`&optional` *menu-item*
`window-can-do-operation` (*window* `window`) *operation*
`&optional` *menu-item*

Description The `window-can-do-operation` generic function returns a Boolean value indicating whether *view* can perform *operation*. (This is a more general replacement for the older MCL function `window-can-undo-p`, which could check only for Undo.) If the value returned is true, the menu item for *operation* is enabled; otherwise, it is disabled.

The `window-can-do-operation` method for `window` returns `t` if there is a method for *operation* defined for the class of *window* that is more specific than the built-in method defined for the class `window`. Otherwise `window-can-do-operation` returns the result of calling `window-can-do-operation` on the current key handler of *window*, if there is one. If not, it returns `nil`.

The method for `fred-mixin` returns `t` if the operation is meaningful for the current state of the Fred window or Fred dialog item.

Arguments	*view*	A Fred window or Fred dialog item.
	window	A window.
	operation	A symbol specifying one of the standard editing operations: `cut`, `clear`, `copy`, `paste`, `select-all`, `undo`, or `undo-more`.
	menu-item	The corresponding Edit menu item.

Example

```
? (window-can-do-operation *top-listener* 'paste)
T
```

Floating windows

Floating windows are a subclass of windows that appear frontmost on the screen. (That is, they always "float to the top.") Floating windows are generally used for creating tool palettes.

Floating windows respond to clicks, handle `activate` and `deactivate` events, and respond to keystroke events. See the file `windoid-key-events.lisp` in the MCL Examples folder for commented sample code.

These expressions are used in defining and counting floating windows.

`windoid` [*Class name*]

Description The class `windoid` is the class of floating windows, built on `window`. Floating windows may be mixed in with other window classes, such as dialog boxes. In this case, `windoid` should appear first in the inheritance path.

initialize-instance [*Generic function*]

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

Description The `initialize-instance` primary method for `windoid` initializes a floating window.

Arguments

windoid A floating window.

initargs A list of keywords and values used to initialize the floating window. No special keywords are used. The following keywords have default values:

`:view-size`
The default value of the size of the floating window is `#@(115 150)`.

`:window-do-first-click`
The value of this initialization argument determines whether the click that selects a window is also passed to `view-click-event-handler`. For all floating windows, the default value of this variable is true.

The click that selects an application in Multifinder is not passed to the application unless either the clicked window is not the front window or the Get Front Clicks bit is set in the application's size resource.

windoid-count [*Variable*]

Description The `*windoid-count*` variable contains the number of visible floating windows currently in the MCL environment.

Chapter 6 **Dialog Items and Dialogs**

Contents

The dialog functionality in Macintosh Common Lisp / 187
- Dialog items / 187
- Dialog boxes / 187
- A simple way to design dialogs and program dialog items / 188
- Changes to dialogs in Macintosh Common Lisp version 2 / 188

Dialog items / 190

MCL forms relating to dialog items / 190

Advanced dialog item functions / 201

Specialized dialog items / 206
- Buttons / 206
- Default buttons / 207
- Static text / 209
- Editable text / 210
- Checkboxes / 216
- Radio buttons / 218
- Table dialog items / 221
- Pop-up menu dialog items / 234
- Scroll-bar dialog items / 236
- Sequence dialog items / 243

User-defined dialog items / 245

Dialogs / 246
- Modal dialogs / 247
- Modeless dialogs / 248

Simple turnkey dialog boxes / 248

MCL forms relating to dialogs / 256

This chapter describes the dialog functionality and the built-in dialog item classes in Macintosh Common Lisp.

The dialog functionality is very flexible in Macintosh Common Lisp. Dialog items display information and may initiate an action when clicked by the user. In Macintosh Common Lisp, dialog items can appear in any window. They are built from the class `dialog-item`, which is not used directly; you specialize it and use the subclasses. In turn, `dialog-item` is built from the class `simple-view`, since dialog items have no subviews. Built-in subclasses of `dialog-item` include radio buttons, checkboxes, and editable-text fields, as well as pop-up menus, scroll bars, and tables in dialog boxes.

You should read this chapter if you are programming specialized types of dialog items.

Before reading this chapter, you should be familiar with the MCL implementation of views and windows, described in Chapter 5, "Views and Windows." The subclass of `dialog-item` that supports editable text is `fred-dialog-item`, documented in Chapter 14, "Programming the Editor."

The dialog functionality in Macintosh Common Lisp

In the standard Macintosh interface, actions are performed by dialog items within dialog boxes. Macintosh Common Lisp supports this functionality and makes it more generalized.

Dialog items

Instead of setting up a specialized class for dialog boxes and alerts, Macintosh Common Lisp defines any structured communication as simply a collection of dialog items in a window. You can add dialog items to any view or window, or you can write specialized classes based on `window`, in which dialog items may appear.

Therefore, for creating dialog functionality the important class is `dialog-item`.

Built-in MCL dialog items include buttons, radio buttons, checkboxes, tables, editable text, scroll bars, pop-up menus, and static text. They are discussed in the "Dialog Items" section later in this chapter.

In addition, the sample files in the Examples and Library folders contain code for kinds of dialog items. Of course, you can also define your own classes of dialog items.

Dialog boxes

Dialog boxes initiate and control well-defined actions in a structured way. You use them whenever you want the user to do something complex in which the range of response is predictable or needs to be controlled. The Print Options dialog box is a good example; it includes text fields, which the user fills in, and a defined set of choices that are represented by radio buttons and checkboxes.

Alerts query an action, displaying a message such as "Are you sure you want to reformat your hard disk?" They request the user to confirm explicitly before proceeding, or to cancel.

MCL dialogs are unspecialized subclasses of `window`, provided for backward compatibility with earlier versions of Macintosh Common Lisp. They have methods for all `window` and `view` operations, but no methods of their own. Display, for instance, works the same way in dialogs as in all other windows. Dialogs are only one of the places you can use dialog items.

Macintosh Common Lisp provides four predefined standard dialog boxes for alerts and user responses, discussed in the section “Simple Turnkey Dialog Boxes” later in this chapter.

You can write standard Macintosh dialog boxes quite easily, while the same functionality also adapts well to other uses. For example, you can create a hypertext system that includes text, graphics, and dialog items, or an interactive forms manager, or a spreadsheet, all using largely the same code.

A simple way to design dialogs and program dialog items

Macintosh Common Lisp version 2 contains a dialog design tool, part of the Interface Toolkit, that works like a simple paint system. You can choose buttons and fields from a palette and move them into a new dialog. You can set their default states and actions. This tool is supplied as source code so it can be customized; you’ll find it in the Interface Tools folder supplied with your copy of Macintosh Common Lisp. Its operation is described in Chapter 8, “The Interface Toolkit,” and in Chapter 3 of *Getting Started With Macintosh Common Lisp* .

Changes to dialogs in Macintosh Common Lisp version 2

If you have used an earlier version of Macintosh Common Lisp, you will find that the implementation of dialogs has changed substantially, making them more flexible to use and easier to program.

- The `dialog` class, which is a subclass of `window`, exists only for compatibility. No methods are specialized on it and it adds no slots.
- Dialog items may now be added to all views.
- Some functions have changed to reflect the new definition of `dialog`.
- All new functions are CLOS generic functions.

The file `old-dialog-hooks.lisp` , distributed in the Examples folder that is part of Macintosh Common Lisp, contains code defining the old dialog functions in terms of the new ones. You should find it quite easy, however, to port your old dialog code to Macintosh Common Lisp version 2.

Table 6-1 summarizes the functions that have changed.

n **Table 6-1** Summary of changed dialog functions in Macintosh Common Lisp version 2

Old	New
add-dialog-items	add-subviews
add-self-to-dialog	install-view-in-window
buffer-char-font	buffer-char-font-spec
buffer-replace-font	buffer-replace-font-spec
buffer-set-font	buffer-set-font-spec
catch-abort	use restart-case
catch-error	use handler-case
catch-error-quietly	ignore-errors
color-window-mixin	:color-p initialization argument
:dialog-item-colors	:part-color-list intialization argument
dialog-item-default-size	view-default-size
dialog-item-dialog	view-container
(set-)dialog-item-font	(set-)view-font
dialog-item-nick-name	view-nick-name
(set-)dialog-item-size	(set-)view-size
(set-)dialog-item-position	(set-)view-position
ed-skip-fwd-wsp&comments	buffer-skip-fwd-wsp&comments
find-named-dialog-items	view-named, find-named-sibling
item-named	view-named
markp	buffer-mark-p
:parent keyword to windows, etc.	:class keyword
remove-dialog-items	remove-subviews
remove-self-from-dialog	remove-view-from-window
window-(de)activate-event-handler	view-(de)activate-event-handler
window-buffer	fred-buffer
window-click-event-handler	view-click-event-handler
window-font	view-font
window-hpos	fred-hpos
window-line-vpos	fred-line-vpos
window-mouse-position	view-mouse-position
(set-)window-position	(set-)view-position
(set-)window-size	(set-)view-size
window-start-mark	fred-display-start-mark
window-update	fred-update
window-vpos	fred-vpos

Dialog items

Dialog items do two things: appear within a view, and perform actions. Generally speaking, a dialog item inherits its display behavior from `simple-view` or from its class; its default methods are also determined at the class level. You can add specific action at the instance level.

The base class from which all other dialog items inherit is `dialog–item` . This class is not meant to be instantiated directly. Instead, it is the superclass from which more specific classes of dialog items are built.

The dialog item subclasses provided by Macintosh Common Lisp are

`button-dialog-item`

`check-box-dialog-item`

`editable-text-dialog-item`

`fred-dialog-item`

`radio-button-dialog-item`

`sequence-dialog-item` (a subclass of `table-dialog-item`)

`static–text-dialog–item`

`table-dialog-item`

The class `fred-dialog-item` is discussed in Chapter 14, “Programming the Editor.” The others are discussed in this chapter.

In addition, you can use sample files in your MCL Examples and Library folders to make several other kinds of dialog items, including scroll bars, icons, and pop-up menus, and of course you can create your own subclasses of `dialog-item` .

MCL forms relating to dialog items

The following MCL expressions are used in creating dialog items.

dialog-item [*Class name*]

Description The class `dialog-item` provides the basic functionality shared by all dialog items. It is built on `simple-view` .

initialize-instance [*Generic function*]

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

Description The `initialize-instance` primary method for `dialog-item` initializes a dialog item. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments

dialog-item A dialog item.

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

`:view-size`
The size of the dialog item. If not specified or `nil`, this value is calculated so that the item's `dialog-item-text` is visible. If the specified value is too small, the item is clipped when it is drawn. The default value is `nil`.

`:view-container`
The dialog box or other view that contains the item.

`:view-position`
The position in the dialog box where the item will be placed, in the coordinate system of its container. If this argument is not specified or is specified as `nil`, the first available position large enough to hold the item is used. If no space is large enough, the dialog item is placed in the upper-left corner of the dialog.

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

`:view-font`
: The font in which the text of the dialog item appears. If `nil`, the window font is used.

`:dialog-item-text`
: The text of the dialog item. The initial value is `nil`.

`:dialog-item-handle`
: For advanced programmers, this option specifies the handle of the dialog item. See the description of the `dialog-item-handle` generic function in the next section of this chapter, "Advanced Dialog Item Functions." This option is used only for creating specialized subclasses of dialog items. The handle is usually allocated by the `install-view-in-window` method. Its initial value is `nil`.

`:dialog-item-enabled-p`
: The state (enabled or disabled) of the item. Disabled items are dimmed, and their actions are not run when the user clicks them.

`:part-color-list`
: A property list of colors to which the parts of the dialog item should be set. The four possible keywords are: `:frame`, the outline of the dialog item; `:text`, its text; `:body`, its body; and `:thumb`, its scroll box. (The scroll box is the white box that slides inside the scroll bar; scroll bars are the only dialog items that can have them.)

`:dialog-item-action`
: The action run when the dialog item is selected. The value of this keyword should be a function or a symbol with a global function definition. It is called with a single parameter, the dialog item.

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

`:wptr`

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

dialog-items

[*Generic function*]

Syntax `dialog-items` (*view* `view`) `&optional` *item-class must-be-enabled*

Description The `dialog-items` generic function returns a list of the dialog items in *view* .

Arguments

view A view.

item-class If the value of *item-class* is specified and non-`nil`, then only dialog items matching *item-class* are returned. The default value is `nil`.

must-be-enabled If the value of *must-be-enabled* is true, then only dialog items that are enabled are returned. The default value is `nil`.

make-dialog-item

[*Function*]

Syntax `make-dialog-item` *class position size text* `&optional` *action* `&rest` *attributes*

Description The `make-dialog-item` function creates a dialog item using `make-instance` .

Arguments

class The class of the dialog item.

position The position of the dialog item with respect to its container.

size The size of the dialog item.

text The text included within the dialog item.

action The action associated with the dialog item.

attributes One or more attributes belonging to the dialog item. The number and nature of these depend on the type of dialog item.

Example

This function could be defined as follows:

```
? (defun make-dialog-item (class position size text
                           &optional action &rest attributes)
       (apply #'make-instance class
              :view-position position
              :view-size size
              :dialog-item-text text
              :dialog-item-action action
              attributes))
```

dialog-item-action [*Generic function*]

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

Description The generic function `dialog-item-action` is called whenever the user clicks a dialog item. The method for `dialog-item` calls *item*'s `dialog-item-action-function` , if it is not `nil`. Otherwise, it does nothing.

The `dialog-item-action` function is normally called when the mouse button is released, not when it is pressed.

If an item is disabled, its action is not run.

Since `dialog-item-action` is usually called by `view-click-event-handler` as a result of event processing, event processing is ordinarily disabled while the `dialog-item-action` function is running. This means that other dialog items cannot be selected during the action. To avoid locking out other event processing, you can use `eval-enqueue` to insert forms into the read-eval-print loop. For details, see Chapter 11, "Events."

Argument *item* A dialog item.

dialog-item-action-function [*Generic function*]

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

Description The generic function `dialog-item-action-function` returns the value set by the `:dialog-item-action` initialization argument or the `set-dialog-item-action-function` generic function. Unless it is `nil`, this function is called with a single argument, *item*, by the `dialog-item-action` method for `dialog-item` .

This generic function is called by the `view-click-event-handler` method for `dialog-item` when the user clicks a dialog item.

Argument *item* A dialog item.

set-dialog-item-action-function [*Generic function*]

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

Description The generic function `dialog-item-action-function` sets the value it accesses.

Arguments *item* A dialog item.

new-function A function of one argument or a symbol that has a global function binding or is `nil`.

view-click-event-handler [*Generic function*]

Syntax `view-click-event-handler` (*item* `dialog-item`) *where*

Description The generic function `view-click-event-handler` is called by the event system when the user clicks the dialog item. The method for `dialog-item` calls *item*'s `dialog-item-action-function` with *item* as the single argument. If *item*'s `dialog-item-action-function` is `nil`, nothing is done.

Arguments *item* A dialog item.

where The cursor position. It is ignored.

view-focus-and-draw-contents [*Generic function*]

Syntax `view-focus-and-draw-contents` (*item* `dialog-item`)
`&optional` *visrgn cliprgn*

Description The method for dialog items of the generic function `view-focus-and-draw-contents` focuses on the container of the dialog item, then calls `view-draw-contents`.

Arguments

item	A dialog item.
visrgn, *cliprgn*	Region records from the view's `wptr`. They are ignored.

view-size [*Generic function*]

Syntax `view-size` (*item* `dialog-item`)

Description The method for dialog items of the generic function `view-size` returns the size of the dialog item as a point.

Argument

item	A dialog item.

set-view-size [*Generic function*]

Syntax `set-view-size` (*item* `dialog-item`) *h* `&optional` *v*

Description The method for dialog items of the generic function `set-view-size` changes the size of the dialog item to the width and height represented by *h* and *v*, and returns the new size.

Arguments

item	A dialog item.
h	Horizontal position.
v	Vertical position. If *v* is `nil`, *h* is assumed to represent a point.

Example

```
? (add-subviews my-window
    (setf eddie
      (make-instance 'editable-text-dialog-item)))
NIL
? (point-string (view-size eddie))
"#@(6 17)"
? (set-view-size eddie #@(300 20))
1311020
```

dialog-item-text

[*Generic function*]

Syntax `dialog-item-text` (*item* `dialog-item`)

Description The generic function `dialog-item-text` returns the string of text associated with the dialog item.

Argument *item* A dialog item.

set-dialog-item-text

[*Generic function*]

Syntax `set-dialog-item-text` (*item* `dialog-item`) *text*

Description The generic function `set-dialog-item-text` sets the text associated with the dialog item to *text* and returns *text*.

The text of a dialog item has a different meaning for each class of dialog item. It is the text of static-text and editable-dialog text items. It is the label displayed inside buttons and to the right of radio buttons and checkboxes.

If you prefer to put text in a different location, set the text to the empty string and use a separate static-text item to place the text where you would like it.

Tables do not display their dialog item text.

Arguments *item* A dialog item.

text A string to be used as the new text of the dialog item.

view-font

[*Generic function*]

Syntax `view-font` (*item* `dialog-item`)

Description The generic function `view-font` returns, as a font spec, the font used by *item* , or `nil` if *item* does not have its own font. (If *item* does not have its own font, it uses its container's font.)

Argument *item* A dialog item.

set-view-font

[*Generic function*]

Syntax `set-view-font` (*item* `dialog-item`) *new-font*

Description The generic function `set-view-font` sets the font of the dialog item to *new-font* .

Arguments *item* A dialog item.

new-font A font specifier. If `nil`, the dialog item uses the font of its window.

view-font-codes

[*Generic function*]

Syntax `view-font-codes` (*item* `dialog-item`)

Description The generic function `view-font-codes` returns two values, the font face code and mode-size code for *item* 's font. (Font codes, an efficient way of encoding font specs, are described in *Inside Macintosh* and in the section "Implementation of Font Codes" of Chapter 3, "Points and Fonts.")

Argument *item* A dialog item.

set-view-font-codes

[*Generic function*]

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

Description The generic function `set-view-font-codes` changes the view font codes of *item* .

Arguments

item	A dialog item.
ff	The font/face code. A font/face code is a 32-bit integer that stores the encoded name of the font and its face (plain, bold, italic, and so on).
ms	The mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size.
ff-mask	A mask that instructs `set-view-font-codes` to look only at certain bits of the font/face integer.
ms-mask	A mask that instructs `set-view-font-codes` to look only at certain bits of the mode/size integer.

part-color

[*Generic function*]

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

Description The generic function `part-color` returns the color of the part indicated by *part* .

Arguments

item	A dialog item.
part	A keyword specifying which part of the dialog item should be set. The four possible keywords are `:frame` , the outline of the dialog item; `:text` , its text; `:body` , its body; and `:thumb` , its scroll box. (The scroll box is the white box that slides inside the scroll bar; scroll bars are the only dialog items that can have them.)

set-part-color

[*Generic function*]

Syntax `set-part-color` (*item* `dialog-item`) *part new-color*

Description The generic function `set-part-color` sets the color of part of the dialog item, as specified by the arguments, and returns *new-color* .

If you create a new class of dialog items, you may want to define `view-draw-contents` to pay attention to these values.

In addition to the keywords specified by *part* , individual cells in table dialog items can have colors. See the method on `set-part-color` for `table-dialog-item` .

Arguments

item	A dialog item.
part	A keyword specifying a part of the dialog item. The same keywords are allowed as for `part-color` .
new-color	A color, encoded as an integer.

part-color-list

[*Generic function*]

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

Description The generic function `part-color-list` returns a list of part keywords and colors for all the colored components of the dialog item. Components whose color has not been set are not included.

Argument

item	A dialog item.

dialog-item-enable

[*Generic function*]

Syntax `dialog-item-enable` (*item* `dialog-item`)

Description The generic function `dialog-item-enable` enables the dialog item. The item is not dimmed, and its action is run when the user clicks it. The function returns `nil`.

Argument

item	A dialog item.

dialog-item-disable [*Generic function*]

Syntax `dialog-item-disable` (*item* `dialog-item`)

Description The generic function `dialog-item-disable` disables the dialog item. The dialog item is dimmed; clicks in the item are ignored, and the action of the item is never run. Disabling a checkbox does not alter its status as checked, and disabling a radio button does not alter its status as clicked (you may want to remove the check or click explicitly). The function returns `nil`.

Argument *item* A dialog item.

dialog-item-enabled-p [*Generic function*]

Syntax `dialog-item-enabled-p` (*item* `dialog-item`)

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

Argument *item* A dialog item.

Advanced dialog item functions

The following functions can be defined for user-created classes of dialog items. They can also be shadowed in specialized classes of the predefined dialog items. For general use of dialog items, you do not need to use the following functions.

Several sample files demonstrate the use of dialog items. In your MCL Examples folder, `text-edit-dialog-item.lisp` shows how to implement dialog items if you do not want to make Fred a part of your implementation. In the Library folder, `graphic-items.lisp`, `scroll-bar-dialog-items.lisp`, and `scrolling-fred-dialog-item.lisp` implement several specialized types of dialog items.

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

Syntax `install-view-in-window` (*item* `dialog-item`) *window*

Description The generic function `install-view-in-window` is called by `set-view-container` when an item becomes part of a view.

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

The default method sets the size of the dialog item if it does not already have one, and finds an empty position for the dialog item if it does not already have a position.

Arguments

item A dialog item.

window The window to which the dialog item is being added.

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

Syntax `remove-view-from-window` (*item* `dialog-item`)

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

Argument

item A dialog item.

dialog-item-handle

[*Generic function*]

Syntax `dialog-item-handle` (*item* `dialog-item`)

Description The generic function `dialog-item-handle` retrieves the handle associated with *item*. Dialog items are often associated with handles to Macintosh data structures, such as control records. By convention, this handle is stored in the location referenced by `dialog-item-handle` and modified by `set-dialog-item-handle`. The handle is usually `nil` when the dialog item is not contained in a window. It is generally set by `install-view-in-window` and is reset to `nil` by `remove-view-from-window`.

Argument *item* A dialog item.

set-dialog-item-handle

[*Generic function*]

Syntax `set-dialog-item-handle` (*item* `dialog-item`) *handle*

Description The generic function `set-dialog-item-handle` sets the dialog item handle associated with *item* to a new handle.

Arguments *item* A dialog item.
handle A handle to a Macintosh data structure.

view-activate-event-handler

[*Generic function*]

Syntax `view-activate-event-handler :around` (*item* `dialog-item`)

Description The generic function `view-activate-event-handler` is called when the window containing the dialog item is activated.

If the appearance of the dialog item needs to change to indicate that it is active, this is the method that should make that change. For example, Fred dialog items change their highlighting from a pixelwide box to a solid rectangle and scroll bars make their arrows and scroll box visible.

The `view-activate-event-handler` generic function is called by `set-view-container` if the window in which the newly installed view appears is active.

Argument *item* A dialog item.

`view-deactivate-event-handler` [*Generic function*]

Syntax `view-deactivate-event-handler` (*item* `dialog-item`)

Description The generic function `view-deactivate-event-handler` is called when the window containing the dialog items is deactivated.

If the appearance of the dialog item needs to change to indicate that it is not active, this is the method that should make that change. For example, Fred dialog items change their highlighting from a solid rectangle to a 1-pixel-wide box and scroll bars become an empty rectangle.

The `view-deactivate-event-handler` generic function is called by `set-view-container` if the window in which the newly installed view appears is not active.

Argument *item* A dialog item.

`view-default-size` [*Generic function*]

Syntax `view-default-size` (*item* `dialog-item`)

Description The generic function `view-default-size` is called by the default version of `install-view-in-window`. It is called for dialog items that are not given an explicit size. The `dialog-item` method of `view-default-size` calculates a size according to the font and text of the dialog item and the width correction associated with the class of the dialog item. (See the documentation of `dialog-item-width-correction`.)

Argument *item* A dialog item.

dialog-item-width-correction [*Generic function*]

Syntax `dialog-item-width-correction` (*item* `dialog-item`)

Description The generic function `dialog-item-width-correction` returns an integer representing the number of pixels of white space added to the left and right of the text of a dialog item. The default method for `dialog-item` returns `0`. Users can write methods for `dialog-item-width-correction` if they wish to specialize it for their own classes of dialog items.

Argument *item* A dialog item.

with-focused-dialog-item [*Macro*]

Syntax `with-focused-dialog-item` (*item* `&optional` *container*) `&body` *body*

Description The macro `with-focused-dialog-item` executes *body* with the drawing environment set up in the coordinate system of *container* and the font of *item*. This is the correct environment for calling `view-draw-contents` on a dialog item. When the body exits (normally or abnormally), the old drawing environment is restored.

Arguments

item	A dialog item (or any simple view).
container	The view focused on whose coordinate system *body* will run.
body	Forms to be executed with the specified drawing environment.

Examples

The macro `with-font-focused-view` could be defined as follows.

```
(defmacro with-font-focused-view (view &body body)
  (let ((view-sym (gensym)))
    `(let ((,view-sym ,view))
       (with-focused-dialog-item (view view) ,@body))))
```

The macro `view-focus-and-draw-contents` for `dialog-item` could be defined as follows.

```
(defmethod view-focus-and-draw-contents ((item dialog-item) &optional
                                         visrgn cliprgn)
  (declare (ignore visrgn cliprgn))
  (with-focused-dialog-item item
    (view-draw-contents item)))
```

Specialized dialog items

Button, static text, editable text, checkboxes, radio button, tables, sequences, and user-defined dialog items fall into the category of specialized dialog items.

The initialization argument keywords documented for the `dialog-item` class are applicable to all dialog items. Only the additional keywords that are specific to each specialized dialog item are documented in the following sections.

Buttons

Button dialog items are rounded rectangles that contain text. The following MCL expressions operate on button dialog items.

`button-dialog-item` [*Class name*]

Description This is the class used to make buttons. Clicking a button usually has an immediate result. Buttons are generally given a function for `dialog-item-action-function` via the `:dialog-item-action` initialization argument.

`initialize-instance` [*Generic function*]

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

Description The `initialize-instance` primary method for `button-dialog-item` initializes a button dialog item. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments *item* A button dialog item.

initargs A list of keywords and values used to initialize the button. These are its special *initargs* keywords (in addition to those for `dialog-item`):

`:default-button`
An argument specifying whether the button is made the default button. If this value is `nil` (the default), the button is not made the default button. Note that if the dialog has a default button and `:allow-returns` is true for the current key handler, then the Return key will be handled by the key handler rather than the default button.

`:border-p`
An argument specifying whether the button has a border. If this value is true (the default), the button has a border.

Example

```
? (setq pearl (make-instance 'button-dialog-item
                             :default-button t))
#<BUTTON-DIALOG-ITEM #x42C699>
```

press-button

[*Generic function*]

Syntax `press-button` (*button* `button-dialog-item`)

Description The `press-button` generic function highlights *button*, then calls the `dialog-item-action` method for *button*.

Argument *button* A button dialog item.

Default buttons

Default buttons are a convenient subclass of button dialog items; they serve as the default button. A dialog may have one default button. This button has a bold border and usually may be selected by one of the keystrokes Return or Enter.

The following MCL expressions operate on default-button dialog items.

default-button-dialog-item [*Class name*]

Description The `default-button-dialog-item` class is the class of default buttons, a subclass of `button-dialog-item` .

initialize-instance [*Generic function*]

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

Description The `initialize-instance` primary method for `default-button-dialog-item` initializes a default-button dialog item. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments

item A default-button dialog item.

initargs A list of keywords and values used to initialize the button. This class has no additional *initargs* keywords, but has two default values:

`:dialog-item-text`
The default value of this initialization argument is `"OK"`.

`:default-button`
The default value of this initialization argument is true.

default-button [*Generic function*]

Syntax `default-button` (*window* `window`)

Description The `default-button` generic function returns the current default button, or `nil` if the window has no default button. The default button is the button whose action is run when the user presses Return or Enter. It is outlined with a heavy black border.

If carriage returns are allowed in the current editable-text item, they are sent to that item rather than to the default button.

Argument *window* A window.

set-default-button [*Generic function*]

Syntax `set-default-button` (*window* `window`) *new-button*

Description The `set-default-button` generic function changes the default button according to the value of *new-button* and returns *new-button*.

If carriage returns are allowed in the current editable-text item, they are sent to that item rather than to the default button.

Arguments

window A window.

new-button The button that should be made the default button, or `nil`, indicating that there should be no default button.

default-button-p [*Generic function*]

Syntax `default-button-p` (*item* `button-dialog-item`)

Description The `default-button-p` generic function returns true if *item* is the default button in the `view-window` of *item*. Otherwise it returns `nil`.

Argument

item A button dialog item.

Static text

The next two entries define and initialize the class of static-text dialog items.

static-text-dialog-item [*Class name*]

Description This is the class of static-text dialog items. Static text may be positioned anywhere in a dialog window to supply additional information to the user. The text appears in the window's font unless otherwise specified. Clicking text does not generally initiate an action, but it may.

Depending on the amount of text and the size of the item, the text may wrap to fit in its area. If the size is not specified, a size that accommodates the text without wrapping is used.

initialize-instance [*Generic function*]

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

Description The `initialize-instance` primary method for `static-text-dialog-item` initializes a static-text dialog item. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments

item A static-text dialog item.

initargs A list of keywords and values used to initialize the static-text dialog item. The subclass `static-text-dialog-item` does not have any additional keyword arguments beyond those for `dialog-item` .

Editable text

The following entries pertain to the class of editable-text dialog items.

editable-text-dialog-item [*Class name*]

Description This is the class of editable-text dialog items, a subclass of `fred-dialog-item` . Its superclasses include `fred-mixin` and `key-handler-mixin` . The class adds no new initialization arguments, and there is only one method specialized on the class, `view-default-font` .

The user can give standard Macintosh commands to edit the text of such items. For instance, the user can select, cut, copy, and paste the text of editable-text dialog items.

Editable text is usually surrounded by a box, although this feature may be disabled.

At any given time, there is only one current editable-text dialog item. This is the item with a blinking cursor or a highlighted selection. User typing is directed to this item by a call to `view-key-event-handler`. Pressing the Tab key makes the next editable-text dialog item current, cycling back to the first after the last. The current editable-text dialog item can be determined by calling `current-key-handler` and can be changed by calling `set-current-key-handler`.

The text of an `editable-text-dialog-item` can be accessed by calling `dialog-item-text` and changed by calling `set-dialog-item-text`. When an editable text item is created, the initial text is specified using the `:dialog-item-text` initialization argument.

To refer unambiguously to an editable-text dialog item, you can give it a nickname.

Tip The file `text-edit-dialog-item.lisp`, in your Examples folder, provides an implementation of the class `editable-text-dialog-item` using the Macintosh TextEdit Manager. If your application does not require full Fred editing capability in editable text, you may wish to use `text-edit-dialog-item` instead of `editable-text-dialog-item`. Most of the built-in MCL dialogs containing editable text items instantiate these items as `editable-text-dialog-item` rather than as `fred-dialog-item`. If your application needs to use built-in dialogs but does not need Fred editing capability within those dialogs, you can redefine the class `editable-text-dialog-item` to be a subclass of `text-edit-dialog-item`.

`initialize-instance` [*Generic function*]

Syntax

```
initialize-instance (item editable-text-dialog-item)
  &rest initargs
```

Description The `initialize-instance` primary method for `editable-text-dialog-item` initializes an editable-text dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item An editable-text dialog item.

initargs A list of keywords and values used to initialize the editable-text dialog item. It has no new initialization arguments beyond those it inherits from `fred-dialog-item` .

`view-key-event-handler` [*Generic function*]

Syntax `view-key-event-handler` (*item* `fred-mixin`) *char*

Description The generic function `view-key-event-handler` examines the current keystroke and determines what is to be done with it.

The method for `fred-mixin` binds the `*current-keystroke*` variable to the keystroke of the current event and runs the Fred command associated with the keystroke.

Arguments

item An editable-text dialog item.

char Any keystroke. If *char* is a carriage return, this function is called only if `allow-returns-p` is true for the item.

`key-handler-mixin` [*Class name*]

Description The class `key-handler-mixin` should be mixed into any class that handles key events. The class `fred-dialog-item` includes `key-handler-mixin` .

`key-handler-p` [*Generic function*]

Syntax `key-handler-p` (*item* `dialog-item`)
`key-handler-p` (*key-handler* `key-handler-mixin`)

Description The `key-handler-p` generic function checks to see whether *item* is a key handler. When `key-handler-p` is called on an instance of a class one of whose superclasses is `key-handler-mixin` , the function returns `t` unless the key handler is disabled. The method for `dialog-item` returns `nil`.

Arguments

item A dialog item.

key-handler An object one of whose superclasses is `key-handler-mixin` .

exit-key-handler

[*Generic function*]

Syntax `exit-key-handler` (*item* `key-handler-mixin`) *new-text-item*

Description The generic function `exit-key-handler` is called when an editable-text dialog item that is the current key handler is about to be exited. At this point, it is still the current key handler, but soon it won't be. If the function returns `t` (as the method for `key-handler-mixin` does), *new-text-item* is made the new key handler. If it returns `nil`, *item* remains the `current-key-handler` .

Arguments

item	An editable-text dialog item.
new-text-item	The editable-text dialog item about to be made current.

enter-key-handler

[*Generic function*]

Syntax `enter-key-handler` (*item* `key-handler-mixin`) *old-text-item*

Description The generic function `enter-key-handler` is called when a key handler such as an editable-text dialog item has just been made current.

The method for `key-handler-mixin` doesn't do anything; it is a hook on which you can specialize behavior. For example, you can set another dialog item as the current key handler, as in the example.

Arguments

item	An editable-text dialog item.
old-text-item	The previously current editable-text item in the dialog. This is `nil` the first time an editable-text item is added to a dialog.

Example

Here is an example of entering and exiting fields by polling through the key handlers `enter-key-handler` and `exit-key-handler` . The dialog `foo` contains two editable-text dialog items, `Changing` and `Checking` . `Checking` is a simple instance of `editable-text-dialog-item` . `Changing` is an instance of a subclass, `changer-text-item` , which has methods for `enter-key-handler` and `exit-key-handler` . These methods do all the work.

If you edit the text of `Changing`, the `exit-key-handler` method for `changer-text-item` brings up a message when the next item is clicked. If you edit the text of `Checking`, the `enter-key-handler` method for `changer-text-item` returns `nil` and `Checking` remains the `current-key-handler` until the original text is restored.

This example is available as the file `check-and-change.lisp` in the Examples folder distributed as part of Macintosh Common Lisp.

```
;;Checking is a simple editable-text-dialog-item
(setq Checking (make-instance 'editable-text-dialog-item
                :dialog-item-text "Click here to check"
                :view-position #@(16 16)))

;; changer-text-item is a new subclass
(defclass changer-text-item (editable-text-dialog-item) ()
          (:default-initargs :dialog-item-text
                             "Change me and see what happens"))

;; changer-text-item has methods for enter-key-handler
;; and exit-key-handler
(defmethod exit-key-handler
           ((changer-text-item changer-text-item) next-item)
           (declare (ignore next-item))
           (unless (equalp (dialog-item-text changer-text-item)
                           "Change me and see what happens")
             (message-dialog "You changed me!"))
 t)

(defmethod enter-key-handler
           ((changer-text-item changer-text-item) old-text)
           (unless (equalp (dialog-item-text Checking)
                           "Click here to check")
             (set-current-key-handler
              (view-window changer-text-item) old-text)))

(setq foo (make-instance 'dialog))

(setq Changing (make-instance 'changer-text-item
                :view-position #@(10 100)
                :draw-outline nil))

(add-subviews foo Checking Changing)
```

allow-returns-p

[*Generic function*]

Syntax `allow-returns-p (`*item* `key-handler-mixin)`

Description The generic function`allow-returns-p` returns true if carriage returns are allowed in the editable-text dialog item. Otherwise, it returns false.

Argument *item* An editable-text dialog item.

set-allow-returns

[*Generic function*]

Syntax `set-allow-returns (`*item* `key-handler-mixin)` *value*

Description The generic function`set-allow-returns` sets whether carriage returns are allowed in the editable-text dialog item.

Arguments *item* An editable-text dialog item.

value If *value* is true, carriage returns are allowed. If it is `nil`, they are not.

allow-tabs-p

[*Generic function*]

Syntax `allow-tabs-p (`*item* `key-handler-mixin)`

Description The `allow-tabs-p` generic function returns true if tabs are allowed in the editable-text dialog item. Otherwise, it returns false.

Argument *item* An editable-text dialog item.

set-allow-tabs

[*Generic function*]

Syntax `set-allow-tabs (`*item* `editable-text-dialog-item)` *value*

Description The `set-allow-tabs` generic function sets whether tabs are allowed in the editable-text dialog item.

Arguments *item* An editable-text dialog item.

value If *value* is true, tabs are allowed. If it is`nil`, they are not.

cut [*Generic function*]

copy [*Generic function*]

paste [*Generic function*]

clear [*Generic function*]

undo [*Generic function*]

undo-more [*Generic function*]

select-all [*Generic function*]

Syntax

```
cut (window window)
copy (window window)
paste (window window)
clear (window window)
undo (window window)
undo-more (window window)
select-all (window window)
```

Description These generic functions are each specialized on the `window` class (as well as on `fred-mixin`; see Chapter 14, "Programming the Editor"). Each generic function calls the same generic function on the current key handler of *window*, if there is one. The methods applicable to `fred-mixin` perform the operation.

Argument *window* A window whose first direct superclass is `fred-mixin`, which provides editing capability.

Checkboxes

Checkboxes are small squares that toggle an X mark on and off when clicked. The following class and functions govern the behavior of checkboxes.

check-box-dialog-item [*Class name*]

Description The `check-box-dialog-item` class is the class of checkbox dialog items.

initialize-instance

[*Generic function*]

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

Description The `initialize-instance` primary method for `check-box-dialog-item` initializes a checkbox dialog item. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments

item A checkbox dialog item.

initargs A list of keywords and values used to initialize the checkbox. The additional initialization argument keyword for checkboxes is

`:check-box-checked-p`
This keyword specifies whether the item is initially checked. Its value is true if the item is checked and `nil` if it is not. Its default value is `nil`.

dialog-item-action

[*Generic function*]

Syntax `dialog-item-action` (*item* `check-box-dialog-item`)

Description The `check-box-dialog-item` primary method for `dialog-item-action` toggles the state of the box from unchecked to checked or vice versa, then calls `call-next-method` .

Argument

item A checkbox dialog item.

check-box-check

[*Generic function*]

Syntax `check-box-check` (*item* `check-box-dialog-item`)

Description The `check-box-check` generic function places an X in the checkbox. The function merely places an X in the box; it does not run the action of the dialog item.

Argument

item A checkbox dialog item.

check-box-uncheck [*Generic function*]

Syntax `check-box-uncheck` (*item* `check-box-dialog-item`)

Description The `check-box-uncheck` generic function removes the X from the checkbox. The function merely removes the X from the box; it does not run the action of the dialog item. The function returns `nil`.

Argument *item* A checkbox dialog item.

check-box-checked-p [*Generic function*]

Syntax `check-box-checked-p` (*item* `check-box-dialog-item`)

Description The `check-box-checked-p` generic function returns `t` if there is an X in the checkbox and `nil` otherwise. The function merely reports on the state of the box; it does not run the action of the dialog item.

Argument *item* A checkbox dialog item.

Radio buttons

Radio buttons are small circles that contain a black dot when they are selected (“pushed”). Radio buttons occur in clusters, and only one button in a cluster may be pushed at a time. Clicking a radio button unpushes the previously pushed one. The following class and functions govern the behavior of radio buttons.

radio-button-dialog-item [*Class name*]

Description The `radio-button-dialog-item` class is the class of radio-button dialog items.

initialize-instance

[*Generic function*]

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

Description The `initialize-instance` primary method for `radio-button-dialog-item` initializes a radio-button dialog item. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments

item A radio-button dialog item.

initargs A list of keywords and values used to initialize a radio button. The *initargs* keywords, in addition to those for `dialog-item` , are

`:radio-button-cluster`
The cluster to which the radio button belongs. Only one button from a given cluster can be pushed at a time. Whenever the user clicks a button, the function `radio-button-unpush` is applied to all other buttons having the same value for `radio-button-cluster`. To check to see whether two buttons are in the same cluster, use `eq`. The default cluster is 0.

`:radio-button-pushed-p`
This keyword determines whether the radio button is initially pushed. The default value is `nil`.

radio-button-cluster

[*Generic function*]

Syntax `radio-button-cluster` (*item* `radio-button-dialog-item`)

Description The `radio-button-cluster` generic function returns the cluster of *item* as an integer.

Argument

item A radio-button dialog item.

pushed-radio-button [*Generic function*]

Syntax `pushed-radio-button` (*window* `window`) `&optional` *cluster*

Description The `pushed-radio-button` generic function returns the pushed radio button from the specified cluster. The value `nil` is returned if there is no such cluster or if all the radio buttons in a cluster are disabled.

Arguments

window A window.

cluster The cluster of radio buttons to search. Radio button clusters are numbered, starting with 0. The default is 0.

radio-button-push [*Generic function*]

Syntax `radio-button-push` (*item* `radio-button-dialog-item`)

Description The `radio-button-push` generic function pushes a radio button and unpushes the previously pushed one. The function merely toggles the states of the two radio buttons; it does not run any action. The function returns `nil`.

Argument

item A radio-button dialog item.

radio-button-unpush [*Generic function*]

Syntax `radio-button-unpush` (*item* `radio-button-dialog-item`)

Description The `radio-button-unpush` generic function unpushes the radio button and returns `nil`.

Argument

item A radio-button dialog item.

radio-button-pushed-p [*Generic function*]

Syntax `radio-button-pushed-p` (*item* `radio-button-dialog-item`)

Description The `radio-button-pushed-p` generic function returns `t` if the radio button is pushed and `nil` if it is not. The default value is `nil`.

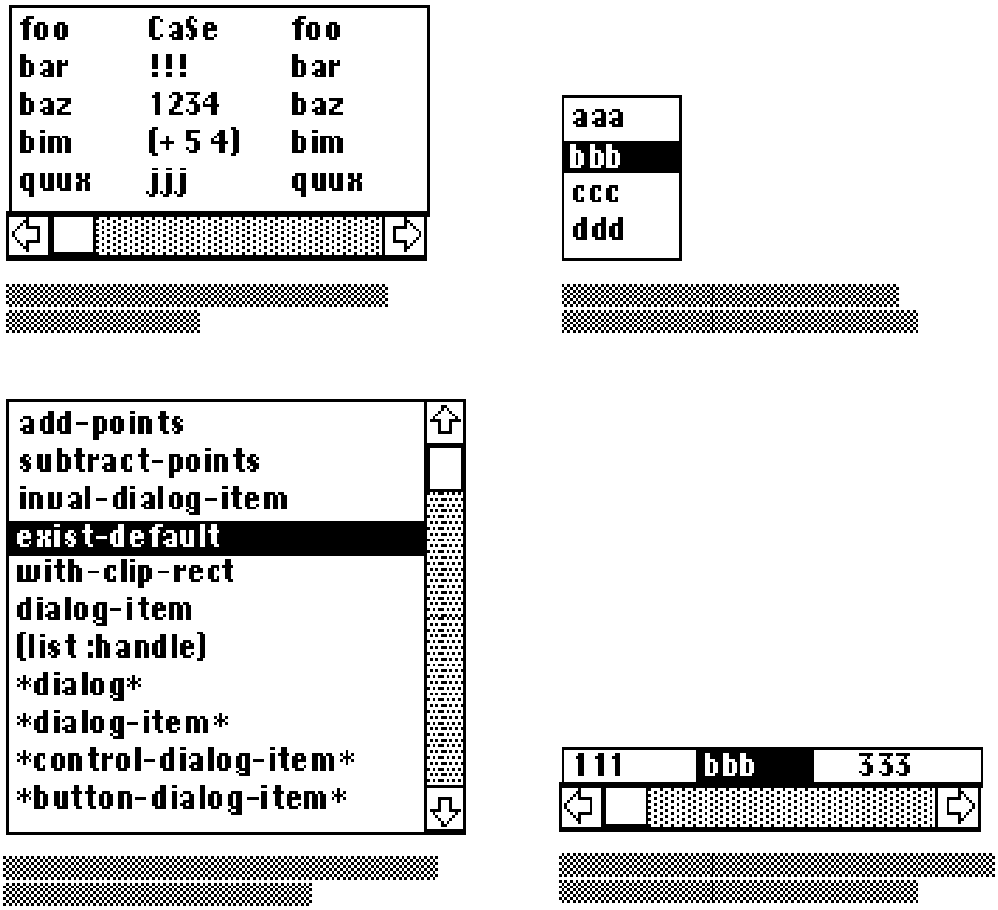
Argument *item* A radio-button dialog item.

Table dialog items

Table dialog items are tables within a window. They allow the user to view a set of items and select items from the set. These tables may be one- or two-dimensional (see Figure 6-1). Two-dimensional tables look like spreadsheets. One-dimensional tables look like the file selection boxes displayed after a user chooses the Save as command. Each item in a table takes up one cell, and there is an 8 KB limit on the total number of cells a table may have.

Table dialog items are implemented using the Macintosh List Manager (but are not called “lists” to avoid confusion with Lisp lists).

n **Figure 6-1** Examples of tables used in dialog boxes



All the functions used with other dialog items (such as `view-size` and `view-position`) work for tables, except that the text of table dialog items is not shown.

Table dialog items are rectangles with a series of cells (see Figure 6-2). Your program can access information about table dialog items, such as the cells that are selected, the position of any cell, and the contents of any cell.

A cell is referenced by a point, encoding the horizontal and vertical indices of the cell within the table.

n **Figure 6-2** Cell positions represented as points

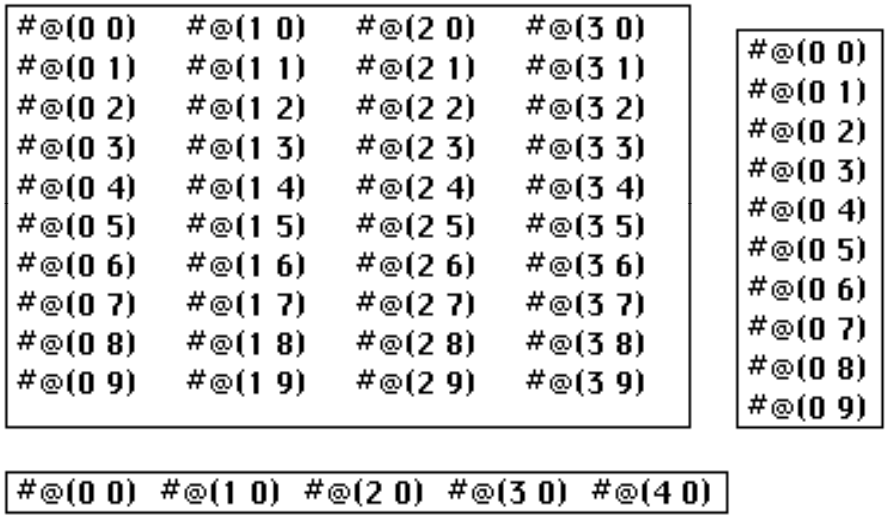


table-dialog-item

[*Class name*]

Description The `table-dialog-item` class provides the base functionality for all types of table dialog items. You should not directly instantiate this class, but should create subclasses from it.

The common uses of table dialog items are provided by sequence dialog items, described later in this chapter in "Sequence Dialog Items." However, you may want to implement your own subclass of table dialog items with specialized behavior. The file `array-dialog-item.lisp` in your MCL Examples folder implements a class of tables displaying multidimensional arrays.

initialize-instance

[*Generic function*]

Syntax

```
initialize-instance (item table-dialog-item )
  &rest initargs &key :table-dimensions :selection-type
  :table-print-function :table-vscrollp :table-hscrollp
  :grow-icon-p :cell-fonts :cell-size :visible-dimensions
```

Description The `initialize-instance` primary method for `table-dialog-item` initializes a table dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

Arguments

item A table dialog item.

initargs The initialization arguments for the menu item and their initial values, if any. These are its special *initargs* keywords (in addition to those for `dialog-item`):

`:table-dimensions`
The horizontal and vertical dimensions of the table in number of cells, expressed as a point. The default value is `#@(0 0)`. Due to a limitation of the Macintosh List Manager, no table dialog item may have more than 8192 (8 KB) cells.

`:selection-type`
This keyword determines whether the table dialog item allows single or multiple selections, and whether multiple selections must be contiguous. Possible keywords are: `:single`, `:contiguous`, and `:disjoint`. The default value is `:single`.

u *Note:* To get a `:disjoint` selection, you must hold down the Command key as you select items. To get a `:contiguous` selection, hold down the Shift key.

`:table-print-function`
The function used by `draw-cell-contents` to print the contents of the cell. The default value is `#'princ`. If given, this should be a function of two arguments, the value to be printed and the stream.

`:table-vscrollp`
This keyword determines whether the table dialog item has a vertical scroll bar. The default is to include a scroll bar if one is needed in order to view the entire table.

`:table-hscrollp`
: This keyword determines whether the table dialog item has a horizontal scroll bar. The default is to include a scroll bar if one is needed in order to view the entire table.

`:grow-icon-p`
: The value passed as the `HasGrow` parameter to the `#_LNew` trap when `install-view-in-window` creates the table . The default value is `nil`.

`:cell-fonts`
: A property list of cells and font specs. See the description of `set-cell-font` , later in this section.

`:cell-size`
: Horizontal and vertical dimensions of the cells in the table dialog item. The default value is `nil`, meaning that the cell size is computed to be big enough to accommodate the values of all the cells.

`:visible-dimensions`
: The visible dimensions of the table. The default value is `nil`, meaning that the visible dimensions of the table are calculated and the entire table is visible.

`cell-contents` [*Generic function*]

Syntax `cell-contents` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-contents` generic function returns the contents of the cell specified by *h* and *v*. The method for `table-dialog-item` returns `nil`.

The `cell-contents` method should be specialized by subclasses of `table-dialog-item` . It is called by `draw-cell-contents` .

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

redraw-cell

[*Generic function*]

Syntax `redraw-cell` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `redraw-cell` generic function redraws the contents of *cell* . When a single cell changes, calling this function explicitly is much more efficient than redrawing the entire table dialog item.

Redrawing the cell involves three operations:

1. Setting the dialog's clip rectangle so that drawing is restricted to the cell.
2. Moving the pen to a position 3 pixels above the bottom of the cell and 3 pixels to the right of the left edge of the cell.
3. Calling `draw-cell-contents` .

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

draw-cell-contents

[*Generic function*]

Syntax `draw-cell-contents` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `draw-cell-contents` generic function draws the contents of *cell* . It may be shadowed to provide a specialized display. This function should not be called directly. It should be called only by `redraw-cell` , which prepares the window for the drawing.

The default method of `draw-cell-contents` shows the printed representation of the cell contents (using the function stored in the function cell of `:table-print-function` , which defaults to `princ`). If the contents are too long to fit in the cell, an ellipsis is added at the end.

The `draw-cell-contents` function may be shadowed to provide specialized drawing (for example, to create a table of icons or patterns). In many cases, however, you don't need to redefine `draw-cell-contents` ; you can often achieve the desired results with a function in `:table-print-function` .

Arguments	*item*	A table dialog item.
	h	Horizontal index.
	v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

highlight-table-cell

[*Generic function*]

Syntax `highlight-table-cell` (*item* `table-dialog-item`) *cell rect selectedp*

Description The `highlight-table-cell` generic function highlights *cell* . This function may be shadowed to provide a specialized display. The `highlight-table-cell` function should not be called directly. It is automatically called by the `view-click-event` handler for `table-dialog-item` .

Arguments	*item*	A table dialog item.
	cell	The cell to be drawn.
	rect	The bounding rectangle of *cell* .
	selectedp	The state (selected or unselected) of the cell. If the value of *selectedp* is true, the cell is selected. If it is `nil`, the cell is unselected.

table-dimensions

[*Generic function*]

Syntax `table-dimensions` (*item* `table-dialog-item`)

Description The `table-dimensions` generic function returns a point indicating the number of cells horizontally and vertically in the table dialog item.

Argument	*item*	A table dialog item.

set-table-dimensions [*Generic function*]

Syntax `set-table-dimensions` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `set-table-dimensions` generic function sets the number of cells horizontally and vertically according to *h* and *v*.

There is an 8 KB limit on the total number of cells.

Arguments

item	A table dialog item.
h	Horizontal number of cells.
v	Vertical number of cells. If the value of *v* is `nil`, *h* is assumed to represent a point.

visible-dimensions [*Generic function*]

Syntax `visible-dimensions` (*item* `table-dialog-item`)

Description The `visible-dimensions` generic function returns a point indicating the number of cells visible in the horizontal and vertical dimensions.

Argument

item	A table dialog item.

set-visible-dimensions [*Generic function*]

Syntax `set-visible-dimensions` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `set-visible-dimensions` generic function resizes the table so that *h* cells are visible per row and *v* cells are visible per column. The new dimensions are returned as a point.

Arguments

item	A table dialog item.
h	Horizontal number of cells.
v	Vertical number of cells. If the value of *v* is `nil`, *h* is assumed to represent a point.

The `cell-size` and `set-cell-size` functions that follow provide an alternative to `view-size` for specifying the size of a table dialog item.

cell-size

[*Generic function*]

Syntax `cell-size` (*item* `table-dialog-item`)

Description The `cell-size` generic function returns the size of a cell in the table dialog item. All the cells have the same size.

Argument

item	A table dialog item.

set-cell-size

[*Generic function*]

Syntax `set-cell-size` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `set-cell-size` generic function sets the cell size according to *h* and *v* and returns the new size as a point.

Arguments

item	A table dialog item.
h	Horizontal size (width).
v	Vertical size (height). If the value of *v* is `nil`, *h* is assumed to represent a point.

cell-font

[*Generic function*]

Syntax `cell-font` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-font` generic function returns the font used by a cell (specified by *h* and *v*) or `nil` if the cell uses the font of the dialog item.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

set-cell-font [*Generic function*]

Syntax `set-cell-font` (*item* `table-dialog-item`) *cell font-spec*

Description The `set-cell-font` generic function sets the font of *cell* to *font-spec* .

Arguments

item	A table dialog item.
cell	A cell in the table dialog item, encoded as a point.
font-spec	A font spec.

part-color [*Generic function*]

Syntax `part-color` (*item* `table-dialog-item`) *part*

Description The `part-color` method for `table-dialog-item` returns the color of the part of the table dialog item indicated by *part*.

Arguments

item	A table dialog item.
part	A keyword. In addition to the keywords allowed for dialog items, *part* may be a point indicating a cell.

set-part-color [*Generic function*]

Syntax `set-part-color` (*item* `table-dialog-item`) *part color*

Description The `set-part-color` method for `table-dialog-item` sets the color of part of the table dialog item, as specified by the arguments, and returns *color* .

Arguments

item	A table dialog item.
part	In addition to the keywords allowed for dialog items, *part* may be an integer indicating a cell. The default cell-drawing routine draws the contents of the cell in the color you have specified.
color	A color, encoded as a point.

cell-select

[*Generic function*]

Syntax `cell-select` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-select` generic function selects the cell specified by *h* and *v*. Previously selected cells are not affected.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

cell-deselect

[*Generic function*]

Syntax `cell-deselect` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-deselect` generic function deselects the cell specified by *h* and *v*.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

cell-selected-p

[*Generic function*]

Syntax `cell-selected-p` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-selected-p` generic function returns `t` if the cell specified by *h* and *v* is selected. Otherwise, it returns `nil`.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

selected-cells [*Generic function*]

Syntax `selected-cells` (*item* `table-dialog-item`)

Description The `selected-cells` generic function returns a list of all the cells selected in the table dialog item. Each cell is represented by a point. If no cells are selected, `nil` is returned.

Argument

item	A table dialog item.

scroll-to-cell [*Generic function*]

Syntax `scroll-to-cell` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `scroll-to-cell` generic function causes the table dialog item to scroll so that the cell specified by *h* and *v* is in the upper-left corner.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

scroll-position [*Generic function*]

Syntax `scroll-position` (*item* `table-dialog-item`)

Description The `scroll-position` generic function returns the cell indices of the cell in the upper-left corner of the table dialog item. (This is not a position in window coordinates but indicates which cell is in the upper-left corner.)

Argument

item	A table dialog item.

cell-position

[*Generic function*]

Syntax `cell-position` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `cell-position` generic function returns the position of the upper-left corner of the cell if the cell is visible. It returns `nil` if the cell is not currently visible. The position returned is in the coordinate system of the item's container.

Arguments

item	A table dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

point-to-cell

[*Generic function*]

Syntax `point-to-cell` (*item* `table-dialog-item`) *h* `&optional` *v*

Description The `point-to-cell` generic function returns the cell enclosing the point represented by *h* and *v*, or `nil` if the point is not within a cell.

Arguments

item	A table dialog item.
h	Horizontal position.
v	Vertical position. If the value of *v* is `nil`, *h* is assumed to represent a point.

table-hscrollp

[*Generic function*]

Syntax `table-hscrollp` (*item* `table-dialog-item`)

Description The `table-hscrollp` generic function returns `t` if *item* has a horizontal scroll bar and `nil` otherwise.

Argument

item	A table dialog item.

table-vscrollp [*Generic function*]

Syntax `table-vscrollp (`*item* `table-dialog-item)`

Description The `table-vscrollp` generic function returns `t` if *item* has a vertical scroll bar and `nil` otherwise.

Argument *item* A table dialog item.

table-print-function [*Generic function*]

Syntax `table-print-function (`*item* `table-dialog-item)`

Description The `table-print-function` generic function returns the function used by `draw-cell-contents` to print the contents of the cell.

Argument *item* A table dialog item.

Pop-up menu dialog items

A pop-up menu dialog item is a menu within a dialog box or other view containing dialog items. The Commands menu in Inspector windows is an example of a pop-up menu. For other examples, look at the file `CCL:library;pop-up-menu.lisp` .

The following MCL expressions govern the behavior of pop-up menus.

pop-up-menu [*Class name*]

Description The class `pop-up-menu` is the class of pop-up menus, built on `menu`.

initialize-instance [*Generic function*]

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

Description The `initialize-instance` primary method for `pop-up-menu` initializes a pop-up menu. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments

menu A pop-up menu.

initargs A set of initial arguments and values used for initializing the pop-up menu:

`:default-item`
An integer identifying the default item that will be selected from the menu. The default is `1`. The first item is `1`, not `0`.

`:auto-update-default`
An argument specifying how defaults are handled. If true (the default), each time an item is selected from the pop-up menu, it becomes the default. Otherwise, the default item remains fixed.

`:item-display`
An argument specifying whether the menu item or its value is displayed. If the value is `:selection` (the default), displays the default menu item. Otherwise the value itself is displayed as if by `(format t "~a"` *value*).

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

`:menu-colors`
A property list of menu parts and colors. The allowable parts are given in the definition of `set-part-color` . See the section “Menu Bar Colors” in Chapter 4, “Menus,” and Chapter 7, “Color.”

`:dialog-item-text`
The text of the pop-up menu. The default value is `""`. If a value is specified and is not `""`, this becomes a label for the pop-up menu, which is displayed to the left of the box for the `:item-display` .

`:dialog-item-action`
: The `dialog-item-action` generic function is not called by `view-click-event-handler` for a pop-up menu.

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

Example

See the file `pop-up-menu.lisp` in your MCL Library folder.

Scroll-bar dialog items

A scroll-bar dialog item is a dialog item that is a scroll bar. The following MCL expressions govern the behavior of scroll-bar dialog items.

`scroll-bar-dialog-item` [*Class name*]

Description The `scroll-bar-dialog-item` class is the class of scroll-bar dialog items.

`initialize-instance` [*Generic function*]

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

Description The `initialize-instance` primary method for `scroll-bar-dialog-item` initializes a scroll-bar dialog item. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.)

For full information on scroll bars, see *Inside Macintosh*.

Arguments

item A scroll-bar dialog item.

initargs A set of initial arguments and values used for initializing the scroll-bar dialog item:

`:direction`
The direction of the scroll bar. Valid values are `:horizontal` and `:vertical` (the default).

`:max` The maximum setting of the scroll bar. This value must be an integer; it defaults to 100.

`:min` The minimum setting of the scroll bar. This value must be an integer; it defaults to 0.

`:page-size`
The amount the setting of the scroll bar will change when the user clicks the gray area above or below the scroll box. The default value is 5.

`:scroll-size`
The amount the setting of the scroll bar will change when the user clicks one of the arrows at its two ends. The default value is 1.

`:setting` The initial setting of the scroll bar.

`:track-thumb-p`
An argument specifying behavior during scrolling. If true, the scroll box is moved and `scroll-bar-changed` is called as the user drags the scroll box. Otherwise, an outline is dragged and the scrolling does not actually happen until the user releases the mouse button. The default value is `nil`.

`:scrollee`
An argument specifying what it is that the scroll bar scrolls. The default value is `nil`.

`:pane-splitter`
An argument specifying the position of a pane splitter. If the scroll bar is `:vertical`, a value of `:top` means above the scroll bar and any other non-`nil` value means below it. If the scroll bar is `:horizontal`, a value of `:left` means to the left of the scroll bar and any other non-`nil` value means to the right of it. If `nil`, there is no pane splitter. The default value is `nil`.

`:help-spec`

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

scroll-bar-length

[*Generic function*]

Syntax `scroll-bar-length` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-length` generic function returns the length of *item* .

Argument *item* A scroll-bar dialog item.

set-scroll-bar-length

[*Generic function*]

Syntax `set-scroll-bar-length` (*item* `scroll-bar-dialog-item`) *new-length*

Description The `set-scroll-bar-length` generic function sets the length of *item* to *new-length* .

Arguments *item* A scroll-bar dialog item.
new-length The new length of *item* .

scroll-bar-max

[*Generic function*]

Syntax `scroll-bar-max` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-max` generic function returns the maximum setting of *item* .

Argument *item* A scroll-bar dialog item.

set-scroll-bar-max

[*Generic function*]

Syntax `set-scroll-bar-max` (*item* `scroll-bar-dialog-item`)
new-value

Description The `set-scroll-bar-max` generic function sets the maximum setting of *item* to *new-value* .

Arguments

item	A scroll-bar dialog item.
new-value	The new maximum setting of *item* .

scroll-bar-min

[*Generic function*]

Syntax `scroll-bar-min` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-min` generic function returns the minimum setting of *item* .

Argument

item	A scroll-bar dialog item.

set-scroll-bar-min

[*Generic function*]

Syntax `set-scroll-bar-min` (*item* `scroll-bar-dialog-item`)
new-value

Description The `set-scroll-bar-min` generic function sets the minimum setting of *item* to *new-value* .

Arguments

item	A scroll-bar dialog item.
new-value	The new minimum setting of *item* .

scroll-bar-page-size

[*Generic function*]

Syntax `scroll-bar-page-size` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-page-size` generic function returns the page size of *item* .

Argument

item	A scroll-bar dialog item.

scroll-bar-scroll-size [*Generic function*]

Syntax `scroll-bar-scroll-size` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-scroll-size` generic function returns the scroll size of *item* .

Argument *item* A scroll-bar dialog item.

scroll-bar-scrollee [*Generic function*]

Syntax `scroll-bar-scrollee` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-scrollee` generic function retrieves the scrollee of *item* (that is, what *item* is scrolling).

Argument *item* A scroll-bar dialog item.

set-scroll-bar-scrollee [*Generic function*]

Syntax `set-scroll-bar-scrollee` (*item* `scroll-bar-dialog-item`) *new-scrollee*

Description The `set-scroll-bar-scrollee` generic function sets the scrollee of *item* (that is, what *item* is scrolling) to *new-scrollee* .

Arguments *item* A scroll-bar dialog item.
new-scrollee The new scrollee of *item* .

scroll-bar-setting [*Generic function*]

Syntax `scroll-bar-setting` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-setting` generic function returns the current setting of *item* .

Argument *item* A scroll-bar dialog item.

set-scroll-bar-setting [*Generic function*]

Syntax `set-scroll-bar-setting` (*item* `scroll-bar-dialog-item`) *new-setting*

Description The `set-scroll-bar-setting` generic function sets the setting of *item* to *new-setting* . It does not call `dialog-item-action` .

Arguments

item	A scroll-bar dialog item.
new-setting	The new setting of *item* .

scroll-bar-track-thumb-p [*Generic function*]

Syntax `scroll-bar-track-thumb-p` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-track-thumb-p` generic function returns a value indicating the behavior of *item* when the scroll box is dragged. If true, the scroll box moves and the function `scroll-bar-changed` is called as the user drags the scroll box. If `nil`, only an outline of the scroll box moves and scrolling does not occur until the user releases the mouse button. The default value is `nil`.

Argument

item	A scroll-bar dialog item.

set-scroll-bar-track-thumb-p [*Generic function*]

Syntax `set-scroll-bar-track-thumb-p` (*item* `scroll-bar-dialog-item`) *value*

Description The `set-scroll-bar-track-thumb-p` generic function sets the value controlling the behavior of *item* when the scroll box is dragged. If true, the scroll box moves and the function `scroll-bar-changed` is called as the user drags the scroll box. If `nil`, only an outline of the scroll box moves and scrolling does not occur until the user releases the mouse button.

Arguments

item	A scroll-bar dialog item.
value	A Boolean value. If *item* does not have a scroll box, the value is `nil`.

scroll-bar-width [*Generic function*]

Syntax `scroll-bar-width` (*item* `scroll-bar-dialog-item`)

Description The `scroll-bar-width` generic function returns the width of *item* .

Argument

item	A scroll-bar dialog item.

set-scroll-bar-width [*Generic function*]

Syntax `set-scroll-bar-width` (*item* `scroll-bar-dialog-item`) *new-width*

Description The `set-scroll-bar-width` generic function sets the width of *item* to *new-width* .

Arguments

item	A scroll-bar dialog item.
new-value	The new width of *item* .

scroll-bar-changed [*Generic function*]

Syntax `scroll-bar-changed` (*scrollee* `t`) (*scroll-bar* `t`)

Description The `scroll-bar-changed` generic function is called by the `dialog-item-action` method for `scroll-bar-dialog-item` if the `dialog-item-action-function` specified by the `:dialog-item-action` initialization argument is `nil`. The *scrollee* argument is the value of (`scroll-bar-scrollee` *scroll-bar*), as set by `set-scroll-bar-scrollee` or the `:scrollee` initialization argument for *scroll-bar* . The default method does nothing.

Writing a `scroll-bar-changed` method is an easy way to cause user mouse clicks on a scroll-bar dialog item to update another view.

Arguments

scrollee	A scroll-bar scrollee; what is scrolled by the dialog item.
scroll-bar	A scroll bar.

Sequence dialog items

A sequence dialog item is a table dialog item that displays the elements of a sequence, either row by row or column by column. The following class and functions govern the behavior of sequence dialog items.

sequence-dialog-item [*Class name*]

Description The `sequence-dialog-item` class is the class of sequence dialog items, used for displaying the elements of a sequence. It is a subclass of `table-dialog-item` . Each instance has an associated sequence. The elements of the sequence are displayed in a table dialog item, in a single row or column, or in multiple rows and columns. The table dialog item has multiple rows and columns only if the length of the sequence is greater than `:sequence -wrap-length` .

initialize-instance [*Generic function*]

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

Description The `initialize-instance` primary method for `sequence -dialog-item` initializes a sequence dialog item. (When instances are actually made, the function used is `make-instance` , which calls `initialize-instance` .)

Arguments

item A sequence dialog item.

initargs A list of keywords and values used to initialize the sequence. These are the *initargs* keywords in addition to those used for `table-dialog-item` :

`:table-sequence`
The sequence to be associated with the table dialog item. This argument must be specified by the user.

`:sequence-order`
This keyword determines whether the sequence will fill the table dialog item row by row or column by column. The value of this keyword should be either `:vertical` or `:horizontal` . The default is `:vertical` .

`:sequence-wrap-length`
The number of elements allowed in a row or column before the table dialog item wraps to the next row or column. This number overrides the `table-dimensions` argument.

table-sequence

[*Generic function*]

Syntax `table-sequence` (*item* `sequence-dialog-item`)

Description The `table-sequence` generic function returns the sequence associated with the dialog item.

Argument *item* A sequence dialog item.

set-table-sequence

[*Generic function*]

Syntax `set-table-sequence` (*item* `sequence-dialog-item`)
new-sequence

Description The `set-table-sequence` generic function sets the sequence associated with the dialog item to *new-sequence* , resets the dimensions of the table dialog item and the scroll bars, and redisplays the dialog item.

Arguments *item* A sequence dialog item.
new-sequence The sequence to be associated with the sequence dialog item. The elements of this sequence are displayed in the cells of the sequence dialog item.

cell-to-index

[*Generic function*]

Syntax `cell-to-index` (*item* `sequence-dialog-item`) *h* `&optional` *v*

Description The `cell-to-index` generic function returns an index into the sequence associated with the dialog item, corresponding to the cell whose indices in the table are *h* and *v*. If there is no such cell, it returns `nil`.

This index is suitable for passing to the Common Lisp function `elt`.

Arguments

item	A sequence dialog item.
h	Horizontal index.
v	Vertical index. If the value of *v* is `nil`, *h* is assumed to represent a point.

index-to-cell

[*Generic function*]

Syntax `index-to-cell` (*item* `sequence-dialog-item`) *index*

Description The `index-to-cell` generic function returns a cell in the dialog item. The cell corresponds to the *index* th element of the table's sequence.

Arguments

item	A sequence dialog item.
index	An index to the sequence (zero based, as would be passed to `elt`).

User-defined dialog items

You can easily add new classes of dialog items to the classes predefined in Macintosh Common Lisp.

New classes of dialog items may be specializations of the types of dialog items listed in this chapter or specializations of the class `dialog-item` . Functions that you may wish to define for classes inheriting from `dialog-item` are listed in "Advanced Dialog Item Functions" earlier in this chapter.

For a commented example of how to implement your own class of dialog item, see the file `scrolling-fred-dialog-item.lisp` in the Library folder distributed with Macintosh Common Lisp.

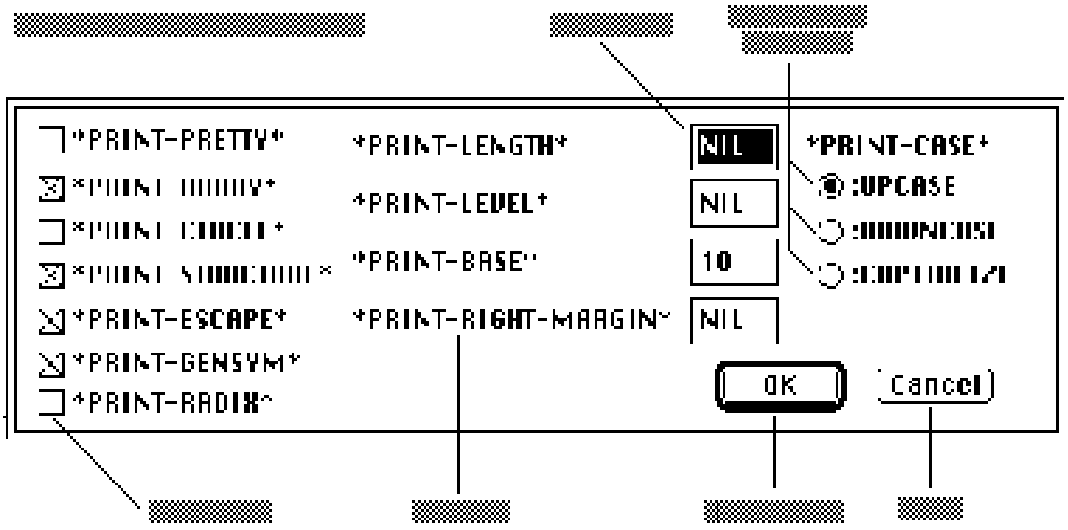
Dialogs

A dialog may be either **modal** or **modeless.**

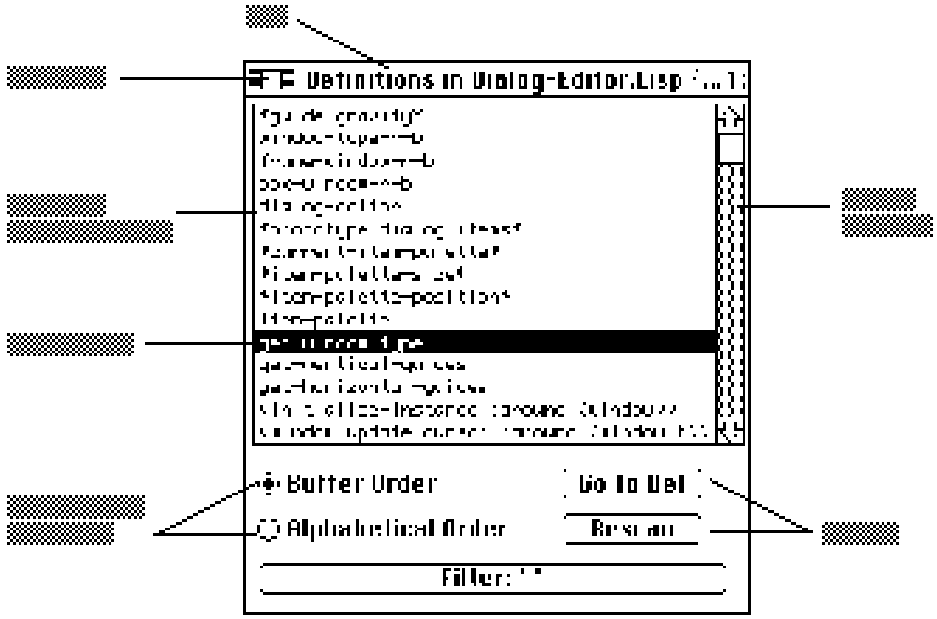
- The user must exit from a modal dialog before performing any other actions. The Print Options dialog box (Figure 6-3) is an example of a modal dialog.
- If the dialog is modeless, other actions can occur while the dialog is still on the screen. The List Definitions dialog box (Figure 6-4) is an example of a modeless dialog.

How the dialog is used determines whether it is modal or modeless. Instance values do not determine its mode.

- **Figure 6-3** A modal dialog (Print Options on the Tools menu)



n **Figure 6-4** A modeless dialog (List Definitions on the Tools menu)



Modal dialogs

A modal dialog is activated by calling the `modal-dialog` generic function on the dialog. The dialog is displayed and made the active window. All subsequent user events are processed by the dialog; illegal events produce a beep, and legal events cause the action of the selected dialog item to be performed. The dialog continues to intercept all events until `return-from-modal-dialog` is called. This macro causes the dialog to be closed or hidden and supplies one or more values to be returned from the call to `modal-dialog` . Modal dialogs may be nested. Command-period can always be pressed to exit one or more modal dialogs.

Some predefined modal dialogs are documented in “Simple Turnkey Dialog Boxes,” later in this chapter.

Modeless dialogs

A modeless dialog is available for use whenever it is visible. Like any window that is not active, a modeless dialog becomes the active window when it is clicked. If a modeless dialog box is the active window, then appropriate user events trigger the actions of its items.

Unless otherwise specified, all the text in a dialog (that is, the text of all the items) appears in the window's current font. The desired font should be set before the dialog is made visible (using `set-view-font` or the `:view-font` initialization argument). A special font may be specified for certain dialog items; the rest of the items appear in the window's current font.

Simple turnkey dialog boxes

Macintosh Common Lisp provides four predesigned dialogs for use by applications.

Three of the following dialog boxes provide facilities for dynamic nonlocal exiting (Common Lisp throwing and catching). Clicking Cancel causes a `throw-cancel` to the nearest `catch-cancel`. If this throw is not caught, clicking Cancel causes a return to the top level (or if it occurs during event processing, the execution of the interrupted program resumes). Common Lisp throw and catch are described in *Common Lisp: The Language.*

`throw-cancel` [*Macro*]

Syntax `throw-cancel &optional` *value-form*

Description The `throw-cancel` macro throws the value of *value-form* to the most recent outstanding `catch-cancel`.

Argument *value-form* A value.

Example

```
? (catch-cancel
    (loop
      (throw-cancel 'foo)))
FOO
```

catch-cancel

[*Macro*]

Syntax `catch-cancel{` *form* `}*`

Description The `catch-cancel` macro sets up a cancel catch and evaluates *form*. It returns the value of the last *form* if there was no cancel throw. Otherwise, it returns the symbol `:cancel`.

Argument *form* Zero or more Lisp forms.

message-dialog

[*Function*]

Syntax `message-dialog` *message* `&key :ok-text :size :position`

Description The `message-dialog` function displays a dialog box containing the string *message* and a single button. The function returns `t` when the user clicks this button or presses Return or Enter.

Arguments

message	A string to be displayed as the message in the dialog box.
`:ok-text`	The text to be displayed in the button. The default button text is `OK`. If the text is too long, this string is clipped (that is, the button is not enlarged to accommodate the longer string). You can set the size with the `:size` keyword.
`:size`	The size of the dialog box. The default size is `#@(335 100)`. A larger size provides more room for text.
`:position`	The position of the dialog box. The default position is the top center of the screen.

Example

```
? (message-dialog "Get along, little dogies"
       :ok-text "Giddyap!" :size #@(250 75))
T
```

Figure 6-5 shows a message dialog box.

n **Figure 6-5** A message dialog box

The file `icon-dialog-item.lisp` in your Examples folder includes a variation of this dialog box containing the standard Macintosh alert icons Stop, Note, and Caution. The file `graphic-items.lisp` in your Library folder shows how to implement generalized graphic items in dialog boxes.

y-or-n-dialog

[*Function*]

Syntax `y-or-n-dialog` *message* `&key :size :position :yes-text :no-text :cancel-text :help-spec`

Description The `y-or-n-dialog` function displays a dialog box containing Yes, No, and Cancel buttons. The display of the dialog box is modal.

If the user clicks the Yes button, the function returns `t`. If the user clicks the No button, the function returns `nil`. If the user clicks the Cancel button, a `throw-cancel` occurs. The default button is the Yes button.

Arguments

message	A string to be displayed as the message in the dialog box.
`:size`	The size of the dialog box. The default size is `#@(318 145)`.
`:position`	The position of the dialog box. The default position is the top center of the screen.
`:yes-text`	The text to be displayed in the Yes button. The default is `Yes`. This is the default button of the dialog box.
`:no-text`	The text to be displayed in the No button. The default text is `No`.

`:cancel-text`
: The text to be displayed in the Cancel button. The default text is `Cancel`. If this argument is `nil` instead of a string, no Cancel button will appear in the dialog box.

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

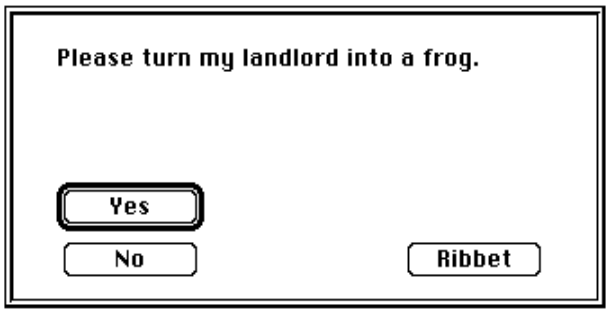
Typing the initial character of the text of a button activates it. For example, typing `Y` activates the Yes button, whereas typing `N` activates the No button. In the following example, typing `R` activates the Cancel button.

Example

```
? (y-or-n-dialog "Please turn my landlord into a frog."
      :cancel-text "Ribbet")
```

Figure 6-6 shows a yes-or-no dialog box.

n **Figure 6-6** A yes-or-no dialog box



get-string-from-user [*Function*]

Syntax `get-string-from-user` *message* `&key :size :position :initial-string :ok-text :cancel-text :modeless :window-title :action-function :allow-empty-strings`

Description The `get-string-from-user` function displays a dialog prompting the user for a string, which it returns. The display of the dialog can be modal or modeless. If the value of `:modeless` is true, the dialog has a close box and no cancel button. If it is `nil`, there is a cancel button and no close box. If the cancel button is clicked, `throw-cancel` occurs.

Arguments

message A string to be displayed as the message in the dialog box.

`:size` The size of the dialog box. The default size is `#@(335 100)`.

`:position` The position of the dialog box. See the *Human Interface Guidelines: The Apple Desktop Interface* for the default position for this dialog box.

`:initial-string` The default string to be displayed in the dialog box.

`:ok-text` The string to be displayed in the default button. If the user clicks this button (or presses Return), `get-string-from-user` returns the current string. The default value is `"OK"`.

`:cancel-text` The string to be displayed in the Cancel button. The cancel button is omitted if the value of `modeless` is true.

`:modeless` An argument specifying whether the dialog box display is modal or modeless. The default is `nil`, meaning that it is modal.

If `:modeless` is specified as true, the `get-string-from-user` function returns the window it creates immediately, without waiting for the user to interact with it. The `action-function` is called when the user clicks the default button or presses the Return or Enter key. The default value is a function that returns the string.

`:window-title` The title of the window. The default is `""`.

`:action-function`

If the `:modeless` argument is true, this argument should be a function of one argument. It is called with the string that the user types each time the user clicks the default button or presses the Return or Enter key. The default value is a function that returns the string.

`:allow-empty-strings`

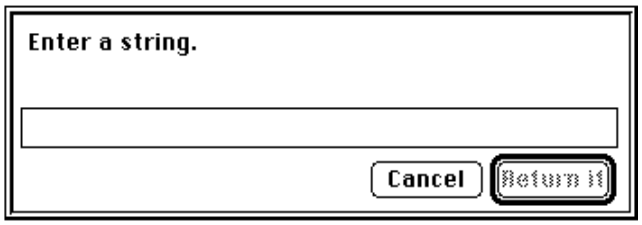
An argument specifying whether the OK button is enabled when the editable-text box contains no text. The default value is `nil`, meaning that the user must type something in the editable-text box to enable the OK button.

Example

```
? (get-string-from-user "Enter a string."
                        :initial-string "A string.")
? (get-string-from-user "Enter a string."
                        :ok-text "Return it")
```

Figure 6-7 shows a dialog box that prompts the user for a string.

Figure 6-7 A `get-string-from-user` dialog box



select-item-from-list [*Function*]

Syntax

```
select-item-from-list list &key :window-title
  :table-print-function :selection-type
  :action-function :modeless :default-button-text
```

Description The `select-item-from-list` function displays a dialog box containing a default button and a table that contains the elements of *list* . The function returns a list of the items selected by the user in reverse order, or `nil` if the user chooses the default button. If the value of `:modeless` is `nil` (the default), the dialog has a cancel button; if the user clicks Cancel, a `throw-cancel` occurs.

Arguments

list A list containing the items to be displayed in the table.

`:window-title` The message displayed at the top of the dialog box.

`:table-print-function` The print function used by the table in the dialog box. The default is `princ` . You can use this argument to customize the dialog box. For example, you could pass a print function that prints only the first element of lists. (See the documentation of this keyword in the section “Table Dialog Items” earlier in this chapter.)

`:selection-type` The type of selection allowed by the table. This should be `:single` , `:contiguous` , or `:disjoint` . The default value is `:single` .

`:action-function` An argument specifying a function to call when the default button is chosen. The function should take one argument, a list of selected items. The default `action-function` returns a list of selected items.

`:modeless` An argument specifying whether the dialog box display is modal or modeless. The default is `t` , meaning that it is modeless.

If `:modeless` is specified as true, the `select-item-from-list` function returns the window it creates immediately, without waiting for the user to interact with it. The `action-function` is called when the user clicks the default button or presses the Return or Enter key.

`:default-button-text`
A string to appear in the default button. The default value is `"OK"`.

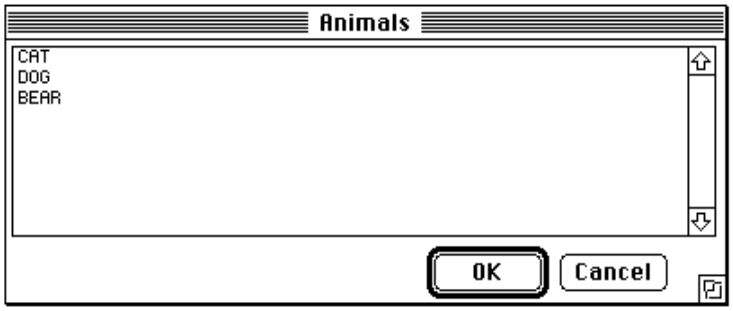
To make a disjoint selection, you must hold down the Command key as you click the selections.

Example

```
? (select-item-from-list '(cat dog bear)
                         :window-title "Animals"
                         :selection-type :disjoint)
; Click the items CAT and BEAR
(BEAR CAT)
```

Figure 6-8 is a modal dialog box that displays a list of items.

n **Figure 6-8** A `select-item-from-list` dialog box



MCL forms relating to dialogs

The following functions, variables, and macros are useful in programming dialogs (that is, to program instances of `view` or `window` that contain dialog items). Remember that any view or window can contain dialog items, which simply act as subviews within the view, and that any generic function that acts on views or windows can act on ones containing dialog items.

`dialog` [*Class name*]

Description The `dialog` class is included for compatibility with earlier versions of Macintosh Common Lisp. No methods in Macintosh Common Lisp version 2 are specialized on `dialog`, and it adds no slots.

Instances of `view` or its subclasses can contain a list of dialog items, as you see in the following example, where dialog items appear in a window.

Example

```
? (setq dialog1 (make-instance 'window
     :window-type :document-with-zoom
     :window-title "Button Dialog"
     :view-position '(:TOP 60)
     :view-size #@(300 150)
     :view-font '("Chicago" 12 :SRCOR :PLAIN)
     :view-nick-name 'button-dialog
     :view-subviews
        (list
         (setq pearlbutton
          (make-dialog-item 'radio-button-dialog-item
             #@(15 28)
             #@(118 16)
             "Pearl Button"
             #'(lambda (item)
                item
               (print "How elegant!"))
              :view-nick-name 'pearlie
              :view-font '("Chicago" 0 :SRCCOPY :PLAIN)))
         (setq flashbutton
          (make-dialog-item 'radio-button-dialog-item
```

```
#@(15 70)
#@(217 16)
"Flashy Plastic Button"
#'(lambda (item)
   item
  (print "How tacky!"))
:view-nick-name 'flash
:view-font '("Chicago" 0 :SRCCOPY :SHADOW))))))
```

modal-dialog

[*Generic function*]

Syntax

`modal-dialog` (*dialog* `window`)`&optional` *close-on-exit eventhook*

Description

The `modal-dialog` generic function displays *dialog* modally. That is, it makes *dialog* the active window, displays it, and then intercepts subsequent user events until `return-from-modal-dialog` is executed. The function returns the value(s) supplied by `return-from-modal-dialog` .

If *close-on-exit* is true (the default), the window is closed on exit; otherwise, it is hidden.

Closing the dialog box automatically prevents the accumulation of numerous hidden windows during development. Modal dialog boxes may be nested.

u *Note:* The body of `modal-dialog` is unwind protected, and so any throw past `modal-dialog` will close or hide the window, as appropriate.

Arguments

window	A window.
close-on-exit	An argument determining whether the window should be closed or simply hidden when the call to `modal-dialog` returns. If this argument is true, the window is closed. If it is false, the window is hidden but not closed. The default is `t` .

eventhook A hook. The function `modal-dialog` binds `*eventhook*` in order to intercept all event processing; this hook is provided so that you can perform any special event processing while the modal dialog is on the screen. The value of *eventhook* should be a function of no arguments, or a list of functions of no arguments. Whenever `modal-dialog` looks for events, it calls the functions in *eventhook* until one of them returns a non-`nil` result. If all of them return `nil`, `modal-dialog` processes events as it normally would. Otherwise, it assumes that the hook function handled the event.

The variable `*current-event*` is bound to an event record for the current event when each hook function is called.

The default value of *eventhook* is `nil`.

`return-from-modal-dialog` [*Macro*]

Syntax `return-from-modal-dialog` *values*

Description The macro `return-from-modal-dialog` causes one or more *values* to be returned from the most recent call to `modal-dialog`.

The dialog is hidden or closed according to the value of *close-on-exit* that was passed to the call to `modal-dialog`. (Any throw past the `modal-dialog` call also causes the dialog box to be hidden or closed.) If the dialog box is only hidden, its contents remain intact and it continues to take up memory until the `window-close` function is explicitly called.

Arguments

values Any values. The following two values have special meanings:

`:closed` If a dialog that is used modally has a close box and the window is closed, `return-from-modal-dialog` is called with the value `:closed`.

`:cancel` If the user selects the cancel button, `return-from-modal-dialog` is called returning `:cancel`. The function `modal-dialog` then performs a `throw-cancel`.

modal-dialog-on-top

[*Variable*]

Description The `*modal-dialog-on-top*` variable is true when a modal dialog is the frontmost window. It is bound during the event processing done by the `modal-dialog` function. Its value is used by the MCL window system code to determine the behavior of floating windows. This value should not be modified by the user, but can be used to determine whether a modal dialog is being processed.

find-dialog-item

[*Generic function*]

Syntax `find-dialog-item` (*dialog* `dialog`) *string*

Description The `find-dialog-item` generic function returns the first item in the view whose `dialog-item-text` is the same as *string* (using `equalp` for the comparison). The items are searched in the order in which they were added to the view.

This function may yield unexpected results in views with editable text items. If the user types text identical to the text of another item, the editable-text item may be returned instead of the desired item. For this reason, `find-dialog-item` is best used during programming and debugging sessions.

To identify items in a dialog, you should use nicknames and the functions `view-named` and `find-named-sibling` .

Arguments

dialog	A view or window containing dialog items.
string	A string against which to compare the text of the dialog items.

Chapter 7 **Color**

Contents

Color encoding in Macintosh Common Lisp / 262
MCL expressions governing color / 263
Operations on color windows / 270
    Coloring user interface objects / 272
    Part keywords / 274
        Menu bar / 274
        Menus / 274
        Menu items / 275
        Windows / 275
        Dialog items / 275
        Table dialog items / 276

This chapter describes the implementation of color in Macintosh Common Lisp.

Macintosh Common Lisp includes high-level tools for handling colors. There are functions for encoding and decoding colors (much as points are encoded and decoded), and there are tools for setting the colors of user interface components (windows, menus, and so on).

You should read this chapter before programming color into your application.

For a complete description of color operations on the Macintosh computer, see *Inside Macintosh* .

Color encoding in Macintosh Common Lisp

The Macintosh stores colors as 48-bit red-green-blue (RGB) values, with 16 bits each for the red, green, and blue components. Because current hardware generally supports a maximum of 24 bits of color, Macintosh Common Lisp encodes colors as fixnums with 8 bits each for red, green, and blue (and 5 bits unused). Therefore, creating a color encoding does not allocate memory.

If your application requires more than 24 bits of color, you can redefine the color encoding and decoding operations.

Although they are stored as 8-bit values when encoded in a color, decoded components are expressed as 16-bit values. This allows compatibility with some Macintosh tools (such as the Color Picker). Unfortunately, it also means that the low 8 bits of each color component are lost when the color is encoded and decoded. For example, consider the following expressions, in which the red component of two colors differs in the low 8 bits. Encoding and decoding loses information:

```
? (make-color 32256 14000 27323)          ;;#$7E00=32256
8271466
? (eql 32256 (color-red 8271466))
T
? (make-color 32333 14000 27323)          ;;#$7E4D=32333
8271466
? (equal 32333 (color-red 8271466))
NIL
```

To compare colors for equality as they are actually displayed on the current display device, use the function `real-color-equal` .

```
? (real-color-equal (make-color 32256 14000 27323)
                    (make-color 32333 14000 27323))
T
```

MCL expressions governing color

This section describes the MCL expressions that govern color.

color-available [*Variable*]

Description The `*color-available*` variable returns a value indicating whether the Macintosh computer on which Macintosh Common Lisp is running supports Color QuickDraw.

If the value of this variable is non-`nil`*, then* the Macintosh computer supports the Color QuickDraw command set. If 32-bit QuickDraw is available, its value is 32.

If the value of this variable is `nil`*, then* Color QuickDraw is not available.

This variable should never be changed by a program.

make-color [*Function*]

Syntax `make-color` *red green blue*

Description The `make-color` function returns an encoded color, with components *red* , *green* , and *blue* . The components should be in the range 0–65535. Each component is stored with an accuracy of ±255.

Arguments

red	The red component of the color. This should be an integer in the range 0–65535.
green	The green component of the color. This should be an integer in the range 0–65535.
blue	The blue component of the color. This should be an integer in the range 0–65535.

Example

Note that the color components change value as they are encoded and decoded.

```
? (make-color 32333 14000 27323)
8271466
? (color-values 8271466)
32256
13824
27136
```

color-red

[*Function*]

Syntax `color-red` *color*

Description The `color-red` function returns the red component of *color* as an integer in the range 0–65535.

Argument *color* A color.

Example

```
? (color-red 8271466)
32256
? (color-red *purple-color*)
17920
```

color-green

[*Function*]

Syntax `color-green` *color*

Description The `color-green` function returns the green component of *color* as an integer in the range 0–65535.

Argument *color* A color.

Example

```
? (color-green 8271466)
13824
? (color-green *purple-color*)
0
```

color-blue

[*Function*]

Syntax `color-blue` *color*

Description The `color-blue` function returns the blue component of *color* as an integer in the range 0–65535.

Argument *color* A color.

Example

```
? (color-blue 8271466)
27136
? (color-blue *purple-color*)
42240
```

color-values

[*Function*]

Syntax `color-values` *color*

Description The `color-values` function returns three values corresponding to the red, green, and blue components of *color* .

Argument *color* A color.

Example

```
? (color-values 8271466)
32256
13824
27136
```

real-color-equal

[*Function*]

Syntax `real-color-equal` *color1 color2*

Description The `real-color-equal` function returns true if *color1* and *color2* are displayed as the same color on the current display device. Otherwise it returns false.

This function may return different results for the same arguments, depending on the current configuration of the computer running Macintosh Common Lisp. For information on the algorithm used to map RGB colors into Macintosh color-table entries, see *Inside Macintosh*.

Arguments

color1	A color.
color2	Another color.

Example

```
? (real-color-equal (make-color 32256 14000 27323)
                    (make-color 32333 14000 27323))
T
```

color-to-rgb

[*Function*]

Syntax

`color-to-rgb` *color* `&optional` *rgb*

Description

The `color-to-rgb` function returns a Macintosh RGB record describing the same color as *color* . RGB records are allocated on the Macintosh heap and are therefore not subject to garbage collection. They must be explicitly deallocated with a call to `dispose-record` or `#_DisposPtr` . For this reason, it is recommended that the macro `with-rgb` be used instead whenever possible.

Most Color QuickDraw traps receive colors in the form of RGB records.

Arguments

color	A color.
rgb	A macptr to an RGB record. The record may be combined with *color* to produce the returned record. (See Chapter 16, “Low-Level Operating System Interface,” for information on macptrs.)

Example

```
? (color-to-rgb 8271466)
#<A Mac Zone-Pointer  Size 6 #x611930>
? (print-record * :rgbcolor)
#<Record :RGBCOLOR :RED 32256 :GREEN 13824 :BLUE 27136>
```

But it is preferable to use `with-rgb`:

```
? (let ((color 8271466))
       (when *color-available*
         (with-rgb (rec color)
           (print-record rec :rgbcolor))))
#<Record :RGBCOLOR :RED 32256 :GREEN 13824 :BLUE 27136>
```

rgb-to-color

[*Function*]

Syntax `rgb-to-color` *rgb-record*

Description Given an RGB record, the `rgb-to-color` function returns a corresponding color encoded as an integer.

Most Color QuickDraw traps receive colors in the form of RGB records.

Argument *rgb-record* An RGB color record stored on the Macintosh heap.

Example

```
? (make-record :rgbcolor
   :red 1000
   :green 2000
   :blue 3000)
#<A Mac Zone-Pointer Size 6 #x611940>
? (rgb-to-color *)      ;*=the last value returned
198411
? (color-values *)
768
1792
2816
```

with-rgb

[*Macro*]

Syntax `with-rgb (`*variable color* `) {`*form*`}*`

Description The `with-rgb` macro evaluates *form* with *variable* bound to an RGB record corresponding to the color *color*. When the body of the macro exits, the RGB record is automatically disposed of.

Most Color QuickDraw traps receive colors in the form of RGB records.

Arguments

variable	A symbol bound to an RGB record for the duration of the macro. This position in the macro call is not evaluated.
color	A color encoded as an integer. This position in the macro call is evaluated.
form	Zero or more forms to be executed with *variable* bound to an RGB record containing *color*.

Example

This macro is useful because it saves the trouble of having to allocate RGB records explicitly. (Remember, RGB records are allocated on the Macintosh heap, and so they are not subject to garbage collection.)

```
? (defmethod set-fore-color ((window window) color)
    (when *color-available*
      (with-rgb (rec color)
        (with-port (wptr window)
          (#_rgbforecolor :ptr rec)))))
#<Method SET-FORE-COLOR (WINDOW T)>
```

user-pick-color

[*Function*]

Syntax `user-pick-color &key :color :prompt :position`

Description The `user-pick-color` function displays the standard Macintosh Color Picker at `:position`, set to color `:color`, with prompt `:prompt`. It returns the selected color if the user clicks OK or throws to the tag `:cancel` if the user clicks Cancel.

Arguments

`:color`	The default color to bring up in the dialog box. The default is `*black-color*`.
`:prompt`	The prompt to display in the dialog box. The default is `"Pick a color"`.
`:position`	The position of the Color Picker on screen. The default is calculated by Macintosh Common Lisp.

black-color [*Variable*]

white-color [*Variable*]

pink-color [*Variable*]

red-color [*Variable*]

orange-color [*Variable*]

yellow-color [*Variable*]

green-color [*Variable*]

dark-green-color [*Variable*]

light-blue-color [*Variable*]

blue-color [*Variable*]

purple-color [*Variable*]

brown-color [*Variable*]

tan-color [*Variable*]

light-gray-color [*Variable*]

gray-color [*Variable*]

dark-gray-color [*Variable*]

Description These variables contain colors corresponding to the 16 colors available by default on a Macintosh computer with a 16-color monitor.

---

black-rgb [*Variable*]

white-rgb [*Variable*]

Description These variables contain RGB records for black and white.

Operations on color windows

The following operations are used to set the foreground and background colors of windows. If the computer display does not support colors, the colors do not appear. However, they remain associated with the windows, and if the same window is moved to a color monitor, they appear in the proper colors.

Windows created with an omitted or null `:color-p` initarg can display only eight colors. Specify `:color-p` as true to use the full range of colors supported by the hardware.

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

Syntax `set-fore-color` (*window* `window`) *color*

Description The `set-fore-color` generic function sets the foreground color of the window to *color* and returns `nil`. Future drawing in the window appears in this color; when the window is redrawn, all drawing appears in this color.

Arguments

window	A window.
color	A color.

Example

```
? (setq mywin (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4BEE99>
? (set-fore-color * *blue-color*)
NIL
```

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

Syntax `set-back-color` (*window* `window`) *color* `&optional` *redisplay-p*

Description The `set-back-color` generic function sets the background color of the window to *color* and returns `nil`.

Arguments

window	A window.
color	A color.
redisplay-p	If the value of this is true (the default), this function invalidates the window, forcing a redrawing. The displayed background color does not change unless the window is redrawn.

Example

```
? (set-back-color mywin *yellow-color* t)
NIL
```

with-fore-color

[*Macro*]

Syntax `with-fore-color` *color* {*form*}*

Description The `with-fore-color` macro sets the foreground color of the window to *color* and executes *form*. When the body of the macro exits, the old foreground color is restored.

This macro should be used only with a port set. That is, it should be used within the dynamic extent of a call to `with-port` or `with-focused-view`.

If Color QuickDraw is not present or *color* is `nil`, the color is not set.

Arguments

color	A color.
form	Zero or more forms to be executed with the foreground color set.

Example

```
? (setq my-new-win (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4D1399>
? (defmethod type-in-color ((view view) color string)
    (with-focused-view view
      (with-fore-color color
        (princ (format nil "~s" string) view))))
#<STANDARD-METHOD TYPE-IN-COLOR (VIEW T T)>
? (type-in-color my-new-win *blue-color* "Hi there")
NIL
```

with-back-color [*Macro*]

Syntax `with-back-color` *color* {*form*}*

Description The `with-back-color` macro sets the background color of the window to *color* and executes *form*. When the body of the macro exits, the old background color is restored.

This macro should be used only with a port set. That is, it should be used within the dynamic extent of a call to `with-port` or `with-focused-view`.

If Color QuickDraw is not present or *color* is `nil`, the color is not set.

Arguments

color	A color.
form	Zero or more forms to be executed with the background color set.

Coloring user interface objects

Methods on the following functions are used for setting the colors of user interface objects such as windows, dialog items, menus, and menu items. This section assumes some familiarity with the use of these classes.

For each class, a set of keywords identifies the parts that can be colored. The keywords for the different classes are given in the next section, “Part Keywords.”

If a user defines a new class of dialog items, the generic function `view-draw-contents` can be defined to use the colors of the parts of the dialog item.

part-color [*Generic function*]

Syntax `part-color` *object part*

Description The `part-color` generic function returns the color of the part of *object* indicated by *part*.

Arguments

object	A user interface object. Built-in methods are defined for `window`, `dialog-item`, `menubar`, `menu`, and `menu-item`.
part	A keyword associated with the class of object. The part keywords are described in the next section.

set-part-color

[*Generic function*]

Syntax `set-part-color` *object part color*

Description The `set-part-color` generic function sets the part of *object* indicated by *part* to *color* and returns *color*, encoded as an integer. If *color* is `nil`, the default color is restored.

Arguments

object	A user interface object. Built-in methods are defined for `window`, `dialog-item`, `menubar`, `menu`, and `menu-item`.
part	A keyword associated with the class of object. The part keywords are described in the next section.
color	A color.

part-color-list

[*Generic function*]

Syntax `part-color-list` *object*

Description The `part-color-list` generic function returns a property list of keywords and colors for all the colored components of *object*. The same keywords apply as for `part-color`. Components whose color has not been set are not included.

Argument

object	A user interface object. Built-in methods are defined for `window`, `dialog-item`, `menubar`, `menu`, and `menu-item`.

Example

Here is an example of the use of part keywords with these functions:

```
? (setf w (make-instance 'window))
#<WINDOW "Untitled" #x3E9229>
? (part-color w :content)
NIL
? (set-part-color w :content *blue-color*)
212
? (part-color w :content)
212
? (part-color-list w)
(:CONTENT 212)
```

Part keywords

You can perform color operations on six objects: menu bars, menus, menu items, windows, dialog items, and table dialog items. This section presents the keywords that identify which parts of certain objects can be colored.

Menu bar

To perform color operations on the menu bar, use the value of the variable `*menubar*`, which contains the one instance of the class `menubar`. You can color the menu bar's titles and its background using the following keywords:

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

`:default-menu-background`
: The default color used for the background of the menus in the menu bar.

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

`:menubar` The background color of the menu bar.

Menus

You can color three parts of menus.

`:menu-title`
: The color used for the title of the menu.

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

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

Menu items

You can color three parts of menu items.

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

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

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

Windows

The window part keywords correspond to different features in different types of windows, because the Macintosh Toolbox uses window color records differently for different window types. You can color windows using these keywords.

`:content` The color used for the background of the window.

`:frame` The color used for the outline of the window and the title bar of :tool windows.

`:text` The color used for the title of :document windows.

`:hilite` The color used for the lines in the title bar of `:document` windows.

`:title-bar`
: The color used for the background of the title bar in :document windows and the title in `:tool` windows.

Dialog items

These part keywords work for built-in dialog items (although not all dialog items have all of these features). You may wish to use the part colors in the `view-draw-contents` method for dialog item classes you define.

`:frame` The color used for the outline of the dialog item.

`:text` The color used for the text of the dialog item.

`:body` The color used for the body of the dialog item.

`:thumb` The color used for the scroll box of the dialog item. (Scroll bars are the only built-in dialog item that have a scroll box.)

Table dialog items

The color of individual table cells can be set and accessed. Simply use the cell coordinates as the part keyword. For example, `(set-part-color my-table #@(0 0) 212)` sets the cell in the upper-left corner of the table to blue (which is encoded as 212).

These colors are used only by the default `draw-cell-contents` function. If you define your own `draw-cell-contents`, you must use `part-color` to access and install the color when you draw the cell.

Chapter 8 **The Interface Toolkit**

Contents

The Interface Toolkit / 279
Loading the Interface Toolkit / 279
Editing menus with the Interface Toolkit / 280
- Using the menu editing functionality / 280
- Creating a new menu bar: Add New Menubar / 282
- Getting back to the default menu bar: Rotate Menubars / 282
- Deleting a menu bar: Delete Menubar / 283
- Creating and editing menus: Add Menu / 283
- Creating menu items / 283
- Editing menu items / 284
- Saving a menu bar / 285
- Editing menu bar source code / 285

Editing dialogs with the Interface Toolkit / 286
- Using the dialog-designing functionality / 287
- Dialog-designing menu items / 287
- Creating dialog boxes / 288
- Adding dialog items / 290
- Editing dialog items / 291

The Interface Toolkit is an application built on top of Macintosh Common Lisp. It is provided as source code in the Interface Tools folder distributed with Macintosh Common Lisp; you can examine and modify it for your own use. It is also useful for building interfaces, and that aspect of it is documented here.

The Interface Toolkit does two things: edits menus and menu bars, and creates and edits dialog boxes. In addition, it prints source code for everything it creates.

You do not need to be familiar with the MCL implementation of menus and dialog boxes before using the Interface Toolkit. You should read Chapter 4, “Menus,” Chapter 5, “Views and Windows,” and Chapter 6, “Dialog Items and Dialogs,” as you look at the source code.

If you are a novice user of Macintosh Common Lisp, read Chapter 3, "Prototyping," of *Getting Started With Macintosh Common Lisp,* which takes you through a sample session with the Interface Toolkit.

The Interface Toolkit

The Interface Toolkit, built on top of Macintosh Common Lisp, is an example of a simple MCL application.

It does the following:

- It creates menu bars and populates them with menus.
- It creates and edits dialogs and dialog items.
- For everything it prototypes, it is able to print source code to a file. When you have developed something in the Interface Toolkit, you can save your work to a Fred file, then edit it.

The Interface Toolkit is supplied as source code in the Interface Tools folder. You are free to examine and modify this source code, to use this source code in developing your own applications, and to include it, as is or modified, within your applications.

Loading the Interface Toolkit

Perform these steps to load the Interface Toolkit.

1. **Open the file `make-ift.lisp` and evaluate its contents.**

   In the Listener, choose Open from the File menu.

   Select the file `make-ift.lisp` from the Interface Tools folder.

   Evaluate its contents by choosing Eval Buffer from the Eval menu.

2. **Type the following to the Listener, or evaluate it in a Fred window:**

   ```
   (ift::load-ift)
   ```

   This function loads the files that make up the Interface Toolkit.

Now your menu bar has one additional menu, the Design menu (Figure 8-1).

- **Figure 8-1** The Interface Toolkit menu on the menu bar

Editing menus with the Interface Toolkit

In the Interface Toolkit you can edit the default menu bar or another menu bar to contain any menus you want. You can add menus to a menu bar and remove them. In the same way, you can add menu items to a menu or remove menu items from a menu. You can use menu items from the menus on the standard menu bar or make your own menu items.

You edit both menus and menu items by double-clicking them and specifying their attributes in an edit window.

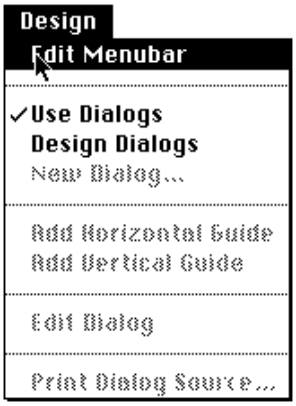
More than one menu bar may be active, and you may edit more than one menu bar at once. You can cut and paste menus among menu bars, including the default menu bar, just as you would cut and paste text from one buffer to another.

At any time, you can generate source code for a menu or for the entire menu bar.

Using the menu editing functionality

After you load the Interface Toolkit, choose Edit Menubar, the first menu item on the Design menu (Figure 8-2). With this menu item you will edit menus and the menu bar.

n **Figure 8-2** Choosing Edit Menubar from the Design menu

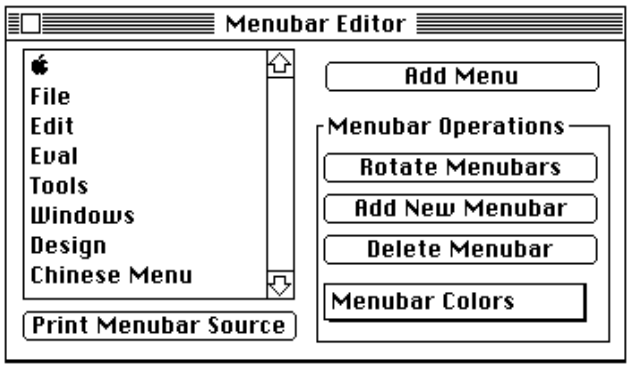


When you choose Edit Menubar from the Design menu, the Interface Toolkit creates two windows, a small floating window and an editor window titled “Menubar Editor”.

The floating window contains the standard editor commands Cut, Copy, Paste, and Clear. You can use this floating window to cut, copy, paste, and clear in situations where you don’t have a working Edit menu.

The Menubar Editor window, shown in Figure 8-3, contains an editable list of the items in the current menu bar.

n **Figure 8-3** The Menubar Editor window



The Menubar Editor window also contains the options listed in Table 8-1.

n **Table 8-1** Menubar Editor window options

Option	Effect
Add Menu	Adds a new, empty menu named “Untitled” to the current menu bar (the one visible in the Menubar Editor’s editable list and at the top of the screen).
Rotate Menubars	If more than one menu bar is active, makes the next menu bar the current menu bar. If only one menu bar is active, this command does nothing.
Add New Menubar	Adds a new, empty menu bar named “Untitled” to the active menu bars. The new menu bar initially contains only the Apple menu.
Delete Menubar	Deletes the current menu bar. The next active menu bar becomes the current menu bar.
Menubar Colors	Sets the colors of the menu bar.
Print Menubar Source	Creates a new Fred window containing the source code for the current menu bar.

Creating a new menu bar—Add New Menubar

To create a new menu bar, choose Add New Menubar from the Menubar Editor window. A new menu bar appears in the Menubar Editor window and at the top of the screen. This new menu bar initially contains only the Apple menu.

You can create any number of new menu bars.

Getting back to the default menu bar—Rotate Menubars

To get to another menu bar or back to the default menu bar, choose Rotate Menubars from the Menubar Editor window.

Deleting a menu bar—Delete Menubar

To delete a menu bar, choose Delete Menubar from the Menubar Editor window. This command deletes the currently installed menu bar and removes it from the rotation.

Creating and editing menus—Add Menu

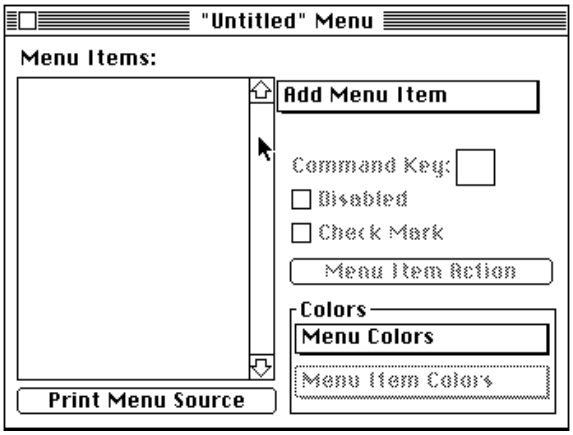
To create a menu, choose Add Menu from the Menubar Editor. The name of the new menu, “Untitled”, appears in the editable list and in the menu bar at the top of the screen.

You can change the name of any menu by choosing it and editing its text. To edit a menu, double-click its name in the list.

Creating menu items

Double-clicking the name of a menu creates a new Menu Editor window for menu items, as shown in Figure 8-4. This window contains an editable list of menu items, which will be empty if the menu is new, and the options listed in Table 8-2.

n **Figure 8-4** A Menu Editor window showing a menu with no items



n **Table 8-2** Menu editing options

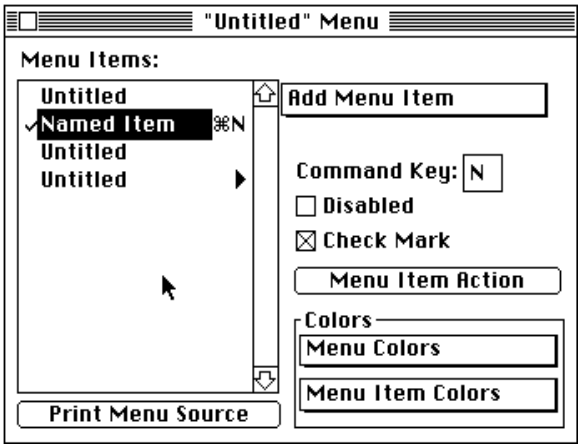
Option	Effect
Add Menu Item	Adds a new, empty menu item named “Untitled” to the current menu. There are three classes of menu items: `menu-item`, a menu item that represents a command; `menu`, a menu item that opens a menu; and `window-menu-item`, a window menu item. (See Chapter 4, “Menus.”) The `menu-item` class defaults to `menu-item`. To change it, edit the menu item source code. You can add further classes by editing the Interface Toolkit source code.
Menu Colors	Sets the colors for parts of the menu.
Print Menu Source	Opens a new Fred window and prints the source code for the menu to it.

Editing menu items

When you add menu items to a menu, you can edit them by double-clicking them, as in Figure 8-5.

Double-clicking a menu item lets you set the features listed in Table 8-3.

n **Figure 8-5** Editing items in the Menu Editor



n **Table 8-3** Menu item editing options

Option	Effect
Command key	Specifies the command key, if any, associated with the menu item.
Disabled	Specifies whether the menu item is disabled. The default is `nil`.
Check Mark	Specifies whether the menu item has a check mark beside it. The default is `nil`.
Menu Item Action	Brings up a Fred window in which you can write or edit code for the menu item action.
Menu Item Colors	Sets the menu item colors.

Saving a menu bar

When you are satisfied with your menu bar, choose Print Menu Source to create source code. Edit your source code as you like, then save it to a file for future use.

The definitions of some menu items in the standard menu bar must be edited. See the next section.

Editing menu bar source code

The Menu Editor is able to print source code for a menu item only if it has access to the source code of the action function of the menu item. If it doesn't, it puts `"Can't find definition"` in the place of the action function source code. You can then edit the code, putting in the real action function definition.

The source code for an action function is available if it was entered directly from the menu editor or loaded from a source file with `*save-definitions*` set to `t`.

It is not available if the menu was loaded from a `fasl` file unless the `fasl` file was compiled with a true value for the `:save-definitions` argument to `compile-file`.

The source code for the action functions of some of the built-in menu items is not available. For example, if you print the source code for the File menu, you need to edit the definition of the New menu item. The definition should make an instance of whatever kind of window you want New to use; for example, if New opens a Fred window, as it does in Macintosh Common Lisp, the definition you add is `(make-instance 'fred-window)` .

You should also delete `INTERFACE-TOOLS::W` from the argument list of the anonymous function.

If you are customizing your MCL menu bar, you may also need to edit the definitions in Table 8-4.

n **Table 8-4** Menu items and corresponding MCL codes

Menu item	MCL code
New	Appropriate code to make an instance of the desired type of window.
Load File	`(load (choose-file-dialog))`
Compile File	`(compile-file (choose-file-dialog :button-string "Compile"))`
Break	`(break)`
Restarts	`(ccl::choose-restart)`
Edit Menubar	`(interface-tools::edit-menubar)`

Editing dialogs with the Interface Toolkit

The Interface Toolkit includes a quick interface designer for dialogs. With it you can create a blank dialog box with any set of attributes you want. Then, from a palette of buttons, radio buttons, checkboxes, editable-text dialog items, tables, and static text, you can drag in dialog items. You can edit them by double-clicking them. In an edit window you can specify the attributes of the dialog item, such as color, font, and associated action.

u *Note:* You can edit the palette to add your own items by editing its source code in the file `item-defs.lisp` , in the Interface Tools folder.

At any time you can generate source code for the dialog box and its items.

u *Note:* When Design Dialogs is checked on the Interface Toolkit's special Design menu,*all* dialog boxes are editable, including the Search/Replace dialog box, the Environment dialog box, and so on. To use dialog boxes rather than edit them, choose Use Dialogs from the Design menu. (If you are in the middle of editing a dialog box, your edits will not disappear; the box will simply become usable.)

Using the dialog-designing functionality

First load the Interface Toolkit according to the directions in the section "Loading the Interface Toolkit" earlier in this chapter.

You see a new menu bar at the top of your screen, containing a Design menu. It should look like the one in Figure 8-2.

Dialog-designing menu items

The Interface Toolkit menu contains eight items, seven of which relate to dialog design (see Table 8-5).

n **Table 8-5** Dialog design menu items

Option	Effect
Edit Menubar	Creates an editor window for the menu bar. This functionality is discussed in the section "Editing Menus With the Interface Toolkit" earlier in this chapter.
Use Dialogs	Allows you to use dialog boxes in your MCL environment. Choosing this menu item automatically disables Design Dialogs, discussed next. These two menu items are the on/off stages of a single toggle. Turning on one turns off the other. When you first load the Dialog Designer, Use Dialogs is enabled. When you are using ordinary MCL dialogs, make sure Use Dialogs is enabled.

(continued)

n **Table 8-5** Dialog design menu items (continued)

Option	Effect
Design Dialogs	Allows you to design dialogs in your MCL environment. Choosing this menu item automatically disables Use Dialogs and makes all dialogs editable, but not usable. (As long as you are in the Interface Toolkit, you can switch back and forth between these modes at will.)
New Dialog...	Brings up a dialog box in which you can specify the type and attributes of a new dialog box. This menu item is discussed in the next section, “Creating Dialog Boxes.”
Add Horizontal Guide	Adds a dotted horizontal guideline to the dialog box. This guideline becomes invisible when you choose Use Dialogs. This menu item is enabled only when you are creating or editing a dialog box.
Add Vertical Guide	Adds a dotted vertical guideline to the dialog window. This guideline becomes invisible when you choose Use Dialogs. This menu item is enabled only when you are creating or editing a dialog box.
Edit Dialog	Allows you to specify the title and position of the window that contains the dialog items. This menu item is enabled only when you are creating or editing a dialog box.
Print Dialog Source...	Prints the source code of the selected dialog box to a new Fred window. This menu item is enabled only when you are creating or editing a dialog box.

Creating dialog boxes

To create a dialog box, first make sure that a check appears next to Design Dialogs. Then choose New Dialog from the Design menu. The system displays a dialog box (Figure 8-6) in which you select the type and attributes of the dialog box you want to create.

n **Figure 8-6** New Dialog dialog box

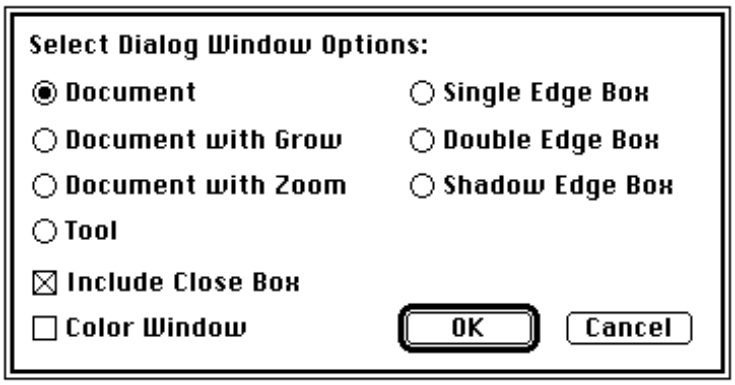


Table 8-6 lists the seven possible types of dialog.

n **Table 8-6** Seven types of dialog

Option	Effect
Document	This is the default. Creates a dialog box with square corners and the title “Untitled Dialog.” By default, a document dialog box includes a close box.
Document with Grow	Creates a document dialog box with a size box.
Document with Zoom	Creates a document dialog box with a size box and a zoom box.
Tool	Creates a dialog box with rounded edges, a solid title bar, and the title “Untitled Dialog.” By default, it also includes a close box.
Single Edge Box	Creates a box with square corners, no title, and no close box. (You must put a close button within a dialog of this type.) Its edge is a single line.
Double Edge Box	Creates a box with square corners, no title, and no close box. Its edge is a double line.
Shadow Edge Box	Creates a box with square corners, no title, and no close box. Its edge is shadowed.

Two attributes are available (see Table 8-7).

n **Table 8-7** Two attributes of dialog boxes

Option	Effect
Include Close Box	Includes a close box in your dialog window. The default value is true.
Color Window	Builds your dialog on top of a Macintosh `CWindowRecord` record.

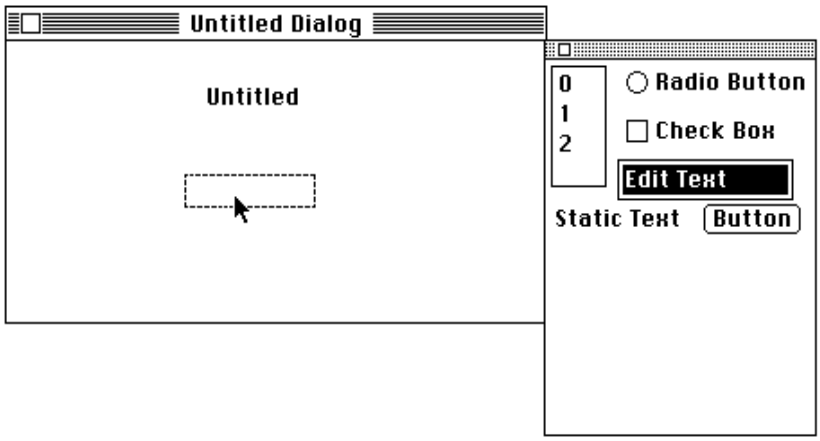
Adding dialog items

Whenever you change from the Use Dialogs menu item to the Design Dialogs menu item, you open a palette of dialog items. If you don't see this palette, choose Use Dialogs, then choose Design Dialogs again. The palette will appear.

The palette contains one of each type of dialog item: a table, a radio button, a checkbox, a field of editable text, some static text, and a button. In Figure 8-7, the palette appears to the right of the new dialog box.

Add dialog items to your dialog box by dragging them from the palette. The original dialog item will remain on the palette, and a copy with the title "Untitled" will appear in your dialog box. Figure 8-7 shows an editable-text dialog item being dragged from the palette to the dialog.

n **Figure 8-7** Dragging an editable-text dialog item into an untitled dialog box



Place dialog items in the dialog box by dragging them. If you want to move the item only vertically or only horizontally, hold down Shift when you drag the box.

To help you place the dialog items, you can add vertical or horizontal guidelines to your dialog box. Click Add Vertical Guide or Add Horizontal Guide in the Design menu. You can select and drag a guide to place it. If you place a dialog item with an edge near a guide, it automatically aligns with the guide.

To resize the display space of any item, first click the item once. Handles (small black boxes) appear around the item. Click the pointer on any of these handles, then drag the item by its handle until you are satisfied with the size.

Editing dialog items

Edit a dialog item by double-clicking it. A dialog box opens. The dialog box varies with the kind of dialog item being edited. Figure 8-8 shows a typical example.

n **Figure 8-8** Edit Dialog Items dialog box

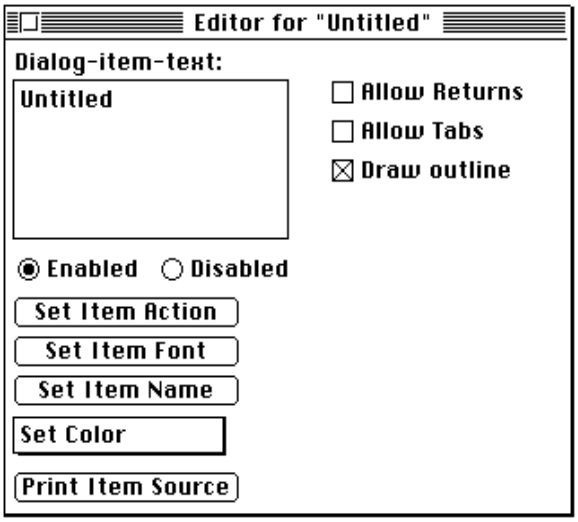


Table 8-8 lists the options available for editing dialog items.

n **Table 8-8** Editable options in dialog items

Option	Effect
Dialog-item-text	Indicates `dialog-item-text` , the label or text the user sees. After you edit this text, you may have to change the size of the dialog item.
Enabled / Disabled	Sets whether the item is enabled or disabled. The default is enabled.
Set Item Action	Sets the code for the action performed by the dialog item.
Set Item Font	Sets the item font. The default is Chicago 12.
Set Item Name	Associates a nickname with the item.
Set Color	Colors one or more parts of the dialog item. You can color the frame, text, body, and thumb.
Print Item Source	Prints the dialog item source code to a new Fred window.

Most dialog item subclasses also allow you to edit special parameters associated with the subclass (see Table 8-9).

n **Table 8-9** Editable options in subclasses of dialog items

Subclass and option	Effect
Radio buttons	
Radio Button Pushed	Indicates whether or not the radio button is selected when the dialog box is first displayed. The default is `nil`.
Set Item cluster	Allows you to move the radio button to a new cluster. Radio button clusters are numbered sequentially, starting with 0. To set the button's cluster, enter a new number.
Buttons	
Default Button	Indicates whether this is the default button. The default value is `nil`.
Edit-text dialog items	
Allow Returns	Indicates whether carriage returns are allowed in the Edit Text field. The default value is `nil`.
Allow Tabs	Indicates whether pressing the Tab key inserts a tab in the buffer or selects the next key handler in the dialog box. The default, `nil`, selects the next key handler.
Draw Outline	Indicates whether an outline is drawn around the dialog item.

Checkboxes

Checkbox Checked	Indicates whether or not the checkbox is checked. The default value is `nil`.

Tables

Set Cell Size	Allows you to set a new default size for table cells.
Horizontal Scroll Bar	Adds a horizontal scroll bar to the table.
Vertical Scroll Bar	Adds a vertical scroll bar to the table.
Set Table Sequence	Sets the sequence in which items appear in the table.
Set Wrap Length	Sets the maximum length a line of text can attain before wrapping to the next line occurs. The default value is `nil`; that is, lines are not wrapped.
Orientation	Determines whether the orientation of the table is vertical or horizontal. The default is vertical.

Chapter 9 **File System Interface**

Contents

Filenames, physical pathnames, logical pathnames, and namestrings / 297
- Changes from earlier versions of Macintosh Common Lisp / 297
- Printing and reading pathnames / 298

Pathname structure / 299
- Macintosh physical pathnames / 300
- Common Lisp logical pathnames / 300
- Defining logical hosts / 301
- Ambiguities in physical and logical pathnames / 301
- More on namestrings and pathnames / 302

Creating and testing pathnames / 303
- Parsing namestrings into pathnames / 305
- The pathname escape character / 306

Loading files / 308

Macintosh default directories / 310

Structured directories / 313

Wildcards / 317

File and directory manipulation / 318
- File operations / 322
- Volume operations / 324

User interface / 327

Logical directory names / 330

This chapter describes filename specification and the functions for manipulating the Macintosh File System. It does not document all Common Lisp file system features, but refers to *Common Lisp: The Language* where appropriate.

You should read this chapter to familiarize yourself with the specification of filenames in Macintosh Common Lisp. It is particularly important if you will deal with other file systems and must translate between them and the file system of Macintosh Common Lisp.

You should be familiar with Chapter 23 of the second edition of *Common Lisp: The Language,* which discusses the Common Lisp file system features.

Filenames, physical pathnames, logical pathnames, and namestrings

The file system interface provides a way of dealing with references to file systems when code may be running on multiple platforms. MCL code must deal with the file system requirements of the Macintosh Operating System and, if the code is meant to be ported, with those of any other operating system on which it is intended to run. Macintosh Common Lisp specifies filenames by means of pathnames, which can be specified as namestrings.

A **filename** is a means of specifying a particular file or directory in a file system. You can represent a filename as either a Lisp object (a pathname) or a string (a namestring). Internally, Macintosh Common Lisp always uses pathnames and converts namestrings to pathnames before using them.

- A **pathname** is a structured Lisp object. It represents a filename as a set of components that can be manipulated in an implementation-independent way. A pathname is not necessarily the name of a file; it is a specification, perhaps partial, of how to access a file.

  A single filename may be represented by two or more quite different pathnames, and the existence of a pathname does not guarantee that the file it specifies exists.

  There are two kinds of pathnames:

  - A **physical pathname** indicates the physical components of the pathname.
  - A **logical pathname** structure has one or more logical components. Logical components may be translated to their physical counterparts.
- A **namestring** is a string that names a file in any one of three syntaxes: Macintosh physical syntax, Common Lisp logical pathname syntax, or MCL logical directory syntax. (MCL logical directory syntax is now deprecated and is likely to disappear in a future release.)

The following sections discuss Macintosh physical syntax and Common Lisp logical pathname syntax. MCL logical directory syntax is described in the section “Logical Directory Names” later in this chapter.

Changes from earlier versions of Macintosh Common Lisp

If you have used previous versions of Macintosh Common Lisp, you should note an important change in the implementation of the file system.

Macintosh Common Lisp version 2 uses logical hosts, bringing it into compliance with the file system interface design described in Chapter 23 of *Common Lisp: The Language*. This can be somewhat confusing, since the old MCL-specific system of logical directories is very similar to the new Common Lisp system of logical hosts. Under earlier versions of Macintosh Common Lisp, "CCL" (for example) was defined as a logical directory, and you could test for the presence of a file like this:

```
? (probe-file "ccl;MCL help")
```

In Macintosh Common Lisp version 2, "CCL" is defined as a logical host, and the syntax is very slightly different:

```
? (probe-file "ccl:MCL help")
```

If your application requires it, you can reproduce the old behavior by defining the logical directory yourself:

```
? (def-logical-directory "ccl"
      (full-pathname "ccl:"))
```

- *Note:* The MCL functionality previously called "logical pathnames" refers to the MCL-specific system of logical directories and is now called "logical directory names." The logical pathname functionality discussed in this chapter refers to the file system interface design described in Chapter 23 of *Common Lisp: The Language.*

Printing and reading pathnames

Common Lisp now specifies that pathnames be printed and read using #P syntax.

In Macintosh Common Lisp, pathnames are printed using the Common Lisp #P reader macro (see *Common Lisp: The Language,* pages 537 and 556), as shown in this example:

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

Macintosh Common Lisp also has a numeric argument that specifies one of four possible unusual conditions in the pathname.

#1P means that the type is :unspecific .

#2P means that the name is "".

#3P means that the type is :unspecific and the name is "".

#4P means that the namestring represents a logical pathname.

All other numeric arguments are illegal.

With this convention, Macintosh Common Lisp avoids the potential loss of information when converting between a pathname and a namestring:

```
(make-pathname :name "foo" :type "lisp")
#P"foo.lisp"
(make-pathname :name "foo" :type nil)
#P"foo"
(make-pathname :name "foo" :type :unspecific)
#1P"foo"
(make-pathname :name nil :type "lisp")
#P".lisp"
(make-pathname :name "" :type "lisp")
#2P".lisp"
(make-pathname :name nil :type nil)
#P""
(make-pathname :name nil :type :unspecific)
#1P""
(make-pathname :name "" :type nil)
#2P""
(make-pathname :name "" :type :unspecific)
#3P""
```

- *Note:* The numeric argument *#n*P is not a part of Common Lisp and may be removed in future releases of Macintosh Common Lisp.

Pathname structure

Common Lisp pathnames (Lisp data objects of type `pathname`) have six components: `host`, `device`, `directory`, `name`, `type`, and `version`.

Macintosh physical pathnames

On a Macintosh computer, filenames have only three components: `directory`, `filename`, and an optional `type`. Macintosh filenames can be translated into Common Lisp pathname structures; when they are, the `host`, `device`, and `version` components of the pathname are `:unspecific`.

The Macintosh physical pathname syntax has the following components:

`[:] {`*directory*`:}* [`*name*`] [.`*type*`]`

A Macintosh physical pathname may have multiple colons. The component of the string preceding its first delimiter does not name a logical host.

```
? (make-pathname :directory "Style&Design:Glossary:"
                 :name "frontmatter")
#P"Style&Design:Glossary:frontmatter"
```

Common Lisp logical pathnames

Common Lisp logical pathname syntax has the following components:

`[`*host*`:] [;] {`*directory*`;}* [`*name*`] [.`*type* `[.`*version*`]]`

In logical pathname syntax, the *host* and *directory* components are indicated by the characters to the left of the last colon or semicolon. Logical pathnames can be distinguished from physical pathnames by the following tests:

- The first delimiter between components is a colon.
- The first delimiter is the only colon.
- The string preceding the first delimiter names a defined logical host.

For example, the following is a Common Lisp logical pathname because the first delimiter between pathname components is a colon, it is the only colon, and `"CCL"`, the string preceding the first delimiter, names a defined logical host:

```
"CCL:Interface Tools;My Menus;custom-menu.lisp"
```

Defining logical hosts

By defining logical hosts, Macintosh Common Lisp is able to exchange logical pathnames conveniently and portably. When a logical host is a different file system, for example, one in which the length of filenames is restricted, logical hosts and logical pathname translations provide a necessary layer of abstraction. Logical hosts are also useful when moving software from one machine to another.

Macintosh Common Lisp will recognize a logical host only after it has been defined. To define a logical host, you create and execute a `setf` form to set `logical-pathname-translations` for the relevant host. You should do this for every file system with which you will interact. Here is a very simple example:

```
? (setf (logical-pathname-translations "home")
          `(("**;*.*" ,(merge-pathnames ":**:*.*"
                        (mac-default-directory)))))
```

When Macintosh Common Lisp is run, two logical hosts are set up automatically:

- The host `"ccl"` is set to the directory holding the MCL application.
- The host `"home"` is set to the directory holding the document that was launched with Macintosh Common Lisp.

After you define a logical host, you can inspect it by clicking Inspect on the Tools menu, then clicking Logical Hosts. This displays a list of all the logical hosts used and generated by Macintosh Common Lisp.

u *Note:* For a full discussion of logical pathname namestrings and their syntax, see *Common Lisp: The Language*, pages 628–629. For information on the philosophy and use of `logical-pathname-translations`, see pages 636–637.

Ambiguities in physical and logical pathnames

In Macintosh Common Lisp, the colon is both the host delimiter in logical pathname syntax and the device / directory delimiter in physical pathname syntax. This can cause ambiguity. For example, in the namestring `"bar:foo.lisp"`, `"bar"` can be either a logical host or a top-level physical directory.

If you have both a top-level physical directory and a logical host with the same name, there is a possibility of ambiguity. For this reason it is advisable not to give a physical device and a logical host the same name.

If you have a name conflict, you should do one of the following:

- Rename one.
- Use the special escape character,#\∂ (Option-D) to quote the colon after the directory name of the physical pathname; this indicates that the pathname is physical. The escape character is documented in the section “The Pathname Escape Character” later in this chapter.
- Create the physical pathname with the function

  `(make-pathname :directory '(:absolute` *namestring*`))`

  where *namestring* is the namestring of the physical directory.

More on namestrings and pathnames

Types may be specified as part of the filename; for instance, you generally specify the type of an uncompiled file of Lisp source code by giving it the type `.lisp`, and compiled source code by giving it the type `.fasl`.

All functions that accept pathnames as arguments also accept namestrings, converting them to pathnames before using them. It is seldom necessary to use `(pathname "hd:foo")`. Instead, you can use `#"hd:foo"`. However, if the pathname is going to be parsed repeatedly, you should use the `pathname` syntax; that is, the value of `*default-pathname-defaults*` should be a pathname, not a string. (See the documentation of `*default-pathname-defaults*` in *Common Lisp: The Language*.)

The Common Lisp function `parse-namestring` converts a namestring to a pathname. The Common Lisp function `namestring` converts a pathname to a string. You can create a pathname directly by specifying its components using the Common Lisp function `make-pathname`.

Creating and testing pathnames

Common Lisp provides several functions to create pathnames and to test whether an object is a pathname. You can create a pathname directly, merge a pathname with default components, and retrieve components of a pathname.

Full documentation of most of these functions appears in Chapter 23, "File System Interface," of *Common Lisp: The Language*, and they are not redocumented here. Only the following function shows special behavior in Macintosh Common Lisp.

make-pathname [*Function*]

Syntax

```
make-pathname &key :host :device :directory :name
  :type :version :defaults :case
```

Description The Common Lisp function `make-pathname` constructs and returns a pathname. After the components specified by the `:host`, `:device`, `:directory`, `:name`, `:type`, and `:version` arguments are filled in, missing components are taken from the `:defaults` argument. The Macintosh Operating System does not support hosts, devices, or versions, so Macintosh Common Lisp recognizes only logical hosts. In Common Lisp, a logical host is a string that has been defined as a logical pathname host using `setf` and `logical-pathname-translations`. (See page 632 of *Common Lisp: The Language* for a discussion of how this is done.)

Arguments

`:host` Specifies the host component. The `:host` argument determines whether a pathname is physical or logical. If the `:host` argument is `:unspecific`, or if it is omitted and the `:defaults` argument is a physical pathname, then a physical pathname is created. Otherwise the `:host` argument must be `nil` or a string, and a logical pathname is created.

`:device` Specifies the device component. Because the Macintosh computer does not support devices, this argument is ignored and `pathname-device` always returns `:unspecific`.

`:directory` Specifies the directory component. The value of the `:directory` argument is `nil`, `:wild`, `:wild-inferiors`, *string*, or *list*.

- nil — Specifies that the directory component should be taken from the defaults.
- `:wild` — Specifies the wildcard `"*"`.
- `:wild-inferiors` — Specifies the wildcard `"**"`.

string A string, which may be a wildcard or empty, and which may end in a colon or semicolon. Unless the `:host` argument is a logical host, Macintosh Common Lisp interprets a string argument with colons or semicolons as a Macintosh-syntax directory namestring.

list A list beginning with either `:absolute` or `:relative` followed by the individual directory component strings.

`:name` Specifies the name component. The value of the `:name` argument is `nil`, `:wild`, or *string*.

`nil` Specifies that the name component should be taken from the defaults.

`:wild` The wildcard `"*"`.

string A string, which may be a wildcard or empty. Quoted colons are allowed in the `:name` component, but they cause an error when they are passed to the Macintosh File System.

`:type` Specifies the type component. Its value is `nil`, `:wild`, or *string*.

`nil` Specifies that the type component should be taken from the defaults.

`:wild` The wildcard `"*"`.

string A string, which may be a wildcard or empty. Quoted colons are allowed in the `:type` component, but they cause an error when they are passed to the Macintosh File System.

`:version` Ignored unless the `:host` argument is a logical host. For logical pathnames, the value of the `:version` argument may be `nil`, `:unspecific`, `:wild`, `:newest`, or a positive integer.

`nil` Specifies that the version component should be taken from the default.

`:unspecific` Indicates whether the version number is unspecified.

`:wild` The wildcard `"*"`.

`:newest` The newest version.

integer A positive integer representing the version number. Currently Macintosh Common Lisp allows only 0.

`:defaults`	Specifies which defaults to use. The default value of the `:defaults` argument is a pathname whose host component is the same as the host component of `*default-pathname-defaults*` and whose other components are all `nil`.
`:case`	Determines how character case is treated. The value of `:case` may be `:common` or `:local`. A full description of `:case` is given in *Common Lisp: The Language*, starting on page 617.

Full documentation of `make-pathname` is given in *Common Lisp: The Language*, on page 643.

Parsing namestrings into pathnames

The MCL pathname parser uses the following rules to break namestrings into their components.

- Unspecified components are given the value `nil`. Neither defaults nor logical directory names are merged at parse time, with the exception of the `:host` component of `*default-pathname-defaults*`. The function `merge-pathnames` merges one pathname with another by replacing `nil` components of its first argument with corresponding components of its second argument. The function `full-pathname` performs the logical-to-physical pathname translation.
- The `:directory` component is identified as the characters from the end of the host component to the last colon or semicolon. The colon is the standard Macintosh separator character for directories. The semicolon is the separator for logical directory names. A directory name that begins with a colon is relative to the Macintosh default directory.
- The `:name` component is identified as the characters that follow the directory component until either the end of the string or the beginning of the type component. The period between the name and the type component is only a separator and is not part of the `:name` component. To make a name containing a period, use the escape character (see the next section, “The Pathname Escape Character”). To specify a file that has an empty string as its name, use a single period after the directory separator character.
- The `:type` component is composed of the characters from the name component to either the version component or the end of the string.

- The `:version` component, if present, is always either `:newest` or `0`. It is the last component before the end of the string.

Table 9-1 contains some examples of namestring-to-pathname parsing.

Table 9-1 Some namestrings parsed into pathnames

		Pathname components		
Namestring	**Host**	**Directory**	**Name**	**Type**
`"hd:foo.lisp"`		`(:absolute "hd")`	`"foo"`	`"lisp"`
`"hd:"`		`(:absolute "hd")`	`nil`	`nil`
`"hd:."`		`(:absolute "hd")`	`""`	`nil`
`":foo"`		`(:relative)`	`"foo"`	`nil`
`"foo"`		`nil`	`"foo"`	`nil`
`"foo."`		`nil`	`"foo"`	`nil`
`"foo.fasl"`		`nil`	`"foo"`	`"fasl"`
`"hd:sub-dir:foo.text"`		`(:absolute "hd" "sub-dir")`	`"foo"`	`"text"`
`"sys:bar;foo.lisp"`	`"sys"`	`(:absolute "bar")`	`"foo"`	`"lisp"`

The pathname escape character

If you need to use a colon, semicolon, period, or asterisk as part of a pathname, quote it with the special escape character, #\∂ (Option-d). This escape character works very much like the backslash character in strings. Any character preceded by a ∂ loses any special meaning.

- *Note:* Asterisks must be quoted in physical pathnames, because Common Lisp mandates that functions such as `truename` and `open` must signal an error if given a wild pathname.

Table 9-2 illustrates the quoting mechanism in pathnames.

n **Table 9-2** Effect of escape characters

	Pathname components		
Namestring	**Directory**	**Name**	**Type**
`"hd:foo.lisp"`	`(:absolute "hd")`	`"foo"`	`"lisp"`
`"hd:foo∂.lisp"`	`(:absolute "hd")`	`"foo∂.lisp"`	`nil`
`":fo∂o∂."`	`(:relative)`	`"foo∂."`	`nil`
`";ccl∂;foo"`	`(:relative)`	`"ccl∂;foo"`	`nil`
`"ccl;fo∂∂o"`	`(:absolute (:logical "ccl"))`	`"fo∂∂o"`	`nil`
`"hd:fo\"o.lisp"`	`(:absolute "hd")`	`"fo\"o"`	`"lisp"`

Only the *needed* escape characters are retained (for example, the "∂" before the "o" in the third line is removed, but the "∂" before the period is retained). Of course, this mechanism is meant to work only for the MCL additions; you can specify a filename that includes a colon, but you cannot open such a file, because Macintosh computers do not accept filenames that contain colons.

u *Note:* The escape characters are not part of the true name. They are included only in the Lisp representation of the pathname, not in the Macintosh system's representation of the pathname.

The `make-pathname` function attempts to insert the appropriate escape characters in components that need them. The user need only insert escape characters in front of semicolons that are part of directory components, and in front of the character ∂. Here are some examples of the use of `make-pathname` .

```
? (make-pathname :directory "Hd:" :name "foo" :type "lisp")
#P"Hd:foo.lisp"
? (make-pathname :directory nil
                 :name "foo"
                 :type "fasl")
#P"foo.fasl"
? (make-pathname :directory nil :name "foo."
                 :type "fasl")
#P"foo∂..fasl"
? (make-pathname :directory "hd;"
                 :name "foo."
                 :type (pathname-type *.lisp-pathname*))
#P"hd;foo∂..lisp"
```

Loading files

The following functions and variables govern the loading of files. For Common Lisp functions governing the loading of files, see Section 23.4, “Loading Files,” starting on page 657 of *Common Lisp: The Language*.

.lisp-pathname [*Variable*]

Description The `*.lisp-pathname*` variable contains the file type for MCL source code files. The initial value of this variable is `#P".lisp"`.

.fasl-pathname [*Variable*]

Description The `*.fasl-pathname*` variable contains the file type for MCL compiled files. The initial value of this variable is `#P".fasl"`.

pathname-translations-pathname [*Variable*]

Description The *pathname-translations-pathname* variable contains a pathname whose host is :ccl and whose type is the string "pathname-translations" .

require [*Function*]

Syntax require *module* &optional *pathname*

Description The require function was once a Common Lisp function but is now specific to Macintosh Common Lisp. It attempts to load the files in *module* if they have not already been loaded.

Arguments

module	The name of the module.
pathname	A pathname or list of pathnames indicating the files contained in the module.

There are three ways to tell require how to look for a module:

- If *pathname* is given, it should be a pathname or a list of pathnames whose files should be loaded in order, left to right.
- If *pathname* is not given, require first looks in the variable *module-file-alist*, which is bound to an association list. In this association list, the car of each element should be a module name, and the cdr of each element should be a pathname or list of pathnames making up the module. The require function loads all the files listed. Initially, *module-file-alist* is empty. Here is how to add something to *module-file-alist* .

```
? (push '(my-system . ("my-sys;definitions.fasl"
                       "my-sys;actions.fasl"))
        *module-file-alist*)
```

- If the module is not registered in *module-file-alist* , require looks for a file with the same name as the module name in the locations specified by the variable *module-search-path* . The *module-search-path* variable should be bound to a list of pathnames, each specifying the directory and possibly a file type (the name component is ignored and replaced by the name of the module). If no file type is given, both *.lisp-pathname* and *.fasl-pathname* are looked for, and the more recent file is used.

For example, `(push "ccl:misc;" *module-search-path*)` causes `(require 'tools)` to look for the file `ccl:misc;tools.fasl` or `ccl:misc;tools.lisp`, whereas `(push "ccl:misc;.fasl" *module-search-path*)` causes `(require 'tools)` to look for `ccl:misc;tools.fasl` before searching for other versions of the tools file. The initial value of `*module-search-path*` is `(#4P"ccl:" #4P"home:" #4P"ccl:library;" #4P"ccl:examples;")`.

Macintosh Common Lisp keeps a list of files currently being loaded. This helps ensure that files requiring each other do not cause infinitely recursive calls to `require`.

For documentation of the state of `require`, see *Common Lisp: The Language*, pages 277–278.

`provide` [*Function*]

Syntax `provide` *module*

Description The `provide` function was once part of Common Lisp but is now specific to Macintosh Common Lisp. It adds a new module name to the list of modules maintained in the variable `*modules*`, indicating that the module *module* has been provided.

For documentation of the state of `provide`, see *Common Lisp: The Language*, pages 277–278.

Argument *module* The name of the module.

Macintosh default directories

The Macintosh Operating System maintains a default directory of its own. Any namestring that begins with a colon or semicolon is relative. The directory component of a relative pathname is appended to the directory component of `*default-pathname-defaults*` before accessing the file system. If the resulting pathname is still relative, then the value of `mac-default-directory` is used.

- *Note:* Desk accessories and other background processes may change the default directory without notice. If you must access the Macintosh default directory, you should set it just before accessing it, or (preferably) specify a directory explicitly in file system calls.

The Macintosh default directory is initially the directory containing Macintosh Common Lisp.

mac-default-directory [*Function*]

Syntax `mac-default-directory`

Description The function `mac-default-directory` returns the Macintosh default directory.

Example

```
? (mac-default-directory)
#P"BigTowel:CCL:"
```

set-mac-default-directory [*Function*]

Syntax `set-mac-default-directory` *pathname*

Description The function `set-mac-default-directory` sets the Macintosh default directory to the directory component of *pathname*.

If the directory component of a pathname is empty, the Macintosh computer looks for the directory in the Macintosh default directory. To ensure that the Macintosh default directory is *not* used, specify the directory component of the pathname. (One way to do this is by specifying a merge with some other default.)

The default directory returned by `mac-default-directory` can change at any time; set it explicitly just before using it, or (preferably) specify a directory explicitly in file system calls.

Argument *pathname* A pathname, string, or stream associated with a file. If the directory specified by the pathname exists, `set-mac-default-directory` sets the Macintosh default directory to the directory component of *pathname* . If it does not exist, `set-mac-default-directory` returns `nil` and the Macintosh default directory is not changed.

Example

```
? (set-mac-default-directory #P"BigTowel:CCL Test:")
#P"BigTowel:CCL Test:"
```

---

mac-namestring [*Function*]

Syntax `mac-namestring` *pathname*

Description The `mac-namestring` function translates *pathname* from a logical to a physical pathname. If *pathname* is a logical pathname or a string describing a logical pathname, it is translated to a physical pathname. If *pathname* contains MCL logical directories, they are expanded. The function returns the physical pathname as a namestring. The function then prepares *pathname* for passing to the Macintosh File Manager by verifying that the namestring contains no wildcards or quoted colons and by removing all quoting. If *pathname* contains wildcards or quoted colons, an error is signaled.

Argument *pathname* A pathname or a string.

Example

```
? (mac-namestring "ccl:examples;dialog-editor.lisp")
"hd:myccl:examples:dialog-editor.lisp"
```

mac-directory-namestring [*Function*]

Syntax `mac-directory-namestring` *pathname*

Description The function `mac-directory-namestring` turns *pathname* into a namestring, expands all logical directories into physical directories, then prepares it for passing to the Macintosh File Manager by verifying that the namestring contains no wildcards or quoted colons and by removing all quoting. If *pathname* contains wildcards or quoted colons, an error is signaled. It returns only the directory component of the pathname as a string.

Argument *pathname* A pathname, string, or stream.

mac-file-namestring [*Function*]

Syntax `mac-file-namestring` *pathname*

Description The function `mac-file-namestring` turns *pathname* into a namestring, then prepares it for passing to the Macintosh File Manager by verifying that the namestring contains no wildcards or quoted colons and by removing all quoting. If *pathname* contains wildcards or quoted colons, an error is signaled. It returns only the part of the string excluding the directory specification (that is, the filename and file type).

Argument *pathname* A pathname, string, or stream.

Structured directories

Common Lisp provides a portable format for specifying directories, discussed in *Common Lisp: The Language*, starting on page 620. Macintosh Common Lisp follows that format, with the exception that the symbols `:up` and `:back` are equivalent in the current Macintosh File System.

The following function extends the Common Lisp function `directory`.

directory [*Function*]

Syntax `directory` *pathname* `&key :directories :files :directory-pathnames :test :resolve-aliases`

Description The `directory` function takes a pathname as its argument and returns a list of pathnames, one for each file in the file system that matches the given pathname.

You can use `directory` with any of the wildcards described in the next section. When you use wildcards, this function returns a list of the true names of all matching files in all matching directories. If no files match the specified pathname, `directory` returns `nil`.

Arguments

pathname A value. If the directory specified by the pathname exists, `directory` returns a list of pathnames of files included in that directory. If it does not, `directory` returns `nil`.

`:directories` An argument specifying whether to include directories in the returned list. The default value is `nil`.

`:files` An argument specifying whether to include files in the returned list. The default value is true.

`:directory-pathnames` An argument specifying whether to represent directory pathnames in the returned list as directories or files (`foo:baz:` or `foo:baz`). The default value is true, which means that they are represented as directories.

`:test` A test function to be applied to each matching pathname. The `:test` argument is called only if all the other conditions are satisfied.

`:resolve-aliases` An argument specifying whether to resolve aliases. If the value of `:resolve-aliases` is `:show-alias`, then aliases are resolved but the pathname returned contains the name of the alias rather than the name of the target. Any other non-`nil` value causes aliases to be resolved and the pathname returned to be that of the target. The default value is `nil`.

directoryp [*Function*]

Syntax `directoryp` *pathname*

Description The `directoryp` function returns the true name of the directory if *pathname* names a directory, `nil` if it names an ordinary file; otherwise it signals an error. (For true names, see *Common Lisp: The Language* under the function `truename` .)

Argument *pathname* A pathname or string.

full-pathname [*Function*]

Syntax `full-pathname` *pathname-or-namestring* `&key :no-error`

Description The `full-pathname` function returns a pathname whose logical components are all translated into physical components. If the function is called on a namestring, the namestring is first converted into a Lisp pathname. It can translate both Common Lisp logical pathnames and MCL logical directories (see the section “Logical Directory Names” later in this chapter). The pathname is merged with `*default-pathname-defaults*` .

This function was formerly called `expand-logical-namestring` .

Arguments *pathname-or-namestring*
A pathname or namestring.

`:no-error` If the value of `:no-error` is `t` (the default) and there is no physical directory for a logical directory in *pathname* , Macintosh Common Lisp returns `nil`. If the value of `:no-error` is `nil`, Macintosh Common Lisp signals an error.

Example

This example creates a logical-to-physical mapping and gets its full pathname.

```
;Create the logical to physical mapping:
? (setf (logical-pathname-translations "misc")
        '((**;**" "hd:ccl-misc:**.*.*")))
NIL
;Load the file "hd:ccl-misc:hacks.lisp":
? (load "misc:hacks.lisp")
;Loading "hd:ccl-misc:hacks.lisp"...
#P"hd:ccl-misc:hacks.lisp"

? (full-pathname "misc:hacks.lisp")
"hd:ccl-misc:hacks.lisp"
? (full-pathname "MISC:hacks.lisp")
"hd:ccl-misc:hacks.lisp"            ;Note case insensitivity.
```

directory-pathname-p

[*Function*]

Syntax `directory-pathname-p` *pathname*

Description The `directory-pathname-p` function returns a Boolean value `t` if *pathname* is a pathname specifying a directory, `nil` if it is not. A pathname is a directory pathname if its name is `nil` or the empty string and its type is `nil` or `:unspecified` .

Argument *pathname* A pathname, string, or stream.

Example

```
? (directory-pathname-p "ccl:foo;")
T
? (directory-pathname-p "ccl:foo")
NIL
? (directory-pathname-p "hd:ccl:")
T
? (directory-pathname-p "hd:ccl:init.lisp")
NIL
```

Wildcards

Macintosh Common Lisp supports two forms of wildcards. One is **extended wildcards** as specified in *Common Lisp: The Language*, pages 623–627. Extended wildcards do not depend on a specific wildcard syntax. If you plan to port your code over multiple file systems, use the Common Lisp extended wildcards.

You can also use the simpler wildcard system described here, which is compatible with previous versions of Macintosh Common Lisp.

The wildcards are used in the following ways:

- One asterisk matches zero or more characters in a component.
- One asterisk in place of a directory component matches one directory level.
- Two asterisks used in place of a directory match zero or more subdirectories at all levels of the parent directory.
- Two asterisks used in place of the filename components match any number of components that are left.

The following examples assume the existence of a mounted disk with the name `"hd"`.

- `(directory "hd:*:" :files nil :directories t)` returns a list of all subdirectories directly under `"hd:"`.
- `(directory "hd:**")` returns a list of files under `"hd:"`.
- `(directory "**:**:" :directories t :files nil)` returns a list of all the subdirectories at all levels in all the devices known to the machine.
- `(directory "**:**")` returns a list of all the files at the top level in all the devices known to the machine.
- `(directory "hd:*.lisp")` returns a list of all the files in the top level of `"hd:"` that are of type `"lisp"`.
- `(directory "**:ccl:*:*:prin*12.**")` returns a list of all the files in any device that start with the letters `"prin"` and end in `"12"` and are two levels below a directory named `"ccl:"`.

File and directory manipulation

The functions in this section operate on both directories and files. A directory operation is performed if the filename component is empty (that is, if the pathname ends in a colon or semicolon); otherwise, a file operation is performed.

The functions operate on Lisp pathnames, strings, and streams.

`delete-file` [*Function*]

Syntax `delete-file` *pathname* `&key :if-does-not-exist`

Description This extension of the Common Lisp function `delete-file` deletes the specified *pathname* .

Arguments *pathname* A pathname.

`:if-does-not-exist`
A keyword that can take the value `nil` or `:error` . If *pathname* does not exist and the value of `if-does-not-exist` is `nil` (the default), Macintosh Common Lisp returns `nil`. If it is `:error` , Macintosh Common Lisp signals an error.

`create-file` [*Function*]

Syntax `create-file` *pathname* `&key :if-exists :mac-file-type :mac-file-creator`

Description The `create-file` function creates an empty file or a directory named *pathname* and returns the `truename` of the created file or directory. If necessary, `create-file` creates missing intermediate directories.

The `:mac-file-type` and `:mac-file-creator` keywords are case sensitive. The values of these keywords must be os-types. An os-type is a four-character string or keyword that is case sensitive.

Arguments

pathname A pathname.

`:if-exists` A keyword that determines what to do if the file already exists. If *pathname* already exists and the value of `:if-exists` is `:error` (the default), Macintosh Common Lisp signals an error. If its value is `nil`, Macintosh Common Lisp does nothing and returns `nil`. If it is `:overwrite` or `:supersede`, then Macintosh Common Lisp overwrites or replaces the previous file and returns the new file.

`:mac-file-type` The os-type of the new file. The default is `:TEXT`. Directories do not have Macintosh types.

`:mac-file-creator` The creator of the new file. The default is `:CCL2`. Directories do not have Macintosh creators.

---

open

[*Function*]

Syntax

```
open filename &key :direction :element-type
:if-exists :if-does-not-exist :external-format
:mac-file-creator :fork
```

Description

The Common Lisp function `open` opens a stream to the file specified by *filename*, which may be a string, a pathname, a logical pathname, or a stream. Two new keywords, `:mac-file-creator` and `:fork`, distinguish the MCL implementation from Common Lisp's; the keyword arguments `:direction` and `:if-exists` can each take an additional value. The additional MCL keywords and values are documented next.

Arguments

`:direction` A pathname or string. This keyword can now take the value `:shared` in addition to `:input`, `:output`, `:io`, and `:probe`. The value `:shared` is the same as `:io` except that more than one stream can be open to a file at the same time. It defaults to `:input`.

`:if-exists` The action to take when the direction is `:output` or `:io` and the file already exists. This argument can take the value `:dialog` in addition to the values `:append`, `:error`, `:new-version`, `:rename`, `:rename-and-delete`, `:overwrite`, `:supersede`, and `nil`. The default is `:error`. The values `:dialog`, `:rename`, and `:new-version` cause a dialog box to request confirmation if the file already exists.

`:external-format`
A four-character string to store as the Macintosh file type. Its value defaults to `:default`, in which case the Macintosh file type is `:TEXT`.

`:mac-file-creator`
The Macintosh file creator. It defaults to `:CCL2`.

`:fork` An argument specifying whether to open the data fork or the resource fork. It may have the value `:data` (the default) or `:resource`.

---

rename-file

[*Function*]

Syntax `rename-file` *old-pathname new-pathname* `&key :if-exists`

Description The Common Lisp function `rename-file` renames the specified *old-pathname*. The new name is the result of merging *new-pathname* with *old-pathname*. Both arguments may be a string, stream, or Lisp pathname. If *new-pathname* is an open stream associated with a file, then the stream itself and the file associated with it are affected.

If successful, the `rename-file` function returns three values. The first value is the renamed *old-pathname*. The second value is the true name of the *old-pathname* before it was renamed. The third value is the true name of the *old-pathname* after it was renamed. An error is signaled if the renaming operation is not successful.

Arguments

old-pathname The old pathname of the file or directory.

new-pathname The new pathname of the file or directory.

`:if-exists` A keyword that determines what to do if the file already exists. If *new-pathname* already exists and the value of `:if-exists` is `:error` (the default), Macintosh Common Lisp signals an error. If its value is `nil`, Macintosh Common Lisp returns `nil`. If it is `:overwrite` or `:supersede`, then Macintosh Common Lisp overwrites or replaces the previous file and returns the new file.

Example

```
? (rename-file "hd:doc:file      system  notes"
       "BigTowel:misc:renamed       notes")
#P"BigTowel:misc:renamed  file system notes"
#1P"hd:doc:file system notes"
#1P"BigTowel:misc:renamed  file system notes"
```

---

file-create-date [*Function*]
file-write-date [*Function*]
set-file-create-date [*Function*]
set-file-write-date [*Function*]

Syntax

`file-create-date` *pathname*
`file-write-date` *pathname*
`set-file-create-date` *pathname time*
`set-file-write-date` *pathname time*

Description

These functions report on or modify the creation and modification dates of files. The `file-create-date` function returns the time when the volume, directory, or file specified by *pathname* was created. The `file–write–date` function returns the time when the volume, directory, or file specified by *pathname* was last modified. The corresponding `set-` functions change these parameters.

Arguments

pathname A pathname, string, or stream.

time A time, given in the Common Lisp universal time format. (The Common Lisp universal time format is described in *Common Lisp: The Language,* on page 703.)

File operations

The following functions operate on files only. These functions, in conjunction with the `directory` function, provide the needed flexibility for operating on directories.

`copy-file` [*Function*]

Syntax `copy-file` *old-pathname new-pathname* `&key :if-exists :fork`

Description The `copy-file` function copies the file to a file corresponding to the pathname specified by merging *new-pathname* with *old-pathname* . Arguments may be either strings, Lisp pathnames, or streams. If *new-pathname* does not have a filename component, then the filename of *old-pathname* is used.

If successful, the `copy-file` function returns three values. The first value is the new pathname with the filename component filled in. The second value is the true name of the file before it was copied. The third value is the true name of the copied file. An error is signaled if the copying operation is not successful.

Arguments

old-pathname	The old pathname of the file.
new-pathname	The new pathname of the file.
`:if-exists`	If *new-pathname* already exists and the value of `:if-exists` is `:error` (the default), Macintosh Common Lisp signals an error. If its value is `nil`, Macintosh Common Lisp returns `nil`. If it is `:overwrite` or `:supersede`, then Macintosh Common Lisp overwrites or replaces the previous file and returns the new file.
`:fork`	The type of fork. This value can be `:both`, `:data`, or `:resource`. The default is `:both`.

Example

```
? (copy-file "BigTowel:misc:renamed notes"
     "BigTowel:CCL Doc:copy")
#P"BigTowel:CCL Doc:copy"
#1P"BigTowel:misc:renamed notes"
#1P"BigTowel:CCL Doc:copy"
```

lock-file [*Function*]
unlock-file [*Function*]
file-locked-p [*Function*]

Syntax

`lock-file` *pathname*
`unlock-file` *pathname*
`file-locked-p` *pathname*

Description

These functions allow you to manipulate the software lock that prevents modifications to a particular file. The `file-locked-p` function returns `nil` if the file is not locked.

If a file is locked, opening it creates a read-only buffer. You can look at the file but you cannot modify it.

Argument

pathname A pathname, string, or stream.

mac-file-type [*Function*]
mac-file-creator [*Function*]
set-mac-file-type [*Function*]
set-mac-file-creator [*Function*]

Syntax

`mac-file-type` *pathname*
`mac-file-creator` *pathname*
`set-mac-file-type` *pathname os-type*
`set-mac-file-creator` *pathname os-type*

Description

Every Macintosh file has two parameters specifying the type of the file and the application that created the file. These parameters, called *os-type* s, are specified by four-character keywords or symbols that are case sensitive.

The `mac-file-type` and `mac-file-creator` functions return keywords indicating the type and creator parameters of *pathname* .

The `set-mac-file-type` and `set-mac-file-creator` functions destructively modify the type or creator of *pathname* . The new type or creator is returned as a keyword.

Arguments

pathname A pathname, string, or stream.

os-type The parameters specifying the type of the file and the application that created it. The *os-type* parameter may be a string of four characters or a four-character keyword. Files created by Macintosh Common Lisp have the creator `:CCL2` and the type `:TEXT` or `:FASL`. The *os-type* arguments are case sensitive and may contain spaces.

open-file-streams [*Variable*]

Description The `*open-file-streams*` variable is bound to a list of all streams open to disk files. The user should not change this variable. It is updated automatically by file stream operations.

Volume operations

Volume operations take as an argument either an integer (the volume number) or a pathname or string. If the argument is a pathname or string, only the volume component (the root directory) is used. Volume numbers are unique negative integers assigned to each mounted volume. Volumes numbers change from session to session and may change if a volume is unmounted and remounted. Within these limits, volume numbers allow a program to distinguish between multiple volumes with the same name. The volume number 0 is used to specify the default volume. If a string is used to specify a volume, it must contain a colon.

Drive numbers are positive integers denoting physical devices.

The following functions signal an error if the number or pathname given as an argument does not correspond to a mounted volume.

volume-number [*Function*]

Syntax `volume-number` *volume*

Description The `volume-number` function returns the volume reference number of *volume*. If *volume* is a valid volume number, it is simply returned.

Argument *volume* An integer, pathname, or string representing a volume.

Example

See the example under `drive-name`.

`eject-disk` [*Function*]

Syntax `eject-disk` *volume*

Description The `eject-disk` function ejects *volume* if possible. It is not possible to eject hard disks. If successful, `eject-disk` returns the true name of *volume;* otherwise, it signals an error. It does not unmount the volume.

Argument *volume* A volume number, drive number, pathname, or string representing a volume.

`eject&unmount-disk` [*Function*]

Syntax `eject&unmount-disk` *volume*

Description The function `eject&unmount-disk` ejects and unmounts *volume* if possible. If successful, `eject&unmount-disk` returns the true name of *volume;* otherwise, it signals an error. It is not possible to eject hard disks.

Argument *volume* A volume number, drive number, pathname, or string representing a volume.

`disk-ejected-p` [*Function*]

Syntax `disk-ejected-p` *volume*

Description The `disk-ejected-p` function returns `t` if the volume is ejected and `nil` otherwise. It signals an error if the specified *volume* is not mounted. The `probe-file` function can be used to check whether a volume is mounted.

Argument *volume* A volume number, drive number, pathname, or string representing a volume.

hfs-volume-p [*Function*]

Syntax `hfs-volume-p` *volume*

Description The `hfs-volume-p` function returns `t` if *volume* uses the Hierarchical File System (HFS) and `nil` if it uses the Macintosh File System (MFS). Most current Macintosh computers use only HFS devices, with the exception of floppy disks.

The HFS and MFS file systems are described in *Inside Macintosh*.

Argument *volume* A pathname or string representing a volume.

flush-volume [*Function*]

Syntax `flush-volume` *volume*

Description Some file system manipulations are buffered for execution at a later time. The `flush-volume` function ensures that all buffered file manipulations to a specified volume are performed. The `flush-volume` function returns the name of the volume affected.

Argument *volume* A pathname or string representing a volume.

drive-name [*Function*]

Syntax `drive-name` *number*

Description The `drive-name` function returns the name of the drive whose drive number or volume number is *number*.

Argument *number* A fixnum. A positive number is a drive number; a negative number, a volume number.

Example

```
? (volume-number #P"Dr. Johnson:")
-1
? (volume-number -1)
-1
? (drive-name -1)
#P"Dr. Johnson:"
```

drive-number

[*Function*]

Syntax `drive-number` *pathname*

Description The `drive-number` function returns the drive number of the drive indicated by *pathname* .

Argument *pathname* A pathname or string.

User interface

The following functions let the user choose or set a pathname to a file or directory.

choose-file-dialog

[*Function*]

Syntax `choose-file-dialog &key :mac-file-type :directory :button-string`

Description The `choose-file-dialog` function displays the standard Macintosh `SFGetFile` dialog box, allowing you to select a file for reading. Unless the dialog is canceled, this function returns a pathname.

Arguments

`:mac-file-type`
An os-type parameter or list of os-type parameters. If specified, only files with the given Macintosh file type are displayed in the dialog box. Os-types are case sensitive.

`:directory` A pathname or string. Specifies the directory shown when the dialog box first appears. It defaults to the last directory shown by the Choose File dialog box or Choose New File dialog box.

`:button-string`
A string. Specifies the text that appears in the button that opens the chosen file. The default is `Open`.

---

choose-new-file-dialog

[*Function*]

Syntax

```
choose-new-file-dialog &key :directory :prompt
:button-string
```

Description

The `choose-new-file-dialog` function displays the standard Macintosh `SFPutFile` dialog box, allowing you to specify a destination file for writing. An alert dialog box requests confirmation if an existing file is chosen. Unless canceled, it returns a pathname.

Arguments

`:directory` Specifies the directory shown when the dialog box first appears. It defaults to the last directory shown by the Choose File dialog box or Choose New File dialog box.The filename component of `:directory` is used as the default filename in the editable-text item of the dialog box.

`:prompt` Specifies the text to display above the area in which the user types the filename. If supplied, `:prompt` should be a string. The default prompt is As….

`:button-string`
Specifies the text that appears in the button that opens the file. The default is Save.

choose-directory-dialog [*Function*]

Syntax `choose-directory-dialog &key :directory`

Description The function `choose-directory-dialog` displays a variation of the standard Macintosh `SfGetFile` dialog box. Unless canceled, it returns a directory pathname.

Argument `:directory` Specifies the directory shown when the dialog box first appears. It defaults to the last directory shown by the `choose-file-dialog` , `choose-new-file-dialog` , or `choose-directory-dialog` dialog box.

choose-file-default-directory [*Function*]

Syntax `choose-file-default-directory`

Description The function `choose-file-default-directory` returns the namestring of the last directory selected by the `choose-file-dialog` , `choose-new-file-dialog` , or `choose-directory-dialog` dialog box. Initially, this is the directory that is the translation of `"home:"` .

set-choose-file-default-directory [*Function*]

Syntax `set-choose-file-default-directory` *pathname*

Description The function `set-choose-file-default-directory` sets the default directory used by the `choose-file-dialog` , `choose-new-file-dialog` , or `choose-directory-dialog` dialog box to *pathname* . It returns *pathname* .

Argument *pathname* A pathname or string.

Logical directory names

If you are new to Macintosh Common Lisp, you do not need to read this section.

Previous versions of Macintosh Common Lisp provided a facility, called logical pathnames, that is now called **logical directory names.** It is not connected with the new Common Lisp logical pathname facility. You can still use logical directory names; however, they will probably go away in future releases of Macintosh Common Lisp. For your new code, you should use Common Lisp logical pathnames.

Logical directory names serve as variables in a pathname string. Their goal is to allow code with embedded pathname information to run under different directory hierarchies.

Unlike physical directories, which end with colons, logical directory names end with semicolons.

Because of the use of a semicolon as the directory delimiter in MCL logical directories, a namestring containing semicolons but no host will not parse to a Common Lisp logical pathname. However, if it is merged with a logical pathname, the result is a logical pathname.

```
? (ccl::logical-pathname-p (pathname "blotz;blitz;"))
NIL
? (ccl::logical-pathname-p
   (merge-pathnames
    (pathname "blotz;blitz;")
    (pathname "ccl:")))
T
```

The following MCL functions and variables govern logical directory names.

logical-directory-alist [*Variable*]

Description The `*logical-directory-alist*` variable contains an association list that maps between logical and physical pathnames. Its value is shown in the Inspector Central window under Logical Directory Names. (For information on the Inspector Central window, see the section “The Inspector” in Chapter 10, “Debugging and Error Handling.”)

This variable was formerly called `*logical-pathname-alist*` .

def-logical-directory [*Function*]

Syntax `def-logical-directory` *logical-directory-name physical-pathname*

Description The function `def-logical-directory` defines a new logical directory name and adds it to `*logical-directory-alist*`. It returns the new value of `*logical-directory-alist*`.

To remove a logical pathname from the environment, call `def-logical-directory` with a *physical-pathname* of `nil`.

This function was formerly called `def-logical-pathname`.

Arguments

logical-directory-name
A logical directory name.

physical-pathname
The physical pathname associated with *logical-directory-name*. It may contain logical components.

Chapter 10 **Debugging and Error Handling**

Contents

Debugging tools in Macintosh Common Lisp / 334
Compiler options / 335
Fred debugging and informational commands / 336
Debugging functions / 340
Error handling / 348
    Functions extending Common Lisp error handling / 348
Break loops and error handling / 349
    Functions and variables for break loops and error handling / 353
Stack Backtrace / 355
Single-expression stepper / 358
Tracing / 359
    MCL expressions associated with tracing / 363
Advising / 366
The Inspector / 369
    Using the Inspector / 369
    Inspector functions / 371
Other debugging macros / 372

This chapter discusses debugging tools in Macintosh Common Lisp. These tools include compiler options, Fred commands, debugging functions, error-signaling functions, functions to break or cancel operations, backtrace, facilities to step through a program, trace functions, and an advise function. In addition, any part of any MCL object can be inspected and, when appropriate, edited within the Inspector.

You should read this chapter to familiarize yourself with the debugging environment in Macintosh Common Lisp.

Debugging tools in Macintosh Common Lisp

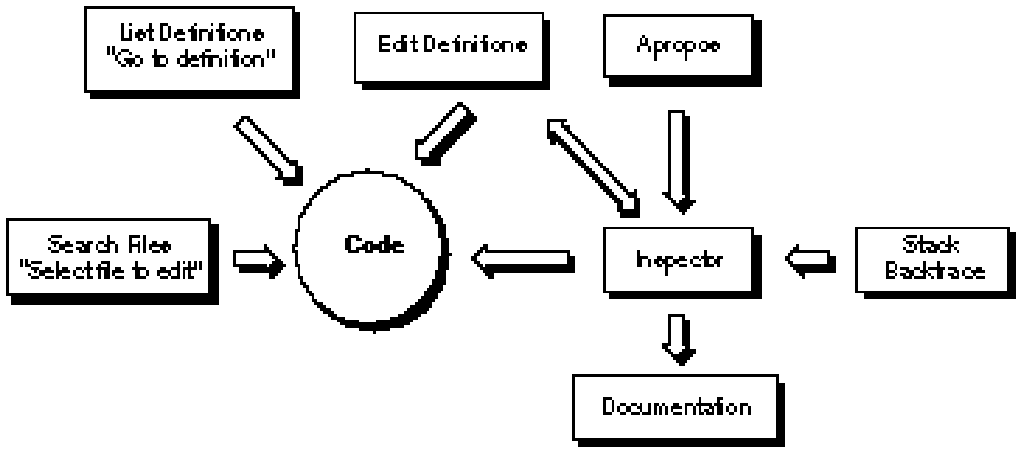
Macintosh Common Lisp provides several tools to help programmers examine and debug functions, source code, and environments:

- compiler options to retain information useful for later programming
- a set of Fred commands
- debugging functions
- a set of functions for signaling errors and aborting operations (these functions may optionally enter a break loop)
- a break-loop facility, which interrupts a program and allows you to look at the stack and examine dynamic values before returning
- a Stack Backtrace
- a single-expression stepper
- a trace function
- an Inspector

The Tools menu contains most of these tools and the Fred Commands window; the other tools are available through MCL expressions.

MCL debugging tools form an integrated whole, allowing you to look at your code from a variety of perspectives. Figure 10-1 shows the MCL debugging tools and their relationships. From each of the listed windows you can examine code in the windows they point to.

- **Figure 10-1** MCL debugging tools



Here is what the various components of Figure 10-1 do.

- The Apropos window accepts a string and finds all definitions containing that string.
- The Stack Backtrace window examines the state of the stack during a break loop.
- The Documentation window brings up documentation for Common Lisp and MCL symbols.
- The Inspector window allows you to examine all the components of any data object.
- The Edit Definitions window accepts the name of a definition and finds its source code.
- The List Definitions window lists all definitions in the current buffer and allows you to pick one for editing.
- Search Files lets you search files for the presence of a string.

When available, code is always the best documentation. Two keyboard commands are often used to examine code.

- Pressing Meta-period when the insertion point is within or next to an expression in code allows you to examine its source code. You can examine the source code of many MCL expressions.
- Pressing Control-Meta and clicking an expression acts like pressing Meta-period but also allows you to examine expressions within Inspector windows.

Compiler options

The MCL compiler can optionally retain information useful for later programming. It can also provide useful debugging information at compile time. The behavior of the compiler is regulated by the global variables listed in Table 10-1.

n **Table 10-1** Compiler options

Variable	Purpose
`*fasl-save-definitions*`	Provides a default value for the `:save-definitions` keyword argument to `compile-file` ; determines whether lambda expressions are saved in the compiled file. *Default is* `nil`; lambda expressions are not saved in the compiled file and are not available when the file is loaded. If true, lambda expressions are saved. Compiled functions without lambda expressions cannot be stepped.
`*fasl-save-doc-strings*`	Provides a default value for the `:save-doc-strings` keyword argument to `compile-file` ; determines whether documentation strings are saved in the compiled file. *Default is* `nil`; documentation strings are not saved and are not available when the file is loaded. If true, documentation strings are saved in the compiled file and are available through the Inspector and the `documentation` function (bound to the keyboard command Control-X Control-D).
`*fasl-save-local-symbols*`	Provides a default value for the `:save-local-symbols` keyword argument to `compile-file` . *Default is* `nil`; local symbols are not saved in the compiled file. If true, local symbols are saved in the compiled file and are available when the file is loaded. Generally increases `.fasl` file size by about 15–20 percent.
`*record-source-file*`	Determines whether compiler records source file of definitions. The definition contains a pointer to the source file. You can retrieve the definition by pressing Meta-period when the insertion point is next to the symbol name. *Default is true;* compiler records source file of all definitions. Meta-period retrieves source code. If `nil`, no record is kept, and Meta-period cannot retrieve source code.
`*save-definitions*`	Determines whether compiled functions can be uncompiled. *Default is* `nil`; lambda expressions are not retained; functions cannot be stepped through. If true, lambda expressions are retained; functions can be stepped through.

(continued)

n **Table 10-1** Compiler options (continued)

Variable	Purpose
`*save-doc-strings*`	Determines whether documentation strings are retained. *Default is* `nil`; documentation strings are discarded. This can save memory. If true, documentation strings are retained.
`*save-local-symbols*`	Determines whether names of arguments and local variables are saved when functions are compiled. *Default is* `nil`; information is discarded. If true, information is retained, and `*arglist-on-space*` and `backtrace` will show actual argument names.
`*warn-if-redefine*`	Helps prevent accidental redefinition of a function defined somewhere else. *Default is true;* compiler issues a warning whenever a function, macro, or variable is redefined from a new file. If `nil`, compiler does not issue warnings when user functions are redefined (but does when built-in functions are redefined).
`*warn-if-redefine-kernel*`	Helps prevent accidental redefinition of a built-in function. *Default is true;* compiler signals a continuable error whenever a built-in function is redefined. If `nil`, compiler does not signal an error when built-in functions are redefined. Use with caution.

Fred debugging and informational commands

Several Fred command keystrokes help the programmer get information about MCL expressions and the MCL environment.

Remember that you access Meta commands by pressing the Option key. You access Control commands by pressing the Control key (if your keyboard has one) or by pressing Command or Command-Shift.

Several of these commands are on the Tools menu; those menu items are listed in Table 10-2.

n **Table 10-2** Fred debugging and informational commands

Purpose	Keystroke/menu item	Effect
Display Fred commands	Control-?, Fred Commands on Tools menu	Brings up the Fred Commands window. This window contains a list of all Fred keyboard commands available in the global command table. The list is regenerated each time the window is created. The Fred Commands window may be searched, saved, and printed.
Edit definition	Meta-period, Control-Meta-click, Edit Definition on Tools menu	Attempts to bring up the source code definition for the symbol surrounding the insertion point. If the symbol is defined in more than one source file, the user is given a choice of definitions. If the symbol is defined as a slot in a `defclass`, Meta-period finds the *approximate* location of the symbol. Search backward with Control-R to find the location at which the symbol is defined. This function works for most forms that are defined with `*record-source-file*` set to `t`.
Get argument list information	Control-X Control-A	Prints the argument list of the function bound to the symbol surrounding the insertion point. Argument list is displayed in the minibuffer if the value of `*mini-buffer-help-output*` is `t`; otherwise, it is displayed in the `*standard-output*` stream. The `ed-arglist` function works for built-in functions and macros, and for most functions and macros defined with `*save-local-symbols*` or `*fasl-save-local-symbols*` set to `t`.

(continued)

n **Table 10-2** Fred debugging and informational commands (continued)

Purpose	Keystroke/menu item	Effect
Get documentation for current expression	Control-X Control-D, Documentation on Tools menu	Opens a dialog box displaying the symbol surrounding the insertion point and the documentation string of the function bound to that symbol. If no documentation string is available, displays "No documentation available." This function works for built-in functions and macros and for most forms defined with `*save-doc-strings*` set to true.
Inspect current expression	Control-X Control-I	Inspects the current symbolic expression.
Macroexpand current expression	Control-X Control-M	Macroexpands the current expression with `macroexpand` and pretty-prints the result to `*standard-output*`.
Macroexpand current expression repeatedly	Control-M	Macroexpands the current expression repeatedly with `macroexpand-1` until the result is no longer a macro call and pretty-prints the result to `*standard-output*`.
Print information about active window	Control–=	Prints information about the current Fred window to `*standard-output*`.
Read current expression	Control-X Control-R	Prints the result of reading the current symbolic expression. This is useful for tracking read-time bugs, particularly in expressions containing backquotes.

Here are some examples of using these Fred keyboard equivalents.

To perform macroexpansion with Control-X Control-M:

```
? (defmacro foo (x y)
    `(+ ,x ,y))
FOO
? (defmacro bar (z)
    `(foo ,z ,z))
BAR
? (foo 10 20)       ;Control-X Control-M
(+ 10 20)

? (bar 10)          ;Control-X Control-M
(+ 10 10)
```

To perform macroexpansion with Control-M:

```
? (foo 10 20)       ;Control-M
(+ 10 20)

? (bar 10)          ;Control-M
(foo 10 10)
(+ 10 10)
```

To read the current expression with Control-X Control-R:

```
(print `(2  ,(+ 3 4) 6))        ;<c-x c-r>
(print (cons 2 (cons (+ 3 4) '(6))))

#@(2 2)             ;<c-x c-r>
131074
```

Debugging functions

The following functions and variables are useful when programming. They provide information on the MCL programming environment and aid in testing and tracking functions.

apropos

[*Function*]

Syntax `apropos` *string-or-symbol* `&optional` *package*

Description The `apropos` function finds all interned symbols whose print names contain *string* as a substring and prints the name, function definition, and global value of each symbol. The value `nil` is returned. The result is printed to `*standard-output*` .

The `apropos` function is not case sensitive.

The functionality of `apropos` is also available through Apropos on the Tools menu. In the Apropos dialog box, you can type a symbol name or part of a symbol name. The Apropos dialog box displays a scrollable list of symbol names. Double-clicking one brings up an Inspector window for that symbol.

Arguments

string-or-symbol
Any string or symbol.

package
A package within which to search for *string-or-symbol*. When *package* is `nil`, all packages are searched.

Example

```
? (apropos 'bitmap)
BITMAP
$BITMAP.TOPLEFT,  Value: 6
$BITMAP.TOP,  Value: 6
$BITMAP.LEFT,  Value: 8
_SCRNBITMAP, Def: MACRO FUNCTION,  Value: 43059
$AFPBITMAPERR,  Value: -5004
$ICONBITMAP,  Value: 2574
:BITMAP,  Value: :BITMAP
```

- *Note:* If a symbol is given, it is interned (that is, a symbol is created and installed in the current package) and therefore the symbol always appears in the output of `apropos`. So, for example, typing `(apropos 'i-just-made-this-up)` retrieves `(i-just-made-this-up)`. This can confuse new programmers who are using `apropos` to check on the existence of a symbol. As you would expect, the Apropos dialog box does not intern strings typed into it as symbols; however, after a previously nonexistent symbol is interned with `apropos`, the Apropos dialog box will find it.

apropos-list

[*Function*]

Syntax

`apropos-list` *string-or-symbol* `&optional` *package*

Description

The `apropos-list` function returns a list of all symbols whose print names contain *string-or-symbol* as a substring.

If a symbol is given, it is interned and therefore always appears in the list returned by `apropos-list`. So, for example, typing `(apropos-list 'i-made-this-up-too)` retrieves `(i-made-this-up-too)`.

The `apropos-list` function is not case sensitive.

Arguments *string-or-symbol*
Any string or symbol.

package A package within which to search for *string-or-symbol*. When *package* is `nil`, all packages are searched.

Example

```
? (apropos-list 'bitmap)
(:BITMAP $ICONBITMAP $AFPBITMAPERR _SCRNBITMAP $BITMAP.LEFT $BITMAP.TOP
$BITMAP.TOPLEFT BITMAP)
? (setq make-syms (apropos-list 'bitmap))
(:BITMAP $ICONBITMAP $AFPBITMAPERR _SCRNBITMAP $BITMAP.LEFT $BITMAP.TOP
$BITMAP.TOPLEFT BITMAP)
? (setq make-syms (sort make-syms #'string<
                        :key #'symbol-name))
($AFPBITMAPERR $BITMAP.LEFT $BITMAP.TOP $BITMAP.TOPLEFT $ICONBITMAP
:BITMAP BITMAP _SCRNBITMAP)
? (pprint make-syms)
($AFPBITMAPERR
$BITMAP.LEFT
$BITMAP.TOP
$BITMAP.TOPLEFT
$ICONBITMAP
:BITMAP
BITMAP
_SCRNBITMAP)
```

---

arglist

[*Function*]

Syntax `arglist` *symbol* `&optional` *include-bindings use-help-file*

Description The `arglist` function returns two values, the argument list of *symbol* and how the list was computed. The second value can be one of `:definition` , `:declaration` , `:analysis` , `:unknown` , or `nil`. The value `:definition` means that `*save-definitions*` was true when the function was compiled; the value `:declaration` means that either the argument list was found in the MCL Help file or you declared the argument list with `(setf (arglist` *symbol*`)` *arglist*). The value `:analysis` means that the argument list was computed from information stored with the function; `:unknown` means that the symbol was bound to a function, but no argument list information could be determined; and `nil` means that the symbol was not bound to a function.

Arguments

symbol A symbol.

include-bindings
A value. If this value is specified and true, then the default values of optional and keyword parameters are included, if known.

use-help-file A Boolean value. If true (the default), the argument list is taken from the MCL Help file. If `nil`, the argument list is computed directly from information stored within the function. (This parameter is useful if you suspect that the MCL Help file may be incorrect.)

`documentation` [*Generic function*]

Syntax `documentation (`*x* `thing)&optional` *doc-type*

Description The generic function `documentation` returns the documentation string of *doc-type* for *x*. If *x* is a method object, a class object, a generic function object, a method combination object, or a slot-description object, *doc-type* may not be supplied, or an error is signaled. If *x* is a symbol or a list of the form `(setf` *symbol*`)`, *doc-type* must be supplied. See Table 10-3 for the documentation type that should be supplied for various MCL constructs.

Documentation strings can be changed with the Common Lisp generic function `(setf documentation)` , documented on page 842 of *Common Lisp: The Language*.

Documentation strings are retained only if the value of `*save-doc-strings*` is true when the definition occurs. If no documentation string is available, `documentation` returns `nil`.

Arguments

x A method object, class object, generic function object, method combination object, slot-description object, symbol, or list of the form `(setf` *symbol*`)`.

doc-type One of the symbols `variable`, `function`, `structure`, `type`, or `setf`.

Example

```
? (documentation 'view-draw-contents 'function)
"The event system calls this generic function whenever a view needs to
redraw any portion of its contents.  For a view, the function is applied
focused on the view; for a simple view, it is focused on the view's
container."
? (documentation 'window 'type)
"The window class, from which all window objects inherit. Windows in
turn inherit from view. All windows are streams."
```

Table 10-3 lists the values of *doc-type* that should be supplied with various MCL constructs.

Table 10-3 Constructs and their documentation types

Construct	Documentation type
Function	`function`
Generic function	`function`
Special form	`function`
Macro	`function`
Variable	`variable`
Constant	`variable`
`defstruct` structure	`structure`
Class object	`type`
Type specifier	`type`
`defsetf` definition	`setf`
`define-setf-method` definition	`setf`
Method combination	`method-combination`

edit-definition-p [*Function*]

Syntax `edit-definition-p` *name* `&optional` *type specializers qualifiers*

Description The function `edit-definition-p` returns source file information for a symbol, method, or function.

It returns five values: a list of definition types and source file names where the definition occurs (the first file in the list is the one containing the most recent definition); the name of *name;* the definition type found (one of `function`, `method`, `structure`, `class`, and so on); a list of its method qualifiers, such as `(:before)`, `(:after)`, or `(:around)`, and a list of the method specializer classes. If *name* is not the name of a method, the two last values are `nil` and `t`.

Arguments

name	A symbol, method, or function.
type	The type of definition desired. Allowable values are any data type that can have a source file: for example, `function`, `method`, `structure`, `class`, or `variable`. The default value, `t`, finds whatever exists.
specializers	A list of specializer classes for a method. Giving this argument a non-`nil` value forces the value of the *type* argument to be `'method`.
qualifiers	A list of qualifiers for a method, for example, `(:before)`, `(:after)`, or `(:around)`. The default value is `t`, which finds a method with any qualifier.

Example

```
? (edit-definition-p 'pop-up-menu)
((CLASS . "ccl:library;pop-up-menu.lisp"))
POP-UP-MENU
T
NIL
T
? (edit-definition-p 'view-draw-contents 'method '(basic-editable-text-dialog-item) :after)
NIL
VIEW-DRAW-CONTENTS
METHOD
(BASIC-EDITABLE-TEXT-DIALOG-ITEM)
:AFTER
```

help-output [*Variable*]

Description The `*help-output*` variable specifies the stream to which documentation string and argument list information is printed when accessed through Fred keyboard commands or the Inspector. This variable is initially bound to `*standard-output*` .

print-call-history [*Function*]

Syntax `print-call-history`

Description The function`print-call-history` writes a full Stack Backtrace to `*debug-io*` .

select-backtrace [*Function*]

Syntax `select-backtrace`

Description The function`select-backtrace` opens a Stack Backtrace window if it is meaningful to backtrace. If there is no context for backtracing, the function signals an error.

room [*Function*]

Syntax `room &optional` *detailed-p*

Description The `room` function prints information on the amount of space available in the Lisp operating system.

Argument *detailed-p* A value indicating how much information to print. If this value is `nil`—the default—minimal information is printed. If it is non`nil`, more detailed information is printed.

Example

```
? (room)
There are at least 1356752 bytes of available RAM.

                  Total Size            Free              Used
Mac Heap:       540576 (527K)       132600 (129K)       407976 (399K)
Lisp Heap:     2097152 (2048K)     1224152 (1195K)      849016 (829K)
 (Static):      458752 (448K)
Stacks:         218100 (212K)
? (room t)
There are at least 1344548 bytes of available RAM.

                  Total Size            Free              Used
Mac Heap:       540576 (527K)       132604 (129K)       407972 (399K)
Lisp Heap:     2097152 (2048K)     1211944 (1183K)      860736 (840K)
 (Static):      458752 (448K)
Stacks:         218100 (212K)
Markable objects:      777112 (758K) dynamic, 212776 (207K) static.
Immediate objects:      83624 (81K) dynamic, 242992 (237K) static.
```

inspect [*Function*]

Syntax `inspect` *thing*

Description The `inspect` function inspects *thing*.

Macintosh Common Lisp supports the Common Lisp `inspect` function with a window-based Inspector. In addition to calling the `inspect` function, there are two other ways of invoking the Inspector directly: choosing Inspect from the Tools menu, or giving the keyboard equivalent, Control-X Control-I. In addition, double-clicking a symbol name from the Apropos dialog box, or choosing a symbol and clicking the Inspect button, invokes the Inspector on that symbol.

Argument *thing* Any Lisp data object.

Example

```
? (inspect 'windows)
#<INSPECTOR-WINDOW "WINDOWS" #x467281>
```

top-inspect-form [*Function*]

Syntax `top-inspect-form`

Description The `top-inspect-form` function returns the form being inspected by the active Inspector window.

Example

```
? (top-inspect-form)
WINDOWS
```

For full details on the Inspector, see the section “The Inspector” later in this chapter.

Error handling

Macintosh Common Lisp uses the Common Lisp condition system, which reconceptualizes and adds to Common Lisp’s previous error-detection and error handling capabilities.

A *condition* is an interesting situation that has been detected and announced within a program. An *error* is a condition from which the program cannot continue normally, but requires some sort of intervention, either by program control or from the user.

Most MCL error-handling functions now follow the definitions of those functions given in *Common Lisp: The Language*, Chapter 24, “Errors,” and Chapter 29, “Conditions.” (Note that pages 886–887 of *Common Lisp: The Language* supersede the earlier discussion of `error` and `cerror` in Chapter 24 of the same book.) MCL extensions to those functions are described next.

Functions extending Common Lisp error handling

The following functions extend the Common Lisp condition system.

abort-break [*Function*]

Syntax `abort-break`

Description If the current read loop is waiting for input, the Common Lisp function `abort` calls the non–Common Lisp function `abort-break`, which decrements the abort level by 1. If there is input in the current read loop, the Common Lisp function `abort` calls the `abort` restart.

`cancel` [*Function*]

Syntax `cancel`

Description The `cancel` function throws to the nearest *catch-cancel*. (See the section "Simple Turnkey Dialog Boxes" in Chapter 6, "Dialog Items and Dialogs." This function is generally called when the user clicks Cancel in a modal dialog box.

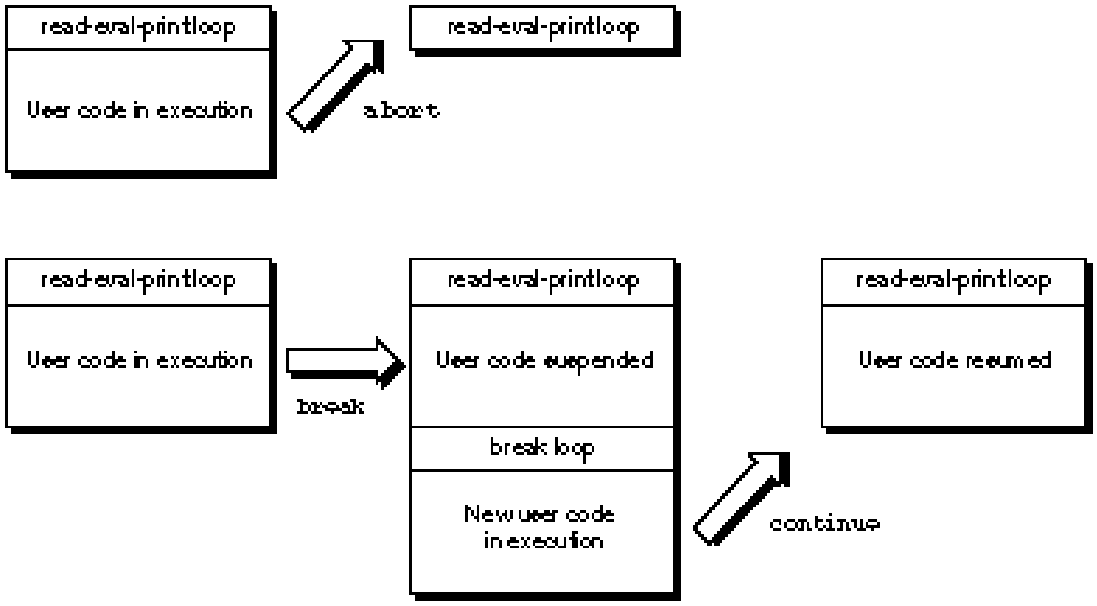
For information on the syntax of Common Lisp `throw` and `catch`, see *Common Lisp: The Language,* in particular page 192.

Break loops and error handling

At any point during an MCL program, program execution may be suspended and control passed to a break loop. A break loop behaves like the top-level read-eval print loop. However, when you enter a break loop you do not exit your program and return control to the top level (as `abort` does). Instead, a break loop suspends your program and allows interaction on top of your program. From a break loop, you can resume the program or return to the top level.

Figure 10-2 shows the execution stack of Macintosh Common Lisp. Newer items are added to the bottom. The diagrams show that break loops add new areas to the stack, but `abort` and `continue` remove areas from the stack. New items are added to the bottom.

n **Figure 10-2** Effects on the stack of `break`, `abort`, and `continue`



Within a break loop, the MCL question mark prompt is replaced by a number and an angle bracket. Expressions can be evaluated, just as they are in the normal Listener loop. Because the break loop runs on top of the interrupted program, all global variables have the values they had when the interrupted program was suspended, as the following code shows.

```
? *print-case*
:downcase
? *load-verbose*
t
? (defun show-specials ()
        (let ((*print-case* :upcase)
              (*load-verbose* nil))
          (break)
          (print "Now we have continued.")
          t))
show-specials
? (show-specials)
>Break:
> While executing: SHOW-SPECIALS
> Type Command-/ to continue, Command-. to abort.
> If continued: Return from BREAK.
```

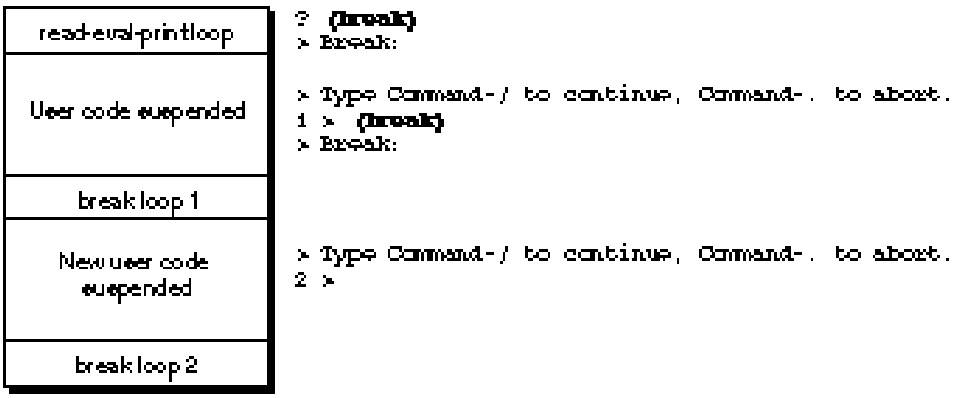
```
See the Restarts… menu item for further choices.
1 > *print-case*
:UPCASE
1 > *load-verbose*
NIL
1 > (continue)
Continuing...
"Now we have continued."
t
? *print-case*
:downcase
```

Break loops retain the dynamic environment of the interrupted program (that is, the values of global variables), but they do not retain the lexical environment of the interrupted program. For this reason, forms that you type into the break loop do not have access to the lexical variables of the interrupted program, as shown in the following code. (You can look at the lexical variables with the Stack Backtrace, described later in this chapter.)

```
? (defun double (num)
    (unless (numberp num)
      (break))
    (+ num num))
DOUBLE
? (double 5)
10
? (double 'ten)
>Break:
> While executing: DOUBLE
> Type Command-/ to continue, Command-. to abort.
> If continued: Return from BREAK.
See the Restarts… menu item for further choices.
1 > num
> Error: Unbound variable: NUM
> While executing: SYMBOL-VALUE
> Type Command-/ to continue, Command-. to abort.
> If continued: Retry getting the value of NUM.
See the Restarts… menu item for further choices.
2 > (abort-break)
Aborted
1 > (abort-break)
Aborted
```

Break loops may be nested; that is, you can enter a break loop from a break loop, and so on. The current level is indicated by the number in the Listener prompt (see Figure 10-3).

n **Figure 10-3** Nesting of break loops



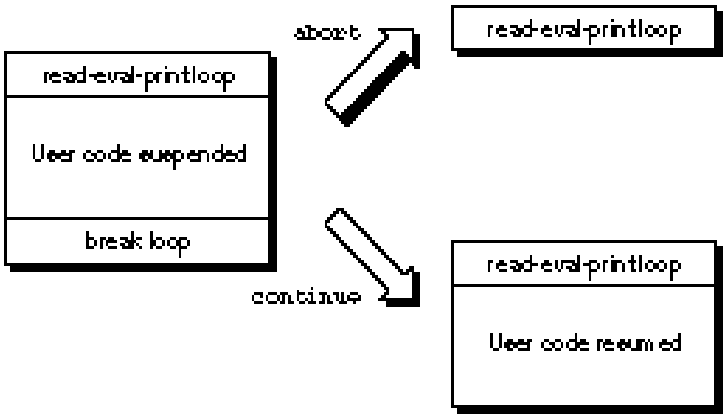
You can enter a break loop explicitly by calling the function `break` or `cerror`. In addition, if the value of `*break-on-errors*` is true, Macintosh Common Lisp enters a break loop whenever an error is signaled. If the value of `*break-on-warnings*` is true, Macintosh Common Lisp enters a break loop whenever `warn` is called. These functions and variables are described in the section "Functions and Variables for Break Loops and Error Handling" later in this chapter.

Break is also available as a command on the Eval menu.

There are two ways to leave a break loop: by calling `continue` and by calling `abort` (see Figure 10-4). Calling `continue` resumes the program from the point at which it was interrupted. Calling `abort` returns to the previous read loop. This may be the top-level loop or a prior break loop. In the case of `abort`, the suspended program is not resumed.

Abort and Continue are available as commands on the Eval menu. You can also invoke `abort` at any point by pressing Command-period. Within a break loop, you can invoke `continue` by pressing Command-slash (Command-/).

n **Figure 10-4** Two ways to leave a break loop



Functions and variables for break loops and error handling

The following functions and variables control break loops and error handling.

`break` [*Function*]

Syntax `break &optional` *format-string* `&rest` *arguments*

Description The `break` function prints the message specified by *format-string* and *arguments* and enters a break loop. It returns `nil` when continued.

The `break` function can also be invoked through the Eval menu. This provides a convenient method for suspending a program at any point of execution.

If `break` is called during the dynamic extent of a call to `without-interrupts`, no action is taken.

Arguments

format-string	A `format` control string used to construct the break message.
arguments	Zero or more `format` arguments used to construct the break message.

continue

[*Function*]

Syntax `continue &optional` *condition*

Description The `continue` function resumes execution of the code suspended by the most recent call to `break` or `cerror`. If there have been no calls to `break` or `cerror`, `continue` simply returns to the top level. If `condition` is present, the restart for *condition* is invoked.

Argument *condition* A condition.

break-on-errors

[*Variable*]

Description The `*break-on-errors*` variable determines whether Macintosh Common Lisp enters a break loop when an error is signaled. The default value is true.

If the value of this variable is true, then Macintosh Common Lisp enters a break loop when an error is signaled.

If the value of this variable is `nil`, *then* errors simply cause a return to the read-eval-print loop.

break-on-warnings

[*Variable*]

Description The `*break-on-warnings*` variable determines whether Macintosh Common Lisp enters a break loop when a warning is issued. The default value is `nil`.

If the value of this variable is true, then Macintosh Common Lisp enters a break loop when a warning is issued.

If the value of this variable is `nil`, *then* warnings do not interrupt program flow.

backtrace-on-break

[*Variable*]

Description The `*backtrace-on-break*` variable determines whether Macintosh Common Lisp displays the Stack Backtrace whenever it enters a break loop. The default value is `nil`.

If the value of this variable is true, then Macintosh Common Lisp displays the Stack Backtrace window.

If the value of this variable is `nil`, *then* you must choose Backtrace from the Tools menu to see the Stack Backtrace dialog box.

Stack Backtrace

Beyond `print-call-history` , which prints a backtrace to `*debug-io*` , Macintosh Common Lisp provides a Stack Backtrace dialog box.

When inside a break loop, the Stack Backtrace command lets you examine the state of the suspended program. To see the Stack Backtrace dialog box, choose Backtrace from the Tools menu when you are in a break loop.

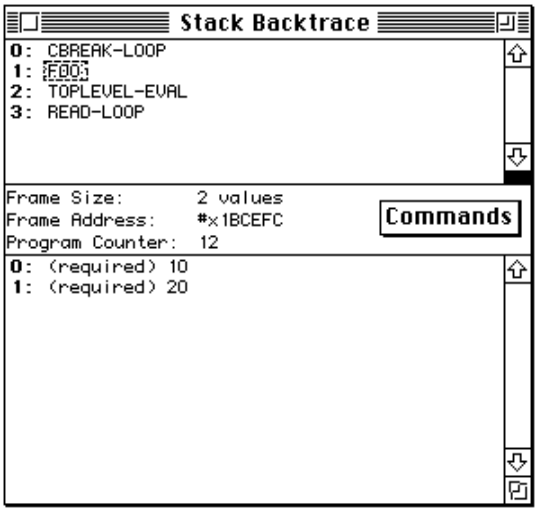
The Stack Backtrace shows all the functions awaiting return values as well as the local variables of these functions (if the functions were compiled with `*save-local-symbols*` set to true). You can easily access and set the values in a stack frame. Finally, information on the program counter and stack frame address is given.

The Stack Backtrace dialog box shows two tables. The upper table shows the functions that are pending on the stack. The lower table is initially blank. When you single-click any function in the upper table, the lower table displays that function's stack frames. You can edit values in the lower table (but do so with caution).

In the space between the tables are three pieces of information about the frame: the number of values in the frame, the memory address of the frame, and the program counter within the function where execution has been suspended. The memory address is useful for low-level system debugging. You can use the program counter with `disassemble` to locate the point of a break within a function.

Figure 10-5 and the code that follows it show editing in the Stack Backtrace dialog box.

n **Figure 10-5** A Stack Backtrace dialog box



Here is the code that produced the Stack Backtrace dialog box in Figure 10-5.

```
? (defun foo (x y)
   (let ((z 10))
     (break)
     (+ x y z)))
foo
? (foo 10 20)
> Break:
> Type Command-/ to continue,
Command-. to abort.
```

Single-click `foo`, then edit within the Stack Backtrace window:

```
1 > (local 1)
20
1 > (setf (local 1) 50)
50
1 > (continue)
Continuing...
70
```

This only works if the value of `*compile-definitions*` was true when `foo` was compiled. Otherwise, the result is still 40.

Double-clicking a function in the top table causes the function object to be inspected, giving you access to `edit-definition`, `documentation`, `arglist`, `disassemble`, and `uncompile-function`.

Because Macintosh Common Lisp supports tail recursion, any function that makes a tail-recursive call will not appear in the backtrace. To ease debugging, you can disable tail recursion with compiler declarations.

The stack frame in the lower table shows the names of local variables, if these were retained at compile time. If these were not retained, the parameters are listed as required, optional, keyword, or rest. You can use the `local` macro to access the values of these frames. In addition, you can double-click a value to inspect it.

`local` [*Macro*]

Syntax `local` *indicator*

Description The macro `local` returns the value in the current stack frame of the slot given by *indicator* . This macro can be used only when a Stack Backtrace dialog box is visible and when a frame is selected.

Argument

indicator — A symbol or number indicating a slot in the stack frame. A symbol can be used if the frame includes local symbol names and if the symbol is unique in the frame. Otherwise, a number giving the position in the frame should be used.

`set-local` [*Macro*]

Syntax `set-local` *indicator new-value*

Description The `set-local` macro changes the value at the specified *indicator* to *new-value* .

You can use `set-local` (or `local` with `setf`) to modify a value in a stack frame. Modify these values with caution, however, because the compiler may have made assumptions based on the initial values.

Arguments

indicator — A symbol or number indicating a slot in the stack frame. A symbol can be used if the frame includes local symbol names and if the symbol is unique in the frame. Otherwise, a number giving the position in the frame should be used.

new-value — The new value of the indicator.

Single-expression stepper

The single-expression stepper allows you to examine a single form, expression by expression.

The `step` macro can be used on compiled functions only if their uncompiled definitions have been retained. If there is no uncompiled definition for a function, it is treated as an atomic unit as it is evaluated. A compiled function call is executed as a whole rather than being evaluated form by form. (This is how the `step` macro treats built-in functions.)

Function definitions are retained if the function is compiled with the `*save-definitions*` variable set to `t` or if a file is compiled with the `*fasl-save-definitions*` variable set to `t`. If the function was compiled with `*save-definitions*` set to `nil`, it must be recompiled or reloaded with `*compile-definitions*` set to `nil` before it can be evaluated.

Because evaluation occurs in a null lexical environment, `step` is usually called only from the top level. If it is called from within a function, it does not have access to the local environment in which it was called. However, internal stepping can be invoked through the `trace` macro, described in the section "Tracing" later in this chapter.

It is not generally possible to step through code that requires the use of `without-interrupts` or code that uses the Macintosh graphics interface.

step [*Macro*]

Syntax `step` *form*

Description The `step` macro evaluates *form* expression by expression, under user control.

Argument *form* Any Lisp form.

step-print-level [*Variable*]

step-print-length [*Variable*]

Description The `*step-print-level*` and `*step-print-length*` variables are used to set the values of `*print-level*` and `*print-length*` during step evaluation.

Tracing

Tracing is useful when you want to find out why a function behaves in an unexpected manner, perhaps because incorrect arguments are being passed.

Tracing causes actions to be taken when a function is called and when it returns. The default tracing actions print the function name and arguments when the function is called and print the values returned when the function returns.

Other actions can be specified. These include entering a break loop when a function is entered or exited, or stepping the function. Trace actions may be conditional.

Several functions can be traced at one time.

When a traced function is traced again, the new trace actions replace the former ones. When a traced function is redefined by evaluation in a buffer, the trace actions are transferred from the old definition to the new definition. When a traced function is redefined while loading a file, the function is untraced and a warning is issued.

Macros and special forms cannot be traced. Functions that are compiled inline cannot be traced (see *Common Lisp: The Language*, pages 229–230). Note that, by default, self-recursive calls are compiled as inline branches. To effectively trace a function with self-recursive calls, you should declare it not inline.

trace

[*Macro*]

Syntax `trace {`*spec* `| (`*spec* `{`*option modifier* `})}`

Description The `trace` macro encapsulates the function or method specified by each *spec* , causing the actions specified by the options. When no options are specified, the default actions print arguments on entry and values on exit.

Invoking `(trace)` without arguments returns a list of the functions currently being traced. If no functions are currently being traced, `(trace)` returns `nil`.

Arguments

spec The specification of the function to be traced. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol* `)`, or a specific method of a generic function in the form `(:method` *symbol* `{`*qualifiers* `}` `(`*specializer* `{`*specializer* `}*))`.

option An option that specifies an action to be performed. The following options and their modifiers are supported:

`:before` Specifies the action to be taken before the traced function is called. The `:before` keyword must be followed by a modifier:

`:print` Prints the name and arguments to the function before the function is called.

`:break` Prints the name and arguments to the function and enters a break loop before the function is called. You can examine the Stack Backtrace, perform operations in the Listener, and continue if desired.

lisp-function

If the `:before` option is a function, it is called before the traced function is called. The arguments to *lisp-function* are the name of the traced function and the arguments passed to the traced function.

`:after` Specifies the action to be taken after the traced function returns. This keyword must be followed by a modifier, which should be one of the following:

`:print` Prints the name of the function and the returned values.

`:break` Prints the name of the function and the returned values, and enters a break loop. You can examine the Stack Backtrace, perform operations in the Listener, and continue if desired.

lisp-function

If the `:after` option is a function, it is called after the traced function returns. The arguments are the name of the traced function and the values returned by the traced function.

`:step` Specifies whether the traced function should be stepped when it is run. For this option to be effective, the function needs to have been compiled with the variable `*save-definitions*` set to `t` or loaded with `*compile-definitions*` set to `nil`. In addition, stepping will not work if the most recent definition comes from a `.fasl` file unless the file was compiled with `*fasl-save-definitions*` set to true.

The `:step` keyword must be followed by a modifier, which is either `t` or a function whose arguments are the name of the traced function and the arguments passed to the traced function. If it is `t` or if the function returns non-`nil`, then the traced function is stepped; otherwise it is run without stepping.

Examples

Here is an example of tracing the function `fact`.

```
? (defun fact (num)
          (declare (notinline fact))
          (if (= num 0)
            1
            (* num (fact (- num 1)))))
FACT
? (trace fact)
NIL
? (fact 5)
 Calling (FACT 5)
  Calling (FACT 4)
   Calling (FACT 3)
    Calling (FACT 2)
     Calling (FACT 1)
      Calling (FACT 0)
```

```
      FACT returned 1
     FACT returned 1
    FACT returned 2
   FACT returned 6
  FACT returned 24
 FACT returned 120

120
```

Here are some examples of the syntax of `trace` and their results. This prints before but not after.

```
? (trace (fact :before :print))
? (fact 5)
 Calling (FACT 5)
  Calling (FACT 4)
   Calling (FACT 3)
    Calling (FACT 2)
     Calling (FACT 1)
      Calling (FACT 0)
120
```

This example breaks before and prints after.

```
? (trace (fact :before :break
               :after :print))
```

This example breaks on entry with an odd argument.

```
? (trace (fact :before
               #'(lambda (func &rest args)
               "only break if number is odd"
               (if (evenp (car args))
                 (format t "~&Calling ~s~%"
                         (cons func args))
                 (break "on calling ~s"
                         (cons func args))))))
```

This example breaks before an instance of the class `foo` is initialized.

```
? (trace ((:method initialize (foo)) :before :break))
```

This example steps through the function.

```
? (trace (fact :step t))
```

This example steps through even invocations of the function.

```
? (trace (fact :step
             (lambda (name &rest args)
             (declare (ignore name))
             (evenp (car args)))))
```

MCL expressions associated with tracing

The following Lisp functions and variables are associated with tracing.

`untrace` [*Macro*]

Syntax `untrace` {*spec*}

Description The `untrace` macro stops each *spec* from being traced. Notices will not be printed when the function enters or returns. The macro returns a list of the functions that are no longer being traced.

If no *spec*s are specified, all traced functions are untraced.

If you untrace a function that wasn't traced in the first place, no action is taken.

Argument

spec The specification of the function to be untraced. This is either a symbol that is the name of a function or generic function, or an expression of the form (`setf` *symbol*), or a specific method of a generic function in the form (`:method` *symbol* {*qualifiers*} (*specializer* {*specializer*})).

trace-print-level [*Variable*]

trace-print-length [*Variable*]

Description The *trace-print-level* and *trace-print-length* variables are used to set the values of *print-level* and *print-length* during trace operations.

trace-level [*Variable*]

Description The *trace-level* variable specifies the depth of calls to the traced function. Each time the traced function is called, this number is incremented. Each time the traced function returns, it is decremented.

Example

This example begins stepping fact after the first five calls.

```
? (trace (fact :step
               (lambda (number &rest args)
                 (declare (ignore number args))
                 (> *trace-level* 5))))
FACT
```

trace-max-indent [*Variable*]

Description The *trace-max-indent* variable specifies the maximum number of spaces to indent trace output. (Normally, trace output is indented one space for each level of nesting.) The default value is 40.

trace-tab [*Function*]

Syntax trace-tab

Description The trace-tab function outputs the appropriate number of spaces and vertical bars in *trace-output* , given the current value of *trace-level* .

trace-bar-frequency [*Variable*]

Description The `*trace-bar-frequency*` variable determines whether and how often vertical bars are printed in trace output. If the value of `*trace-bar-frequency*` is `nil` (the default value), no vertical bars are printed.

Example

```
? (trace fact)
nil
? (setq *trace-bar-frequency* 2)
2
? (fact 5)
Calling (fact 5)
|Calling (fact 4)
| Calling (fact 3)
| |Calling (fact 2)
| | Calling (fact 1)
| | |Calling (fact 0)
| | |fact returned 1
| | fact returned 1
| |fact returned 2
| fact returned 6
|fact returned 24
fact returned 120
120
? (setq *trace-bar-frequency* nil)
nil
? (fact 3)
Calling (fact 3)
 Calling (fact 2)
  Calling (fact 1)
   Calling (fact 0)
   fact returned 1
  fact returned 1
 fact returned 2
fact returned 6
6
```

Advising

The `advise` macro can be thought of as a more general version of `trace`. It allows code that you specify to run before, after, or around a given function, for the purpose of changing the behavior of the function. Each piece of added code is called a piece of advice. Each piece of advice has a unique name, so that you can have multiple pieces of advice on the same function, including multiple `:before`, `:after`, and `:around` pieces of advice.

The unique `:name` and the `:when` keyword serve to identify the piece of advice. A later call to `advise` with the same values for the `:name` and `:when` keywords replaces the existing piece of advice, but a call with different values does not.

`advise` [*Macro*]

Syntax `advise` *spec form* `&key` *when name define-if-undefined*

Description The `advise` macro adds a piece of advice to the function or method specified by *spec* according to *form*.

Arguments

spec The specification of the function on which to put the advice. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* {*qualifiers*} `(`*specializer* {*specializer*}`))`.

form A form to execute before, after, or around the advised function. The *form* can refer to the variable *arglist* that is bound to the arguments with which the advised function was called. You can exit from *form* with `(return)`.

name A unique name that identifies the piece of advice.

when An argument that specifies when the piece of advice is run. There are three allowable values. The default is `:before`, which specifies that *form* is executed before the advised function is called. Other possible values are `:after`, which specifies that *form* is executed after the advised function is called, and `:around`, which specifies that *form* is executed around the call to the advised function. You should use `(:do-it)` in *form* to indicate invocation of the original definition.

define-if-undefined
An argument that determines whether to define the advised function if it is undefined. The default is `nil`, in which case an error is signaled if the function is undefined.

Examples

Here are some examples of the use of `advise`.

The function `foo`, already defined, does something with a list of numbers. The following code uses a piece of advice to make `foo` return zero if any of its arguments is not a number. Using `:around` advice, you can do the following:

```
(advise foo (if (some #'(lambda (n)
                          (not (numberp n)))
                      arglist)
                0
                (:do-it))
        :when :around :name :zero-if-not-nums)
```

To do the same thing using a `:before` piece of advice:

```
(advise foo (if (some #'(lambda (n)
                          (not (numberp n)))
                      arglist)
                (return 0))
        :when :before :name :zero-if-not-nums)
```

unadvise

[*Macro*]

Syntax `unadvise` *spec* `&key` *when name*

Description The `unadvise` macro removes the piece or pieces of advice for everything matching *spec*, *when* , and *name*. When the value of *spec* is `t` and the values of *when* and *name* are `nil`, `unadvise` removes every piece of advice; when *spec* is `t`, *when* is `nil`, and *name* is non-`nil`, `unadvise` removes all pieces of advice with the given name.

Arguments

spec — The specification of the function for which pieces of advice are to be removed. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* {*qualifiers* `}` `(`*specializer* {*specializer* `}))`.

when — The specification of the when value for the piece of advice to be removed. The allowable values are the same as those for `advise`.

name — The unique name of the piece of advice to be removed.

advisedp

[*Macro*]

Syntax `advisedp` *spec* `&key` *when name*

Description The `advisedp` macro returns a list of existing pieces of advice that match *spec*, *when* , and *name*. When the value of *spec* is `t` and the values of *when* and *name* are `nil`, `advisedp` returns all existing pieces of advice.

Arguments

spec — The specification of the function to check for pieces of advice. This is either a symbol that is the name of a function or generic function, or an expression of the form `(setf` *symbol*`)`, or a specific method of a generic function in the form `(:method` *symbol* {*qualifiers* `}` `(`*specializer* {*specializer* `}))`.

when — The specification of the when value for the piece of advice to be removed. The allowable values are the same as those for `advise`.

name — A unique name that identifies the piece of advice.

The Inspector

Macintosh Common Lisp supports the Common Lisp `inspect` function with a window-based Inspector.

The Inspector lets you look quickly at any component of one or more data objects. For instance, you can use it to look at the current state of the system data. Double-click any form or component of a form in an Inspector window to bring up a window with a definition of the form or component; double-click any item in that window to bring up its definition, and so on.

Because objects are editable in Inspector windows, you can change the state of system data and other components on the fly. You should be careful about doing so, however; it is generally safe to change the value of a global variable in the Inspector, but you should use the standard interface functions to change the values associated with object keywords.

To see the Inspector, choose Inspect from the Tools menu.You can also call `inspect` on a Lisp object or use the keystroke command Control-X Control-I. If you have an extended keyboard, you can also press the Help key. When you choose Apropos from the Tools menu and double-click a symbol name, Macintosh Common Lisp creates an Inspector window containing information about that symbol.

Using the Inspector

When you choose the Inspector from the Tools menu, you see the Inspector Central dialog box, which contains a number of options. Clicking any of these options calls up an Inspector window. Double-click any item in a window to inspect it.

The options are listed in Table 10-4.

n **Table 10-4** Options in Inspector Central

Inspector option	Effect
Help	Displays a window giving brief help on Inspector commands.
Inspector History	Lists all Lisp objects that have been inspected. Double-clicking one of them creates an Inspector window showing its definition. To begin keeping a history, evaluate the form shown in the initial window.
Disk Devices	Displays a window listing the names of all currently active devices.
Logical Directory Names	Displays a window listing all logical directory names and their physical equivalents. (These are in Macintosh Common Lisp for backward compatibility only. See Chapter 9, "File System Interface.")
Logical Hosts	Displays an Inspector window listing all logical hosts and their physical equivalents.
`(windows)`	Displays an Inspector window listing the names of all MCL windows returned by `(windows)`.
`(front-window)`	Displays an Inspector window that inspects the window returned by `(front-window)`.
`*features*`	Displays an Inspector window that inspects the list that is the value of the Common Lisp variable `*features*`.
`(list-all-packages)`	Displays an Inspector window that inspects the list returned by the Common Lisp function `(list-all-packages)`.
`*package*`	Displays a window that inspects the value of the Common Lisp variable `*package*`.
`*readtable*`	Displays a window that inspects the value of the Common Lisp variable `*readtable*`.
Record Types	Displays a window that lists all record types.
Record Field Types	Displays a window that lists all record field types.

Inspector functions

The following functions are used with the Inspector.

inspect [*Function*]

Syntax `inspect` *thing*

Description The `inspect` function inspects *thing*.

Argument *thing* Any Lisp data object.

Example

```
? (defun foo (x y)
    (let ((z 10))
      (break)
      (+ x y z)))
FOO
? (inspect 'foo)
#<INSPECT-DIALOG "Symbol: FOO" #x5DE9F9>
```

top-inspect-form [*Function*]

Syntax `top-inspect-form`

Description The `top-inspect-form` function returns the form being inspected by the active Inspector window, or `nil` if there are no active Inspector windows.

inspector-disassembly [*Variable*]

Description The `*inspector-disassembly*` variable specifies whether the Inspector displays a disassembly when you inspect a function.

If the value of this variable is true, the Inspector displays a disassembly.

If the value of this variable is `nil` (the default), no disassembly is displayed.

@ [*Variable*]

Description The @ variable is bound to the last object that was cut or copied. It is used primarily to communicate values between an Inspector window and the Listener.

Other debugging macros

The following macros are useful for testing and optimizing code and for tracing program flow.

time [*Macro*]

Syntax `time` *form*

Description The `time` macro executes *form*, prints the duration of execution (with a special note on garbage collection time, if any), and returns the value returned by *form*. The `time` macro is useful for testing and optimizing code.

Argument *form* Any Lisp form. The form should not be quoted.

Example

```
? (defun make-numlist (positive-number &aux result)
    "returns a list of numbers between 0 and
     positive-number - 1"
    (dotimes (x positive-number)
      (setq result (append result (list x))))
```

```
       ;APPEND is inefficient here.
    result)
MAKE-NUMLIST

? (time (make-numlist 100))
(MAKE-NUMLIST 100) took 449 ticks (7.483 seconds) to run.
Of that, 444 ticks (7.400 seconds) was spent in GC.
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
99)
? (defun make-faster-numlist (positive-number &aux result)
    "returns the same list more quickly"
    (dotimes (x positive-number)
      (setq result (cons x result)))
                ;This is more efficient.
      (nreverse result))
MAKE-FASTER-NUMLIST
? (time (make-faster-numlist 100))
(make-faster-numlist 100) took 0 ticks (0.000 seconds) to run.
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
99)
```

print-db

[*Macro*]

Syntax print-db { *form* }*

Description The print-db macro is equivalent to progn, except that each form is printed as it is evaluated. The *form* itself and the result of evaluating *form* are both printed (unless *form* is a string, in which case it is printed only once). The value of the last *form* is returned.

If multiple forms are given, they are printed on separate lines. Printed output is sent to *error-output*, which makes the Listener the active window before printing. Like progn, print-db returns the value of the last form.

The print-db macro is useful for tracing program flow and for checking the values of variables at various points in a program.

Argument *form* Any Lisp form.

Chapter 11 **Events**

Contents

Implementation of events in Macintosh Common Lisp / 376
How an event is handled / 376
MCL built-in event handlers / 377
Functions for redrawing windows / 387
Event information functions / 391
The event management system / 393
The cursor and the event system / 401
Event handlers for the Macintosh Clipboard / 405
MCL expressions relating to scrap handlers and scrap types / 406

This chapter explains how Macintosh Common Lisp processes events. It describes built-in handlers and functions that give event-related information. It discusses the MCL event management architecture. Finally, it describes two processes involved in event management: updating the cursor and accessing the Macintosh Clipboard.

You should read this chapter to understand or program events and event handlers in Macintosh Common Lisp.

If you are creating handlers for Apple events, you should read this chapter, then read Chapter 12, “Apple Events.”

Implementation of events in Macintosh Common Lisp

Users generate events as a way of directing program flow. Typical events are keystrokes and mouse clicks. Events interrupt a program and often require a response. Whenever possible, Macintosh programs should be event driven.

Macintosh Common Lisp automatically handles events as a background task. When a user generates an event, the current program is interrupted and an event handler handles the event. Program execution does not resume until the event handling function returns. Further event processing is also deferred until the event-handling function returns. For this reason, the computer may not respond to user actions until the event handling is finished.

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

- defining methods associated with specific types of events in a view
- defining all methods associated with a view
- defining a hook procedure that has first priority in processing all events
- disabling all background event processing, and handling events with an event loop

Most programming languages for the Macintosh computer support only the last, and most difficult, method of event handling. MCL programs rarely need to do anything more complex than the first method.

Programs can be initiated from within an event handler; use the function `eval-enqueue` , which lets an event initiate a process with event processing enabled.

How an event is handled

The MCL event system gets each event from the Macintosh Operating System in turn and binds `*current-event*` to it. The event system then determines the type of the event and calls the appropriate event-handling function on the relevant view. If the event is a mouse click, the relevant view is the view in which the click occurred. If the event is a keystroke, the relevant view is the active (frontmost) window.

Functions that end with "`-event-handler`" should be called only by the event system.

Many of the default event-handling methods do nothing, although they are called whenever an event of the appropriate type is processed. These handlers exist so that they may be shadowed by any subclass of `view` that needs to process events of that type.

Some event handlers defined on views do nothing more than invoke the same event handler on each subview. In this way nested views and subviews are processed.

Event-handling functions assume that the `*current-event*` variable is bound to a valid event record (see Chapter 17, "Higher-Level Operating System Interface"). They may also call the current-event information functions (listed in the section "Event Information Functions"), which depend on `*current-event*` being bound.

MCL built-in event handlers

The following are standard event handlers in Macintosh Common Lisp.

`view-click-event-handler` [*Generic function*]

Syntax

`view-click-event-handler` (*view* `simple-view`) *where*
`view-click-event-handler` (*view* `view`) *where*
`view-click-event-handler` (*window-or-item* `fred-mixin`) *where*
`view-click-event-handler` (*item* `dialog-item`) *where*
`view-click-event-handler` (*item* `table-dialog-item`) *where*
`view-click-event-handler` (*item* `scroll-bar-dialog-item`) *where*
`view-click-event-handler` (*menu* `pop-up-menu`) *where*

Description

The generic function `view-click-event-handler` is called by the event system on the window containing the view whenever the user clicks a view or subview.

The `view-click-event-handler` function is not called when the user clicks the title bar, close box, zoom box, or size box of a window. The method for `simple-view` does nothing. Specialized windows provided by the system, such as Fred windows, have special behavior.

If you define any window methods, they must call `call-next-method`.

Arguments

view	A simple view, view, or subview, such as a window or dialog item.
window-or-item	A Fred window or Fred dialog item.
item	A dialog item, table dialog item, or scroll-bar dialog item.
menu	A pop-up menu.
where	The cursor position when the user clicks, expressed in local window coordinates.

Example

The following code displays the cursor coordinates whenever the user clicks `my-window`. (As a subclass of `view`, `window` inherits the `view-click-event-handler` method for `view`.)

```
? (defclass my-window (window)())
#<STANDARD-CLASS MY-WINDOW>

? (defmethod view-click-event-handler
      ((window my-window) where)
    (print (point-string where)))
#<Method VIEW-CLICK-EVENT-HANDLER (MY-WINDOW T)>

? (make-instance 'my-window)
#<MY-WINDOW "Untitled" #x410891>
```

view-key-event-handler

[*Generic function*]

Syntax

```
view-key-event-handler (view simple-view) char
view-key-event-handler (window window) char
view-key-event-handler (item fred-dialog-item) char
view-key-event-handler (window-or-item fred-mixin)
    *current-character*
```

Description

The methods of the generic function `view-key-event-handler` examine the current keystroke and determine what is to be done with it.

The method for `simple-view` calls `ed-beep`. The method for `window` determines whether the key indicates the selection of a default button or indicates a change of the current key handler, then selects the button or passes the keystroke to the appropriate key handler. The method for `fred-mixin` binds the `*current-keystroke*` variable to the keystroke of the current event and runs the Fred command associated with the keystroke. The method for `fred-dialog-item` calls `call-next-method` inside `with-focused-view` and `with-fore-color`.

Arguments

view	A simple view.
window	A window or Fred window.
item	A Fred dialog item.
char	The current keystroke character.
window-or-item	A Fred window or Fred dialog item.

view-activate-event-handler

[*Generic function*]

Syntax

```
view-activate-event-handler (view simple-view)
view-activate-event-handler (view view)
view-activate-event-handler (window window)
view-activate-event-handler (window-or-item fred-mixin)
view-activate-event-handler (item table-dialog-item)
view-activate-event-handler (item scroll-bar-dialog-item)
```

Description The generic function `view-activate-event-handler` is called by the event system when the window containing the view is made active. The method for `view` calls `view-activate-event-handler` on each subview. The method for `simple-view` does nothing.

The method for `window` includes highlighting the window and drawing the size box (if there is one).

Arguments

view	A simple view, view, or subview such as a dialog item.
window	A window or Fred window.
item	A Fred dialog item, table dialog item, or scroll-bar dialog item.
window-or-item	A Fred window or Fred dialog item.

`view-deactivate-event-handler` [*Generic function*]

Syntax

```
view-deactivate-event-handler (view simple-view)
view-deactivate-event-handler (view view)
view-deactivate-event-handler (window window)
view-deactivate-event-handler (window-or-item fred-mixin)
view-deactivate-event-handler (item table-dialog-item)
view-deactivate-event-handler (item scroll-bar-dialog-item)
```

Description The generic function `view-deactivate-event-handler` is called by the event system to deactivate a view. It is called when the window containing the view is active and a different window is made active.

The method for `view` simply calls `view-deactivate-event-handler` on each subview. The method for `window` includes removing the highlight and erasing the size box (if there is one).

Arguments

view	A simple view, view, or subview such as a window or dialog item.
window	A window or Fred window.
item	A Fred dialog item, table dialog item, or scroll-bar dialog item.

key-handler-mixin [*Class name*]

Description The class `key-handler-mixin` should be mixed into any class that handles key events. The class `fred-dialog-item` includes `key-handler-mixin` .

key-handler-list [*Generic function*]

Syntax `key-handler-list` (*view* `simple-view`)

Description The `key-handler-list` generic function returns the list of key handlers associated with *view* .

Argument *view* A simple view or dialog item.

current-key-handler [*Generic function*]

Syntax `current-key-handler` (*window* `window`)

Description The `current-key-handler` generic function returns the current key handler of *window* .

Argument *window* A window.

set-current-key-handler [*Generic function*]

Syntax `set-current-key-handler` (*window* `window`) *item* `&optional` *select-all*

Description The generic function `set-current-key-handler` sets the current key handler of *window* to *item* . If *item* is not already the current key handler and *select-all* is true, `set-current-key-handler` selects all of the window.

Arguments

window	A window.
item	A key handler. If *item* is not a key handler, the function signals an error.
select-all	This variable determines whether the entire text of the key handler is highlighted when it is first selected. The default is `t`; that is, all the text is highlighted and can be manipulated at once.

add-key-handler

[*Generic function*]

Syntax `add-key-handler` (*view* `simple-view`) `&optional` *window*

Description The generic function `add-key-handler` adds a key handler to *view*. It is called by `install-view-in-window` when the view installed is a subclass of `key-handler-mixin`. If *window* has no current key handler, *view* becomes the current key handler.

Arguments

view	A simple view or dialog item.
window	A window to which to add the key handler. The default value is (`view-window` *view*).

remove-key-handler

[*Generic function*]

Syntax `remove-key-handler` (*view* `simple-view`) `&optional` *window*

Description The generic function `remove-key-handler` removes a key handler from a window. It is called by the method of `remove-view-from-window` for `key-handler-mixin`.

Arguments

view	A simple view or dialog item.
window	A window from which to remove the key handler. The default value is (`view-window` *view*).

change-key-handler [*Generic function*]

Syntax `change-key-handler` (*view* `view`)

Description The generic function `change-key-handler` changes the key handler of *view* to the next key handler on `key-handler-list` of *view* .

Argument *view* A simple view or dialog item.

key-handler-p [*Generic function*]

Syntax `key-handler-p` (*item* `dialog-item`)
`key-handler-p` (*key-handler* `key-handler-mixin`)

Description The `key-handler-p` generic function checks to see whether *item* is a key handler. When `key-handler-p` is called on an instance of a class one of whose superclasses is `key-handler-mixin` , the function returns `t` unless the key handler is disabled. The method for `dialog-item` returns `nil`.

Arguments *item* A dialog item.
key-handler A key handler.

key-handler-idle [*Generic function*]

Syntax `key-handler-idle` (*view* `simple-view`) `&optional` *dialog*
`key-handler-idle` (*item* `fred-dialog-item`) `&optional` *dialog*

Description The `key-handler-idle` generic function is called periodically via the default `window-null-event-handler` function to allow a key handler to blink a cursor or perform other periodic activities.

The method for `fred-dialog-item` blinks the insertion point and matches parentheses. The method for `simple-view` does nothing.

Arguments

view	A simple view.
item	A Fred dialog item.
dialog	An argument allowing a dialog to be specified. In system-supplied methods, this argument is ignored.

window-null-event-handler [*Generic function*]

Syntax

```
window-null-event-handler (window window)
window-null-event-handler (window t)
```

Description The generic function `window-null-event-handler` is called on the top window (if there is one) whenever the system is idle. It updates the cursor, runs system tasks, and forces output from `*terminal-io*` . If there is no top window, the unspecialized method simply updates the cursor.

Argument *window* A window.

window-select-event-handler [*Generic function*]

Syntax

```
window-select-event-handler (window window)
```

Description The generic function `window-select-event-handler` is called whenever the user clicks an inactive window. The `window-select-event-handler` function may be specialized, for example, to make a window unselectable.

Argument *window* A window.

window-key-up-event-handler [*Generic function*]

Syntax

```
window-key-up-event-handler (window window)
```

Description The generic function `window-key-up-event-handler` is called whenever a key is released after being pressed. The method for `window` does nothing.

Every key pressed by the user actually generates two events: one when the key is pressed and another when the key is released.

The default Macintosh event mask filters out key-up events. To allow key-up events, call `#_SetEventMask` with an appropriate mask. Note that you must reset the event mask before exiting Lisp. For details on event masks, see Macintosh Technical Note 202 and *Inside Macintosh* .

Argument *window* A window.

`window-mouse-up-event-handler` [*Generic function*]

Syntax `window-mouse-up-event-handler` (*window* `window`)

Description The `window-mouse-up-event-handler` generic function is called whenever the user releases the mouse button. The method for `window` does nothing.

Argument *window* A window.

`window-grow-event-handler` [*Generic function*]

Syntax `window-grow-event-handler` (*window* `window`) *where*

Description The generic function `window-grow-event-handler` is called by the event system whenever the user clicks a window's grow box. The method for `window` calls `#_GrowWindow` , then calls `set-view-size` on the window and the new size.

Arguments *window* A window.

where The position in screen coordinates of the cursor when the mouse button was pressed down.

window-drag-event-handler [*Generic function*]

Syntax window-drag-event-handler (*window* window) *where*

Description The generic function window-drag-event-handler is called by the event system whenever a window needs to be dragged. It calls #_SetClip and #_ClipAbove on the region of the window, copies the contents of the region to the new location of *window*, and calls set-view-position on the window and the new position of the upper-left corner of the window.

Arguments

window A window.

where The position in screen coordinates of the cursor when the mouse button was pressed down.

window-zoom-event-handler [*Generic function*]

Syntax window-zoom-event-handler (*window* window) *message*

Description The generic function window-zoom-event-handler is called by the event system when the user clicks the window's zoom box. It executes the Toolbox calls to zoom the window, then calls window-size-parts.

The function window-size-parts should be specialized if you want to change the contents of a window whenever the window changes size.

Arguments

window A window.

message An integer, #$inZoomOut if the window should move to the window's zoom position and size and #$inZoomIn if the window should move to the position and size it had before zooming out.

window-close-event-handler [*Generic function*]

Syntax `window-close-event-handler` (*window* `window`)

Description The generic function `window-close-event-handler` is called by the event system whenever a window needs to be closed. In the method for `window`, if the Meta key was pressed when the command was given, the command closes all windows in the class of *window*. If the Control key was pressed, *window* is hidden. Otherwise, `window-close` is called on *window*.

Argument *window* A window.

window-do-first-click [*Generic function*]

Syntax `window-do-first-click` (*window* `window`)

Description The generic function `window-do-first-click` determines whether the click that selects a window is also passed to `view-click-event-handler`. The default value is `nil`, meaning that the click that selects a window generates no further action.

You can give a window instance or subclass of `window` its own value for `window-do-first-click`.

Argument *window* A window.

Functions for redrawing windows

Whenever a window is created or uncovered, an update event is posted for the window. The next time events are processed, Macintosh Common Lisp recognizes the update event and calls `window-update-event-handler`.

The following functions relate to redrawing windows.

window-update-event-handler [*Generic function*]

Syntax window-update-event-handler (*window* window)

Description The generic function window-update-event-handler is called by the event system whenever any portion of the window needs to be redrawn. The window version calls #_BeginUpdate to make the VisRgn field of the GrafPort the portion that needs to be redrawn, calls view-draw-contents , and then calls #_EndUpdate to restore the GrafPort VisRgn field.

Because event processing occurs asynchronously, window-update-event-handler may not be called until a moment after a window is created or uncovered. (In the default environment, this may take up to one-third of a second; see event-ticks in the section "The Event Management System" later in this chapter.) This means that anything drawn in the window immediately after it is created or uncovered may be erased when window-update-event-handler is first called.

To fix this problem, simply call event-dispatch before drawing in the window. The function event-dispatch forces the processing of any pending events. Note that it is necessary to call event-dispatch only when drawing occurs soon after a window is created or uncovered.

You should not specialize this function except to note that the window has been updated. To get special drawing behavior, you should instead specialize view-draw-contents .

Argument *window* A window.

view-draw-contents [*Generic function*]

Syntax

```
view-draw-contents (view simple-view)
view-draw-contents (view view)
view-draw-contents (window-or-item fred-mixin)
view-draw-contents (item fred-dialog-item)
view-draw-contents (item table-dialog-item)
view-draw-contents (item scroll-bar-dialog-item)
view-draw-contents (item static-text-dialog-item)
view-draw-contents (menu pop-up-menu)
```

Description

The generic function `view-draw-contents` is called whenever a view needs to redraw any portion of its contents. The `view` method for `view-draw-contents` erases the area in the window's erase region (for new windows, this is the entire content area) and then calls `view-draw-contents` on each subview. You can specialize this function so that a user-defined view can be redrawn when portions of it are covered and uncovered.

When `view-draw-contents` is called by the event system, the view's clip region is set so that drawing occurs only in the portions that need to be updated. This normally includes areas that have been covered by other windows and then uncovered.

Arguments

view	A view or a simple view.
window	A window.
item	A dialog item.
window-or-item	A Fred window or Fred dialog item.
menu	A pop-up menu.

Examples

The following code creates a window that always has a circle drawn in it:

```
? (require 'quickdraw)
"QUICKDRAW"
? (setq foo (make-instance 'window))
#<WINDOW "Untitled" #x4A3BD9>
? (defmethod view-draw-contents ((window (eql foo)))
  (paint-oval window 10 10 100 100))
VIEW-DRAW-CONTENTS
```

(Note that the circle is drawn only after the first time the window is covered and uncovered.)

To add an area (rectangle or region) to the invalid region, call the trap `#_InvalRect` or `#_InvalRgn`. Calling these traps forces the posting of an update event for the window. For this reason, calling these traps from inside `view-draw-contents` or `window-update-event-handler` can lead to an infinite loop.

If you want to invalidate several areas before the update is performed, surround the calls to `#_InvalRect` and `#_InvalRgn` with the special form `without-interrupts`, which temporarily suspends updates.

The following call will force the redrawing of the entire window. It doesn't need `without-interrupts` because there is only one call to `#_InvalRect`. If there were several calls to `#_InvalRect`, `without-interrupts` would postpone updating until the end.

```
(with-port wptr
  (#_invalrect :ptr (rref wptr window.portrect)))
```

The `view-draw-contents` function is not strictly an event handler, since it may be called at any time, not only during event processing. For example, you can use `view-draw-contents` to implement the redrawing that occurs during scrolling, or you can use it to implement a generalized printing mechanism. (For an example, see the file `scrolling-windows.lisp` in your Examples folder.)

`window-draw-grow-icon` [*Generic function*]

Syntax `window-draw-grow-icon` (*window* `window`)

Description The generic function `window-draw-grow-icon` is called when the size box in the lower-right corner of a window must be redrawn. You may need to call this function explicitly if you draw over the size box.

When a window is inactive (that is, not the frontmost window), `window-draw-grow-icon` erases the inside of the size box.

Argument *window* A window.

Event information functions

The following functions give event-related information. To bypass these functions, programs can simply examine `*current-event*` during event handling. (See Chapter 17, "Higher-Level Operating System Interface," for techniques used in examining records.)

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

Syntax

```
view-mouse-position (view simple-view)
view-mouse-position (view null)
```

Description The generic function `view-mouse-position` returns the cursor position as a point expressed in the view's local coordinates. The point is returned as an integer (for a description of points, see Chapter 3, "Points and Fonts"). This function may be called at any time, not just during event processing. The coordinates may be negative, or outside of the view's PortRect, depending on the position of the cursor.

The function `(view-mouse-position nil)` returns the cursor position expressed in screen coordinates.

Argument *view* A simple view.

Example

See the example under `mouse-down-p`.

`mouse-down-p` [*Function*]

Syntax

```
mouse-down-p
```

Description The `mouse-down-p` function returns `t` if the mouse button is pressed and `nil` otherwise. This function may be called at any time, not only during event processing.

Examples

The following example prints the mouse position in window coordinates until the mouse is clicked.

```
(do () ((mouse-down-p))
  (print (point-string (view-mouse-position (front-window)))))
```

The following example prints the mouse position in screen coordinates until the mouse is clicked.

```
(do () ((mouse-down-p))
  (print (point-string (view-mouse-position nil))))
```

double-click-p [*Function*]

Syntax `double-click-p`

Description The `double-click-p` function returns `t` if the click currently being processed was the second half of a double-click. Double-clicks take into account the timing as well as the spacing of consecutive clicks.

The `double-click-p` function always returns `nil` if called from outside event processing. It also returns false if the first click activated the window and `window-do-first-click` is false.

double-click-spacing-p [*Function*]

Syntax `double-click-spacing-p` *point1 point2*

Description The function `double-click-spacing-p` is called by `double-click-p` to see whether two clicks should count as a double-click.

Macintosh guidelines specify that if the cursor is moved excessively between clicks, the clicks do not count as a double-click.

The function `double-click-spacing-p` returns false if *point1* and *point2* are separated by more than 4 pixels, horizontally or vertically. If they are within 4 pixels of each other, both horizontally and vertically, the function returns true.

Arguments

point1	The cursor position during the first click.
point2	The cursor position during the second click.

command-key-p [*Function*]

control-key-p [*Function*]

option-key-p [*Function*]

shift-key-p [*Function*]

caps-lock-key-p [*Function*]

Syntax

```
command-key-p
control-key-p
option-key-p
shift-key-p
caps-lock-key-p
```

Description

Each of these functions has two meanings, depending on whether they are called during event processing or outside it.

If called during event processing, they return true if the corresponding key was pressed during the event; otherwise, they return false. If called outside of event processing, they return true if the key is currently pressed; otherwise, they return false.

Note that some Macintosh keyboards do not have a Control key.

The event management system

This section describes the overall architecture used for implementing event handling in Macintosh Common Lisp.

event-dispatch [*Function*]

Syntax `event-dispatch &optional` *idle*

Description The `event-dispatch` function is called periodically as a background process. The `event-dispatch` function calls `#_WaitNextEvent` and binds the value of `*current-event*` for the duration of the event processing. It then calls `*eventhook*` if `*eventhook*` is not `nil`. If `*eventhook*` returns true, the processing of the event stops. If `*eventhook*` returns false, the event is passed to the system event handlers. Finally, `event-dispatch` checks for deferred Apple events.

If you create a program with a loop that checks for events, you should probably include a call to `event-dispatch` inside the loop. This improves the response time when events occur.

Argument *idle* An argument representing whether the main Lisp process is idle. The default is the value of `*idle*`, which is true when the main Lisp process is idle and `nil` otherwise. The function `event-dispatch` calls `get-next-event` with an event and the value of *idle* .

get-next-event [*Function*]

Syntax `get-next-event` *event* `&optional` *idle mask sleep-ticks*

Description The `get-next-event` function calls `#_WaitNextEvent` to get an event. It disables and reenables the clock sampled by `get-internal-run-time` . (MultiFinder may do a context switch.)

After `#_WaitNextEvent` returns, the function reschedules the `event-dispatch` task, which is the usual caller of `get-next-event` .

Arguments *event* An event record allocated on the stack or the heap.

idle Used to determine the default value of *sleep-ticks*. The default value is `*idle*` , which is true if `get-next-event` is called via `event-dispatch` from the top-level loop when the Listener is waiting for input.

mask This is the `EventMask` argument for `#_WaitNextEvent`, a fixnum. The default is `#$everyEvent`.

sleep-ticks This is the `Sleep` argument to `#_WaitNextEvent`. It determines how many ticks are given to other applications under MultiFinder if no event is pending. The default is determined by the values of the *idle* argument and the global variables `*idle-sleep-ticks*`, `*foreground-sleep-ticks*`, and `*background-sleep-ticks*`. If Macintosh Common Lisp is running in the foreground, then the default is `*idle-sleep-ticks*` if the value of *idle* is true; otherwise, the default is `*foreground-sleep-ticks*`. If Macintosh Common Lisp is running in the background, then the default is `*background-sleep-ticks*` unless that value is `nil`, in which case the default is the same as when Macintosh Common Lisp is running in the foreground.

`*current-event*` [*Variable*]

Description The `*current-event*` variable holds the event record currently being processed. This is bound by `event-dispatch` and is valid only during event processing. The fields of `*current-event*` may be accessed using `rref` (for details see Chapter 17, "Higher-Level Operating System Interface," and *Inside Macintosh*).

The definition of the event record type is

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

eventhook [*Variable*]

Description The *eventhook* variable provides a user hook into the event system. A program can store a function of no arguments in this global variable. The stored function is given the first opportunity to handle all event processing. If the function returns true, the event system assumes the event has been handled and no further processing is done. If the function returns false, the event system assumes the event hasn't been handled and the normal event handlers are invoked.

If *eventhook* is a list of functions with no arguments, they will be called sequentially until either one of them returns true or the list is exhausted. In the latter case, normal event processing occurs.

An *eventhook* function can be used to perform periodic tasks (because it is called whenever there is an event, including a null event).

Note that a slow *eventhook* function can significantly slow down Macintosh Common Lisp.

idle [*Variable*]

Description The *idle* variable signals the event system that the main Lisp process is idle. This changes the sleep time that event dispatch gives to the trap #_WaitNextEvent . This variable is normally bound to true by the read loop as the loop waits for input, and by modal-dialog .

idle-sleep-ticks [*Variable*]

Description The *idle-sleep-ticks* variable holds the value of the sleep time given to #_WaitNextEvent when Macintosh Common Lisp is idle. The initial value is 5.

foreground-sleep-ticks [*Variable*]

Description The `*foreground-sleep-ticks*` variable holds the value of the sleep time given to `#_WaitNextEvent` when Macintosh Common Lisp is running. The initial value is 0.

background-sleep-ticks [*Variable*]

Description The `*background-sleep-ticks*` variable holds the value of the sleep time given to `#_WaitNextEvent` when Macintosh Common Lisp is in the background. The initial value is 5.

event-ticks [*Function*]

Syntax `event-ticks`

Description The `event-ticks` function returns the number of ticks (sixtieths of a second) between calls to `event-dispatch`. This number is applicable when code is running. When Lisp is idling in the main read-eval-print loop, `event-dispatch` is called as close to continuously as possible.

This value is reset on every suspend and resume event, according to the values in `*foreground-event-ticks*` and `*background-event-ticks*`.

set-event-ticks [*Function*]

Syntax `set-event-ticks` *n*

Description The `set-event-ticks` function sets the number of ticks between calls to `event-dispatch` to *n*.

If *n* is too low, `event-dispatch` is called too often, and the system may get bogged down by event processing. If it is too high, the system may not respond smoothly to events. To keep the insertion bar blinking smoothly, for example, a sleep time of 12 to 20 ticks is recommended. This will yield 3 to 5 idle events per second.

This function is called on every suspend and resume event, with the argument `*foreground-event-ticks*` or `*background-event-ticks*`.

Argument *n* An integer determining the number of ticks.

`*foreground-event-ticks*` [*Variable*]

Description The `*foreground-event-ticks*` variable holds the appropriate value for `event-ticks` when Lisp is the foreground application. The initial value is 20.

`*background-event-ticks*` [*Variable*]

Description The `*background-event-ticks*` variable holds the appropriate value for `event-ticks` when Lisp is a background application. The initial value is 5.

`window-event` [*Generic function*]

Syntax `window-event` (*window* `window`)

Description The `window-event` generic function is called by `event-dispatch` to get a window to handle an event. This function is called only when the event system determines the appropriate window. The method of `window-event` for `window` checks the type of the event and calls the appropriate event handler. The `window-event` function should be specialized in windows that need to do something in addition to or different from the default behavior for many types of events.

Argument *window* A window.

`eval-enqueue` [*Function*]

Syntax `eval-enqueue` *form*

Description The `eval-enqueue` function queues up *form* for evaluation in the read-eval-print loop. The `eval-enqueue` function returns immediately. This means that *form* is not executed at the event-handling level but instead is executed as if it had been entered into the Listener. (It is executed only when other forms entered into the Listener or queued up have returned.)

This function is useful for initiating programs from within event handlers. The form is executed as part of the normal read-eval-print loop rather than as part of an event handler. This means that other events can be processed during the execution of *form*.

Note that `eval-enqueue` is a function, and so its argument is evaluated. The result of this evaluation is put into the read-eval-print loop.

Argument *form* An MCL form.

Examples

Here is an example of how to use `eval-enqueue` to evaluate a form with event processing enabled. The first menu item does not disable event handling; the second menu item does. (Note that both can be aborted by typing Command-period.)

```
? (setq my-menu (make-instance 'menu :menu-title "Events"))
#<MENU "Events">
? (menu-install my-menu)
T
? (setq can-process-this-item
        (make-instance 'menu-item
                       :menu-item-title "Process events"
                       :menu-item-action
                       #'(lambda (item)
                           (declare (ignore item))
                           (eval-enqueue
                            '(dotimes (x 100)
                               (print "Choose menus")))))
```

```
#<MENU-ITEM "Process events">
? (setq cant-process-this-item
        (make-instance 'menu-item
                       :menu-item-title
                        "Don't process events"
                       :menu-item-action
                       #'(lambda (item)
                           (declare (ignore item))
                           (dotimes (x 100)
                             (print "Choose if you can"))))))
#<MENU-ITEM "Don't process events">
? (add-menu-items my-menu
                  can-process-this-item
                  cant-process-this-item)
```

The user can also use `eval-enqueue` in dialog boxes. The action is initiated by the dialog box, but the user can still access other parts of the system (including other dialog buttons) while the action is running.

In the following example, the action of the Go button is queued so that other events can be processed while it is running. This allows the user to click the Stop button.

Note that the action of the Stop button does not call `eval-enqueue`. If it did, the queued form would never be run (because the form queued by the Go button would never return). The Stop button communicates with the action of the Go button by changing the value of a lexical variable.

```
? (let ((stop nil))
       (flet ((start ()
                (setq stop nil)
                (loop
                  (if stop (return))
                  (print "Click stop when bored"))))
         (make-instance 'window
           :window-title "Stop and Go"
           :view-subviews
           (list
            (make-instance 'button-dialog-item
              :dialog-item-text "Go"
              :dialog-item-action
              #'(lambda (item)
                  (declare (ignore item))
                  (eval-enqueue `(funcall ,#'start))))
```

```
(make-instance 'button-dialog-item
  :dialog-item-text "Stop"
  :dialog-item-action
  #'(lambda (item)
      (declare (ignore item))
      (setq stop t)))))))
```

without-interrupts [*Special form*]

Syntax `without-interrupts` {*form*}*

Description The `without-interrupts` special form executes *form* with all event processing disabled, including `abort`.

You should use `without-interrupts` sparingly because anything executed dynamically within it cannot be aborted or easily debugged.

However, you must often use `without-interrupts` in code that causes a window to be redisplayed. If you need to invalidate a number of regions in a window, do it inside a `without-interrupts` form to prevent multiple redisplays.

Argument *form* Zero or more Lisp forms.

The cursor and the event system

The *cursor* is the screen image whose motion is controlled by the mouse. As the user moves the mouse on the desktop, the cursor moves correspondingly on the screen.

The cursor often changes shape as it moves over different areas of the screen. For example, when it is on the menu bar or scroll bars, it is shaped like an arrow; when inside a text window, the cursor is shaped like an I-beam.

A program can control the appearance of the cursor in four ways:

- You can define methods for `view-mouse-enter-event-handler` and `view-mouse-leave-event-handler`, specialized on a subclass of `simple-view`. These functions are called when the mouse cursor enters and leaves the area of the view. A possible side effect may be to change the shape of the cursor, for example, from an arrow to an I-beam.
- A view may have a method for the `view-cursor` generic function. The event system sets the cursor according to this method whenever the cursor is over the view.

- The `with-cursor` macro may surround a series of forms. The cursor assumes a given shape for the duration of the macro.
- The variable `*cursorhook*` may be bound to a function or cursor, giving you complete control over the appearance of the cursor.

`view-cursor` [*Generic function*]

Syntax

`view-cursor` (*view* `simple-view`) *point*
`view-cursor` (*view* `view`) *point*
`view-cursor` (*window* `window`) *point*
`view-cursor` (*item* `basic-editable-text-dialog-item`) *point*
`view-cursor` (*window* `fred-window`) *point*

Description

The `view-cursor` generic function determines the cursor shape whenever the window containing the view is active and the cursor is over it. The `view-cursor` function is called by `window-update-cursor`.

Arguments

view	A view or simple view.
window	A window or Fred window.
item	A dialog item.
point	The position of the cursor, expressed as a point.

`window-update-cursor` [*Generic function*]

Syntax

`window-update-cursor` (*window* `null`) *point*
`window-update-cursor` (*window* `window`) *point*

Description

The generic function `window-update-cursor` is called by `update-cursor` whenever the cursor is over the window.

When the mouse is over the front window or any floating window, the `window-update-cursor` method for the `window` class sets the variable `*mouse-view*` to the view containing the mouse, using `find-clicked-subview`. The `window-null-event-handler` method for the `window` class calls `update-cursor`, which calls `*cursorhook*`. The function that is the initial value of `*cursorhook*` calls `window-update-cursor`, which sets the cursor using the value returned by `view-cursor`.

The method for `window` simply sets the cursor to the result of calling the generic function `view-cursor` on the clicked subview of *window* if there is one; otherwise it sets the cursor to the result of calling `window-cursor` on the window.

The `null` method sets the cursor to the value of `*arrow-cursor*` .

The `window-update-cursor` function should be shadowed if the cursor must change according to what part of the window it is over.

Arguments

window	A window or Fred window.
point	The position of the cursor, given in the window's local coordinates.

`view-mouse-enter-event-handler` [*Generic function*]

`view-mouse-leave-event-handler` [*Generic function*]

Syntax

```
view-mouse-enter-event-handler  (view simple-view )
view-mouse-leave-event-handler  (view simple-view )
```

Description The methods of these generic functions for `simple-view` do nothing. You specialize them to create mouse-sensitive items.

Argument

view	A simple view.

`with-cursor` [*Macro*]

Syntax

```
with-cursor cursor {form}*
```

Description The `with-cursor` macro executes zero or more forms with `*cursorhook*` bound to *cursor*.

Arguments

cursor	A cursor record or a `'CURS'` resource ID (see *Inside Macintosh* for details on `'CURS'` resource IDs).
form	Zero or more forms to be executed within the body of the macro.

cursorhook

[*Variable*]

Description The `*cursorhook*` variable may be bound to a function or cursor, giving you complete control over the appearance of the cursor. This variable is bound by `with-cursor` .

If the value of this variable is non-`nil`, then no other cursor functions are called. If the value of `*cursorhook*` is a function, it is called repeatedly in the background and has complete control over the state of the cursor at all times. If it is not a function, it should be a cursor record or a `'CURS'` resource ID.

Its initial value is an internal function that allows events to alter the cursor normally.

update-cursor

[*Function*]

Syntax `update-cursor &optional` *hook*

Description The `update-cursor` function does the actual work of cursor handling. If *hook* is a function or symbol, it is called with no arguments; otherwise, `set-cursor` is called with *hook*.

The `update-cursor` function is called periodically by the global event-handling system. It is not usually necessary to call this function directly, but it may be called to make sure that the cursor is correct at a particular time.

Argument *hook* A function, symbol, or cursor. The default value is `*cursorhook*` .

set-cursor

[*Function*]

Syntax `set-cursor` *cursor*

Description The `set-cursor` function sets the cursor to *cursor*.

Argument *cursor* A cursor record or a `'CURS'` resource ID.

- *Note:* If `set-cursor` is called from anywhere except within a `window-update-cursor` function, a function that is the value of `*cursorhook*`, or a `without-interrupts` special form, the event system's background cursor handling immediately resets the cursor to some other shape. If *cursor* is not of an acceptable type, then no action is taken. To prevent the system from hanging at cursor update time, no error is signaled.

`*arrow-cursor*` [*Variable*]

Description The `*arrow-cursor*` variable specifies the standard north-northwest-arrow cursor shape.

`*watch-cursor*` [*Variable*]

Description The `*watch-cursor*` variable specifies the watch-face shape shown during time-consuming operations, when event processing is disabled.

`*i-beam-cursor*` [*Variable*]

Description The `*i-beam-cursor*` variable specifies the I-beam shape used when the cursor is over an area of editable text.

Event handlers for the Macintosh Clipboard

Data that can be cut and pasted comes in different forms, for example, ASCII text, PICT format graphics, and stylized text. Macintosh Common Lisp provides a simple model for accessing the Macintosh scrap, the structure that supports the Macintosh Clipboard. The Clipboard is accessed through a simple handler, the scrap handler.

Macintosh Common Lisp uses scrap handlers, with one handler for each type of scrap data. The scrap handlers are stored in an association list of the form (*scrap-type-keyword* . *scrap-handler*).

The scrap-type keyword should have a four-character print name. This name is used as an `OSType` data type when the scrap type is communicated to the Macintosh Operating System. (For full details on `OSType` data types, see *Inside Macintosh* .)

In the initial MCL environment, scrap handlers are defined for simple text, formatted Fred text, and Lisp code. You can add handlers for other data types.

When defining new handlers, you should look at the file `pict-scrap.lisp` in the MCL Examples folder to learn how to define a scrap handler for `PICT`s.

For full details on the operation of the Clipboard, you should also consult *Inside Macintosh.*

MCL expressions relating to scrap handlers and scrap types

The following MCL expressions relate to defining scrap handlers and scrap types.

`get-scrap` [*Function*]

Syntax `get-scrap` *scrap-type*

Description The `get-scrap` function returns two values. The first value is the current scrap of *scrap-type* . The second value is `t` if some scrap is found or `nil` if no scrap is found.

The `get-scrap` function looks up the scrap handler for *scrap-type* and calls `get-internal-scrap` with the handler.

Before calling `get-internal-scrap` , `get-scrap` checks to see whether data needs to be imported from the external Macintosh system scrap.

Argument *scrap-type* A scrap type. In the initial MCL environment, the three predefined scrap types are `:text`, `:fred`, and `:lisp`. The file `pict-scrap.lisp` in your Examples folder adds the `:pict` type.

Example

Here is an example of using `get-scrap` to get some text from the Clipboard. (The string `"Here is some text from the Clipboard"` is already in the Clipboard.)

```
? (get-scrap :text)
"Here is some text from the Clipboard"
T
```

put-scrap

[*Function*]

Syntax `put-scrap` *scrap-type scrap-value* `&optional` *overwrite-p*

Description The `put-scrap` function stores *scrap-value* in the scrap, as type *scrap-type* . If the value of *overwrite-p* is true (the default), then all other entries (of any type) in the scrap are cleared; if the value of *overwrite-p* is false, scrap entries of other types are not cleared.

The `put-scrap` function works by looking up the scrap handler for *scrap-type* and calling `set-internal-scrap` with the handler and scrap value.

The `put-scrap` function pushes *scrap-type* onto the `*scrap-state*` list and sets the variable `@` to *scrap-value* .

Arguments

scrap-type	A scrap type. In the initial MCL environment, the three predefined scrap types are: `:text` , `:fred`, and `:lisp` . The file `pict-scrap.lisp` in your Examples folder adds the `:pict` type.
scrap-value	The value of the new scrap: that is, what is stored in the scrap. This should be in a format compatible with *scrap-type* .
overwrite-p	A Boolean variable indicating whether scrap values of other types should be cleared. The default value is true, which clears all other types from the scrap.

Example

The following code puts the phrase “This is only a text” onto the scrap and retrieves it:

```
? (put-scrap :text "This is only a text")
"This is only a text"
? (get-scrap :text)
"This is only a text"
T
```

scrap-state [*Variable*]

Description The *scrap-state* variable contains a list of scrap types and indicates which types currently have a valid scrap. This variable is modified by calls to put-scrap .

scrap-handler-alist [*Variable*]

Description The *scrap-handler-alist* variable contains an association list of scrap-type keywords and scrap-handler objects. Initially, this association list has three entries (one for :text , one for :fred , and one for :lisp). If you define new scrap handlers, you should add entries for them to this list.

Example

This scrap-handler association list contains entries for four scrap handlers. The :pict scrap handler is defined in pict-scrap.lisp in the MCL Examples folder.

```
? *scrap-handler-alist*
((:PICT . #<PICT-SCRAP-HANDLER #x4BB4F9>) (:LISP . #<LISP-SCRAP-HANDLER
#x302CC9>) (:FRED . #<FRED-SCRAP-HANDLER #x3029E1>) (:TEXT . #<TEXT    -
SCRAP-HANDLER #x3025E9>))
```

scrap-handler

[*Class name*]

Description The class `scrap-handler` is the class of scrap handlers. Methods are provided for the scrap-handler functions `get-internal-scrap`, `set-internal-scrap`, `internalize-scrap`, and `externalize-scrap`.

get-internal-scrap

[*Generic function*]

Syntax `get-internal-scrap` (*handler* `scrap-handler`)

Description The `get-internal-scrap` generic function returns the value of the scrap of a given type. This function is called by `get-scrap`.

Argument *handler* A scrap handler.

set-internal-scrap

[*Generic function*]

Syntax `set-internal-scrap` (*handler* `scrap-handler`) *value*

Description The `set-internal-scrap` generic function sets the value of the scrap of a given type. This function is called by `put-scrap`.

Arguments *handler* A scrap handler.
value The new value.

internalize-scrap

[*Generic function*]

Syntax

```
internalize-scrap (handler scrap-handler )
internalize-scrap (handler text-scrap-handler )
```

Description The `internalize-scrap` generic function converts the scrap from external to internal format. This function is called when the user switches into Macintosh Common Lisp from another application or from a desk accessory. The function retrieves data from the Macintosh system heap using the appropriate system calls and then calls `set-internal-scrap` on the result.

The operation of the Macintosh system heap and the appropriate system calls are described in *Inside Macintosh*.

Argument *handler* A scrap handler.

externalize-scrap

[*Generic function*]

Syntax

```
externalize-scrap (handler scrap-handler )
externalize-scrap (handler text-scrap-handler )
```

Description The `externalize-scrap` generic function converts the scrap from internal to external format. This function is called when the user switches from Macintosh Common Lisp to another application or to a desk accessory. The function copies data to the Macintosh system heap using the appropriate system calls.

The default method for `scrap-handler` does nothing.

The operation of the Macintosh system heap and the appropriate system calls are described in *Inside Macintosh*.

Argument *handler* A scrap handler.

Chapter 12 **Apple Events**

Contents

Implementation of Apple events / 412
Receiving Apple events / 412
- How to write an Apple event handler / 413
- Forms associated with Apple events / 414
- Installing Apple event handlers / 417
- Installing handlers for queued Apple event replies / 418
- Standard Apple event handlers / 420

Sending Apple events / 423

This chapter describes how Macintosh Common Lisp supports Apple events and the Apple Event Manager.

You should read this chapter if you want to understand how Macintosh Common Lisp supports required Apple events and if you need to communicate between Macintosh Common Lisp and another process, such as ToolServer or HyperCard.

Before reading this chapter, you should be familiar with the Macintosh Event Manager and with MCL event handling. You should also read about the Apple Event Manager in *Inside Macintosh* . When communicating with another program, you should read the other program's Apple events documentation as well; for example, if you are communicating between Macintosh Common Lisp and HyperCard, you should look at the "AppleEvent Primer" stack in the folder "Your Tour of HyperCard," distributed with version 2.1 of HyperCard.

Implementation of Apple events

The Finder uses Apple events to open applications and quit them, to open documents, and to print documents. In addition, Apple events and the Apple Event Manager may provide services to other applications and request services from them. The Apple Event Manager is available only under System 7. To determine whether the Apple Event Manager is available, call the Gestalt function described in the compatibility guidelines information in *Inside Macintosh* .

Macintosh Common Lisp provides built-in support for receiving Apple events and replying to them. It supports the four required Apple events: Open Application, Open Documents, Print Documents, and Quit Application. It also provides facilities for defining other Apple event handlers.

Creating Apple events and sending them to other applications are not directly supported in Macintosh Common Lisp. However, in your Examples folder are three files illustrating how to send Apple events in Macintosh Common Lisp. These files are

- `appleevent-toolkit.lisp` , containing useful functions for sending Apple events to other processes, including to HyperCard
- `eval-server.lisp` , which shows how to handle standard `doscript` and `eval` Apple events
- `toolserver.lisp` , an example of an Apple events interface to ToolServer

Receiving Apple events

Support for receiving Apple events is built into Macintosh Common Lisp. It directly supports the four required Apple events sent by the Finder: Open Application, Open Documents, Print Documents, and Quit Application.

In addition, the MCL object-oriented event management system provides a high-level, object-oriented way of writing additional Apple event handlers.

How to write an Apple event handler

Macintosh Common Lisp defines a class, `application`, on which Apple event handlers are specialized. Macintosh Common Lisp defines Apple event handlers as generic functions, specialized on the `application` class. Apple event handlers work exactly like other MCL event handlers.

For example, the MCL event handler `window-null-event-handler` is specialized on `window`. To customize this behavior, you can create your own subclasses of `window` and write methods on `window-null-event-handler` for those classes. In just the same way, for Apple event handlers, you can specialize the class `application` and create your own specialized methods on the basic handlers.

You can also write your own handlers.

In Macintosh Common Lisp, only one instance of the class `application` is used at a time. The instance represents the current application object, bound to the variable `*application*`. Apple event handler methods are called on the value of `*application*`.

Because Macintosh Common Lisp bypasses the Apple Event Manager's dispatch routine and does its own dispatching, you do not need to concern yourself with how your function is called by the Apple Event Manager. Lisp takes care of both dispatching and run-time error checking. If no error occurs, the handler should simply return in the normal way. The value returned by the handler is ignored.

If an error occurs, the handler should signal an `appleevent-error` condition, with an error number and an optional error string.

An Apple event handler method has four arguments:

application — The application, always the value of `*application*`.

appleevent — The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc` — a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply — Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon — The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

For an extended example of how to write Apple event handlers, see the file `eval-server.lisp` in the MCL Examples folder.

Forms associated with Apple events

The following functions and variables are associated with receiving Apple events.

`application` [*Class name*]

Description The `application` class is the class of application object on which Apple event handlers are specialized. It has only one instance at a time.

`*application*` [*Variable*]

Description The `*application*` variable is bound to the current instance of the class `application`.

`appleevent-error` [*Condition type*]

Syntax `make-condition 'appleevent-error &rest` *initargs*

Description If an Apple event handler finds an error, it should signal this condition. Any MCL errors that occur while handling the Apple event are automatically caught by Macintosh Common Lisp and handled appropriately.

A condition of the `appleevent-error` condition type is created with the Common Lisp function `make-condition` (see *Common Lisp: The Language*, page 901).

Arguments *initargs* A list of keywords and values used to initialize the `appleevent-error` :

`:oserr` An error number.

`:error-string` A string of human-readable text that identifies the error to a user.

Example

Here is an example of using `appleevent-error`.

```
(error (make-condition 'appleevent-error
                       :oserr #$AEEventNotHandled
                       :error-string "Didn't understand event"))
```

`ae-error` [*Macro*]
`ae-error-str`

Syntax

`ae-error &body (`*form*`)+`
`ae-error-str` *error-string* `&body (`*form*`)+`

Description These macros simplify calls to the Apple Event Manager by signaling the `appleevent-error` condition if the error code returned by the Apple Event Manager is not 0 (`NoErr`). The value of the call should be the value of the body of the macro.

The `ae-error-str` macro lets you specify an error string; the `ae-error` macro does not.

All calls to the Apple Event Manager should be wrapped in a call to either the `ae-error` macro or the `ae-error-str` macro.

Arguments *form* One or more forms to be executed within the body of the macro.

error-string A human-readable error string.

with-aedescs [*Macro*]

Syntax `with-aedescs` (*vars*) `&body` *body*

Description The `with-aedescs` macro creates a temporary record of type `AEDesc` for the extent of the macro. It is similar to the macro `let`. It wraps *body* within an `unwind-protect` , so that no matter how *body* is exited, `with-aedescs` disposes of all its temporary records in the correct way. If the data handle has anything in it, `with-aedescs` calls `#_AEDisposeDesc` . Thus any memory allocated by the Apple Event Manager is properly disposed of.

If you have a need for an `AEDesc` record with indefinite extent, you must use `make-record` . When you want to dispose of the record, you must explicitly call `#_AEDisposeDesc` , then `dispose-record` .

Arguments

vars	One or more variables.
body	One or more forms to be executed within the body of the macro.

Example

There are examples of using `with-aedescs` in `eval-server.lisp` and `tool-server.lisp` in your Examples folder.

check-required-params [*Function*]

Syntax `check-required-params` *error-string appleevent*

Description The `check-required-params` function uses the Apple Event Manager to check whether all required parameters of the Apple event *appleevent* have been extracted. If a parameter has been missed, the `appleevent-error` condition is signaled with `:oserr` `#$AEParamMissed` and `:error-string` *error-string* .

Arguments

error-string	A human-readable error string.
appleevent	An Apple event.

appleevent-idle

[Pascal function]

Syntax `appleevent-idle`

Description The `appleevent-idle` Pascal function should be specified whenever the Apple Event Manager asks for a function to call while it is waiting (for example, in calls to`#_AEInteractWithUser`). It should never be called directly, only passed.

%path-from-fsspec

[*Function*]

Syntax `%path-from-fsspec` *fsspecptr*

Description The `%path-from-fsspec` function extracts a Lisp pathname from *fsspecptr* .

Argument

fsspecptr An MCL macptr that points to an object of Macintosh type `FSSpec` . For more information on macptrs, see the section “Macptrs” in Chapter 16, “Low-Level Operating System Interface.”

Installing Apple event handlers

The following functions install and deinstall Apple event handlers.

install-appleevent-handler

[*Function*]

Syntax `install-appleevent-handler` *class id function* `&optional` *refcon*

Description The function`install-appleevent-handler` installs an Apple event handler.

Arguments

class A four-letter keyword denoting the class of the event, for example, `:|aevt|` .

id A four-letter keyword denoting the ID of the event, for example, `:|odoc|` .

function A function or a symbol with a function binding.

refcon An optional reference identifier, which can be any MCL object; it identifies the specific installation of a handler.

deinstall-appleevent-handler [*Function*]

Syntax `deinstall-appleevent-handler` *class id*

Description The `deinstall-appleevent-handler` function deinstalls an Apple event handler.

Arguments

class A four-letter keyword denoting the class of the event, for example, `:|aevt|`.

id A four-letter keyword denoting the ID of the event, for example, `:|odoc|`.

Installing handlers for queued Apple event replies

Some Apple events received in the event queue, specifically Apple events sent with the `#$kAEQueueReply` mode, are replies to previously sent Apple events. Their handlers are installed differently from other Apple events; the handler is installed when the originating Apple event is sent, and is automatically deinstalled after one use.

The following functions handle queued Apple event replies correctly.

install-queued-reply-handler [*Function*]

Syntax `install-queued-reply-handler` *appleevent-or-id function* `&optional` *refcon*

Description The `install-queued-reply-handler` function installs a handler for a queued reply.

Arguments

appleevent-or-id Either a return ID number or the originating Apple event from which a return ID number can be extracted.

function A function to be called when the reply comes back. This function should be a normal Apple event handler as described in the section "How to Write an Apple Event Handler" earlier in this chapter.

refcon An optional reference identifier, which can be any MCL object; it identifies the specific installation of a handler.

queued-reply-handler [*Generic function*]

Syntax `queued-reply-handler( application application)` *appleevent reply refcon*

Description The generic function `queued-reply-handler` calls the installed reply handler for the return ID of *appleevent*. If there is no applicable reply handler , it calls `no-queued-reply-handler` .

Arguments

application The application, always the value of `*application*` .

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc` —a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

no-queued-reply-handler [*Generic function*]

Syntax `no-queued-reply-handler( application application)` *appleevent reply refcon*

Description The default method of the generic function `no-queued-reply-handler` signals the `appleevent-error` condition with `:oserr #$AEEventNotHandled` .

Arguments

application The application, always the value of `*application*` .

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc` —a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

Standard Apple event handlers

The four basic Apple event handlers are built-in functions in Macintosh Common Lisp.

`open-application-handler` [*Generic function*]

Syntax

`open-application-handler(` *application* `application)`
appleevent reply refcon

Description

The generic function `open-application-handler` handles the Open Application Apple event. The default method does nothing.

Arguments

application The application, always the value of `*application*` .

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc` —a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

quit-application-handler [*Generic function*]

Syntax `quit-application-handler(` *application* `application)`
appleevent reply refcon

Description The generic function `quit-application-handler` handles the Quit Application Apple event. The default method quits Macintosh Common Lisp.

Arguments

application The application, always the value of `*application*` .

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc` —a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

open-documents-handler [*Generic function*]

Syntax `open-documents-handler(` *application* `application)`
appleevent
reply refcon

Description The generic function `open-documents-handler` handles the Open Documents Apple event. The default method opens a file for editing if it is of type `text` or loads it if it is a `fasl` file.

Arguments

application The application, always the value of `*application*` .

appleevent The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc` —a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.

reply Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.

refcon The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

print-documents-handler [*Generic function*]

Syntax `print-documents-handler(` *application* `application)`
appleevent reply refcon

Description The generic function `print-documents-handler` handles the Print Documents Apple event. The default method opens a file or files in Fred and prints them.

Arguments

application The application, always the value of `*application*` .

appleevent	The Apple event, which is an MCL object of type `macptr` and a record of type `AEDesc`—a record with only two fields, a type and a handle. MCL users generally do not have to look at the record structure directly.
reply	Another Apple event record, provided by the Apple Event Manager. If a reply is required, information should be copied into this record using Apple Event Manager calls.
refcon	The handler reference constant, which is any Lisp object. When the handler is installed, you have the option of specifying some Lisp object that serves to distinguish (for instance) two different installations of the same handler. The reference constant is often ignored.

Sending Apple events

There are no built-in functions to send Apple events within Macintosh Common Lisp. However, it is easy to write your own functions for sending Apple events.

To send an Apple event from Macintosh Common Lisp, do the following four steps.

1. Create a target.
2. Create the event.
3. Put data into the Apple event.
4. Send the Apple event.

In the file `appleevent-toolkit.lisp` in your Examples folder, you will find a selection of functions that help you do these steps. For example, you can create Apple events with the MCL function `create-appleevent`.

Since these functions are not part of standard Macintosh Common Lisp, load the file `appleevent-toolkit.lisp` to use them.

The file `eval-server.lisp`, also in the Examples folder, handles `eval`, `dosc`, and `scpt` Apple events. This file also contains code to communicate with HyperCard and MPW.

Chapter 13 **Streams**

Contents

Implementation of streams / 426
MCL expressions relating to streams / 426
Obsolete functions / 439

This chapter discusses the MCL implementation of streams and defines MCL classes and generic functions for dealing with streams.

Implementation of streams

Macintosh Common Lisp implements all of the Common Lisp I/O (input/output) functions by means of streams.

Macintosh Common Lisp implements streams as a simple class of objects, based on the abstract class `stream`. There are only a few things a stream needs to know how to do.

Output streams need to have methods defined for character and string output.

Input streams need to have methods defined to read and "unread" characters (unreading allows "peeking ahead"). Input streams also need to know how to tell when they are at end of file.

You can define your own specialized stream types. The file `serial-streams.lisp` in your Examples folder defines a stream that does input and output through the Macintosh serial port.

MCL expressions relating to streams

The following MCL classes and generic functions deal with streams.

`stream` [*Class name*]

Description The `stream` class is the class from which all streams inherit. It is an abstract class. It should not be directly instantiated but instead should be used to create new subclasses. It has one initialization argument, `:direction`.

`input-stream` [*Class name*]

Description The `input-stream` class is the class of input streams, built on `stream`.

initialize-instance [*Generic function*]

Syntax `initialize-instance (`*stream* `input-stream) &rest` *initargs*

Description The primary method on `initialize-instance` for `input-stream` initializes an input stream. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.) Input streams have one additional initialization argument.

Arguments

stream	A stream.
initargs	This is the additional initialization argument for input streams.
`:direction`	The direction of the input stream. The default value of `:direction` is `:input`.

output-stream [*Class name*]

Description The `output-stream` class is the class of output streams, built on `stream`.

initialize-instance [*Generic function*]

Syntax `initialize-instance (`*stream* `output-stream) &rest` *initargs*

Description The primary method on `initialize-instance` for `output-stream` initializes an output stream. (When instances are actually made, the function used is `make-instance`, which calls `initialize-instance`.) Output streams have one additional initialization argument.

Arguments

stream	A stream.
initargs	This is the additional initialization argument for output streams.
`:direction`	The direction of the output stream. The default value of `:direction` is `:output`.

stream-direction

[*Generic function*]

Syntax `stream-direction ( stream stream)`

Description The `stream-direction` generic function reads the direction of *stream* . It will return `:input` , `:output` , `:io`, or `:closed`.

Argument *stream* A stream.

stream-tyo

[*Generic function*]

Syntax `stream-tyo` (*stream* `output-stream`) *char*

Description The `stream-tyo` generic function directs *stream* to output *char* in an appropriate way; for example, if the stream is a window, `stream-tyo` displays the character in the window.This function must be defined for all output streams.

Arguments *stream* A stream.

char A character or an ASCII value.

Example

The Common Lisp function `write-char` can be defined as

```
? (defun write-char
          (char &optional (stream *standard-output*))
          (if (eq stream t) (setq stream *terminal-io*))
          (stream-tyo stream char))
WRITE-CHAR
```

stream-force-output

[*Generic function*]

Syntax `stream-force-output` (*stream* `output-stream`)

Description The generic function `stream-force-output` physically writes all pending output. It is used for streams that buffer their write operations. For example, it doesn't make sense to do an operating system call or a physical disk access to write a byte every time `stream-tyo` is applied to a disk file. The output is stored in a buffer and written to disk after a certain accumulation or when `stream-force-output` is called. Buffering can significantly increase the speed of streams that have a high per-operation overhead (such as disk output and graphics).

The `stream-force-output` function should be defined for buffered output streams.

Argument *stream* An output stream.

`stream-tyi` [*Generic function*]

Syntax `stream-tyi (`*stream* `input-stream)`

Description The generic function `stream-tyi` reads the next character from the stream and returns it. If this function is at end-of-file, it returns `nil`. Input functions such as `read` and `read-line` work by making repeated calls to `stream-tyi`. This function must be defined for all input streams.

The `stream-tyi` function should never be applied to a Listener directly, but only via `*terminal-io*`.

Argument *stream* An input stream.

Example

The Common Lisp function `read-char` can be defined as

```
? (defun read-char (&optional (stream *standard-input*)
                    (eof-error-p t)
                    eof-value recursive-p)
         (declare (ignore recursive-p))
         (if (eq stream t)
           (setq stream *terminal-io*)
           (or (stream-tyi stream)
               (if eof-error-p
                 (error "End of file on ~s" stream)
                 eof-value))))
READ-CHAR
```

stream-untyi [*Generic function*]

Syntax `stream-untyi` (*stream* `input-stream`) *char*

Description The generic function `stream-untyi` unreads *char* from the stream, effectively pushing it back onto the head of the stream. The next call to `stream-tyi` returns *char* . The `stream-untyi` function cannot be called several times in a row; it can be called only once for each call to `stream-tyi` , and *char* must be the character that was returned by the last call to `stream-tyi` . The `stream-untyi` function must be defined for all input streams.

The `stream-untyi` function is usually implemented in one of two ways: If the stream contains a pointer to a file, string, or other data record, `stream-untyi` simply decrements the pointer. If the stream does not contain a pointer to a data record, then `stream-untyi` sets a variable to the value of *char* . The `stream-tyi` function cooperates by checking the value of the variable; if it is not `nil`, it returns that value instead of getting input from the normal source.

Arguments

stream	An input stream.
char	The last character read from the stream.

Example

The Common Lisp function `unread-char` can be defined as

```
? (defun unread-char (char &optional
                           (stream *standard-input*))
        (if (eq stream t)
          (setq stream *terminal-io*)
          (stream-untyi stream char)))
UNREAD-CHAR
```

stream-writer [*Generic function*]

Syntax `stream-writer` (*stream* `stream`)

Description The generic function `stream-writer` returns two values, a function and a value. Applying the function to the value is equivalent to applying `stream-tyo` to the stream, but is usually much faster. Users can specialize `stream-writer`, but they need to be sure that there are no `stream-tyo` methods specialized on a subclass of the class on which the `stream-writer` method is specialized. The `maybe-default-stream-writer` macro knows how to ensure that there are no such `stream-tyo` methods.

Argument *stream* A stream.

stream-reader [*Generic function*]

Syntax `stream-reader` (*stream* `stream`)

Description The generic function `stream-reader` returns two values, a function and a value. Applying the function to the value is equivalent to applying `stream-tyi` to the stream, but is usually much faster. Users can specialize `stream-reader`, but they need to be sure that there are no `stream-tyi` methods specialized on a subclass of the class on which the `stream-reader` method is specialized. The `maybe-default-stream-reader` macro knows how to ensure that there are no such `stream-tyi` methods.

Argument *stream* A stream.

Example

Here is an example of the use of `stream-reader` and `stream-writer` to define a function that, given two filenames, copies the data fork of the first file to the data fork of the second file. The second file is created if it does not exist. Because `:if-exists :overwrite` has not been specified for the second file, the function signals an error if the second file already exists.

```
? (defun my-copy-file (from-file to-file &aux char)
         (with-open-file (from-stream from-file)
           (with-open-file
             (to-stream to-file :direction :output)
             (multiple-value-bind
               (reader reader-arg)
               (stream-reader from-stream)
               (multiple-value-bind
                 (writer writer-arg)
                 (stream-writer to-stream)
                 (loop
                   (unless
                     (setq char (funcall reader reader-arg))
                     (return))
                   (funcall writer writer-arg char)))))))
MY-COPY-FILE
```

maybe-default-stream-writer [*Macro*]

Syntax `maybe-default-stream-writer` (*stream class*) {*form*}+

Description If the `stream-tyo` method for *stream* is the same as the one for an instance of *class* , the macro `maybe-default-stream-writer` returns the value or values of the last *form* . Otherwise, it returns two values: the effective method for applying `#'stream-tyo` to *stream* , and *stream* itself.

Because `maybe-default-stream-writer` returns the effective method rather than `#'stream-tyo` , it avoids `method-dispatch` .

Arguments

stream	A stream.
class	A Lisp class.
form	One or more Lisp forms.

Example

See the example under `maybe-default-stream-reader` .

maybe-default-stream-reader [*Macro*]

Syntax `maybe-default-stream-reader` (*stream class*) {*form*}+

Description If the `stream-tyi` method for *stream* is the same as the one for an instance of *class* , the macro `maybe-default-stream-reader` returns the value or values of the last *form* . Otherwise, it returns two values: the effective method for applying `#'stream-tyi` to *stream* , and *stream* itself. Because `maybe-default-stream-reader` returns the effective method rather than `#'stream-tyi` , it avoids `method-dispatch` .

Arguments

stream	A stream.
class	A Lisp class.
form	One or more Lisp forms.

Example

Here are examples of `stream-writer` and `stream-reader` created using `maybe-default-` methods.

The MCL `file-stream` class stores a Macintosh pointer containing a parameter block and a buffer for use with the Macintosh `#_Read` and `#_Write` traps. Its `stream-writer` method looks something like this:

```
? (defvar *file-stream-class* (find-class 'file-stream))
*FILE-STREAM-CLASS*
? (defmethod stream-writer ((stream file-stream))
    (maybe-default-stream-writer (stream *file-stream-class*)
      (values #'%ftyo           ; low-level character output function
              (fblock stream))))      ; parameter block pointer
```

The `stream-reader` method for the `file-stream` class is very similar to its `stream-writer` method:

```
? (defmethod stream-reader ((stream file-stream))
    (maybe-default-stream-reader (stream *file-stream-class*)
      (values #'%ftyi          ; low-level character output function
              (fblock stream))))      ; parameter block pointer
STREAM-READER
```

stream-peek

[*Generic function*]

Syntax stream-peek (*stream* stream)

Description The stream-peek generic function returns the last character read into *stream* without removing it from the queue. The next call to stream-tyi or stream-peek reads the same character.

Argument *stream* A stream.

The default stream-peek method is defined as

```
? (defmethod stream-peek ((stream stream))
            (let ((char (stream-tyi stream)))
              (when char (stream-untyi stream char) char)))
#<STANDARD-METHOD STREAM-PEEK (STREAM)>
```

stream-column

[*Generic function*]

Syntax stream-column (*stream* stream)

Description The generic functionstream-column returns the current column of the stream. This is used for tabbing purposes and may also be used by stream-fresh-line .

Argument *stream* A stream.

stream-line-length

[*Generic function*]

Syntax stream-line-length (*stream* stream)

Description The generic functionstream-line-length returns the line length of the stream.

Argument *stream* A stream.

stream-fresh-line [*Generic function*]

Syntax `stream-fresh-line (`*stream* `stream)`

Description The generic function `stream-fresh-line` is called by the Lisp function `fresh-line`. The usual version simply prints a new line. Output streams should provide a definition of this function if they want `fresh-line` to work properly.

The general idea behind `fresh-line` is that it prints a new line if the stream is not already at the beginning of a line.

Argument *stream* A stream.

stream-write-string [*Generic function*]

Syntax `stream-write-string (` *stream* `stream)` *string start end*

Description The generic function `stream-write-string` writes to *stream* the characters of *string* between *start* and *end*.

The expression

```
(stream-write-string stream string 0 (length string))
```

will be faster than

```
(dotimes (i (length string))
   (stream-tyo stream (char string i)))
```

Arguments

stream	A stream.
string	A string.
start	The beginning of the range to copy. The location `0` is before the first character, `1` between the first and the second, and so on.
end	The end of the range to copy.

Example

```
? (stream-write-string *terminal-io* "Hi there" 0 1)
H
NIL
```

stream-clear-input [*Generic function*]

Syntax `stream-clear-input (`*stream* `stream)`

Description The generic function `stream-clear-input` deletes all pending input from the stream. This function is normally defined only for buffered input streams, such as the terminal stream or serial streams.

Argument *stream* A stream.

stream-eofp [*Generic function*]

Syntax `stream-eofp (`*stream* `stream)`

Description The generic function `stream-eofp` returns true if the stream is at its end (that is, if there is no more data to read) and false if there is more data to read from the stream.

The `stream-eofp` function must be defined for all input streams.

Argument *stream* A stream.

stream-listen [*Generic function*]

Syntax `stream-listen (`*stream* `stream)`

Description The generic function `stream-listen` returns true if there are more characters to read from an input stream and false if there are no more characters. The `stream-listen` method for `stream` is simply `(not (stream-eofp))`. The `stream-listen` function does not normally need to be specialized for new `stream` classes.

Argument *stream* A stream.

stream-rubout-handler [*Generic function*]

Syntax `stream-rubout-handler` (*stream* `stream`) *reader*

Description The generic function `stream-rubout-handler` is called by `read` and `read-line` to deal with user editing of the input. This function should call *reader* with one argument, a stream. The default method, specialized on `stream`, simply calls *reader* with an argument of *stream* and provides no deletion handling.

Arguments

stream	A stream.
reader	A stream reader.

Example

The following code handles deletions ("rubouts") for a hypothetical serial I/O stream talking to a dumb terminal:

```
? (defclass serial-io-stream (input-stream output-stream) ())
#<STANDARD-CLASS SERIAL-IO-STREAM>
? (defclass serial-io-stream-rubout-handler
   (input-stream output-stream)
   ((stream :initarg :stream :reader serial-io-stream)
    (buffer :initform (make-array 10
                                  :fill-pointer 0
                                  :adjustable t
                                  :element-type 'character)
            :reader serial-io-stream-buffer)
    (mark :initform nil :accessor serial-io-stream-mark)))
#<STANDARD-CLASS SERIAL-IO-STREAM-RUBOUT-HANDLER>
? (defmethod stream-tyi ((rubout-handler
                          serial-io-stream-rubout-handler))
    (let* ((mark (serial-io-stream-mark rubout-handler))
           (buffer (serial-io-stream-buffer rubout-handler))
           (size (fill-pointer buffer))
           (stream (serial-io-stream rubout-handler))
           (peek (stream-peek stream)))
      (if (and mark (not (eql peek #\rubout)))
        (prog1
          (aref buffer mark)
          (setf (serial-io-stream-mark rubout-handler)
                (and (< (incf mark) size) mark)))
        (let ((char (stream-tyi stream)))
          (when char
```

```
            (if (eql #\rubout char)
              (unless (eql 0 size)
                (tyo #\backspace stream)
                (tyo #\space stream)
                (tyo #\backspace stream)
                (setf (fill-pointer buffer) (decf size))
                (setf (serial-io-stream-mark buffer)
                      (and (> size 0) 0))
                (throw rubout-handler nil))
              (progn
                (vector-push-extend char buffer)
                char)))))))
#<STANDARD-METHOD STREAM-TYI (SERIAL-IO-STREAM-RUBOUT-HANDLER)>
? (defmethod stream-rubout-handler
             ((stream serial-io-stream) reader)
    (let ((rubout-handler
           (make-instance 'serial-io-stream-rubout-handler
            :stream stream)))
      (loop
        (catch rubout-handler
          (return (funcall reader rubout-handler))))))
#<STANDARD-METHOD STREAM-RUBOUT-HANDLER (SERIAL-IO-STREAM T)>
```

---

stream-close

[*Generic function*]

Syntax `stream-close` (*stream* `stream`)

Description The generic function `stream-close` tells the stream that a program is finished with it. After being closed, the stream cannot be used for input or output. Methods on `stream-close` set the `stream-direction` to `:closed` and may perform various cleanup operations, such as disposing of data structures that are no longer needed.

Some streams may be reopened after they have been closed. However, reopened streams have generally lost their previous state.

Argument *stream* A stream.

stream-abort

[*Generic function*]

Syntax `stream-abort (`*stream* `stream)`

Description When the function `close` is called with a true `:abort` keyword, the generic function `stream-abort` is called. The `stream-abort` generic function should handle any bookkeeping for an abnormal closing of the stream.

Argument *stream* A stream.

Obsolete functions

Since Macintosh Common Lisp now uses generic functions, the following two functions are obsolete. They are included for backward compatibility.

tyi

[*Function*]

Syntax `tyi &optional` *stream*

Description The `tyi` function reads one character from *stream* and returns it, using `stream-tyi` . The character is echoed if *stream* is interactive, except that the DEL character is not echoed. This function is included for compatibility with earlier versions of Lisp.

Argument *stream* A stream. The default value is `*standard-input*` .

tyo

[*Function*]

Syntax `tyo` *char* `&optional` *stream*

Description The `tyo` function outputs *char* to *stream* , using `stream-tyo` . It is included for compatibility with earlier versions of Lisp.

Arguments *char* An integer or a character object.

stream A stream. The default value is `*standard-output*` .

Chapter 14 **Programming the Editor**

Contents

How Macintosh Common Lisp handles text editing / 443
- Fred windows / 443
- Fred dialog items / 445
- Buffers and buffer marks / 445
- Copying and deletion mechanism: The kill ring / 446

MCL expressions relating to buffer marks / 447
- Using multiple fonts / 464
- Functions for manipulating fonts and font styles / 465

Fred functions / 470

The class `fred-mixin` / 470

Fred window and Fred dialog item functions / 471
- Creating a Fred window / 471
- Functions relating to Fred windows and Fred dialog items / 477

Functions implementing standard editing processes / 493
- Multiple-level Undo / 495
- Functions relating to Undo / 496

Working with the kill ring / 500
- Functions for working with the kill ring / 501
- Using the minibuffer / 502
- Functions for working with the minibuffer / 502

Defining Fred commands / 504

Fred command tables / 505
- Keystroke codes and keystroke names / 505
- Command tables / 507
- Fred dispatch sequence / 507
- MCL expressions associated with keystrokes / 507
- MCL expressions relating to command tables / 512

This chapter describes the functions and concepts needed to program Fred the editor. You should read it if you are creating an application that uses text editing.

You should be familiar with the standard functionality of the editor, Fred, discussed in Chapter 2, “Editing in Macintosh Common Lisp.” You should also be familiar with the way that Macintosh Common Lisp handles windows, discussed in Chapter 5, “Views and Windows.”

If you are creating editable dialog items, you should also read Chapter 6, “Dialog Items and Dialogs.”

How Macintosh Common Lisp handles text editing

Editability is not an innate property of text in files; it is something that happens in windows and in dialog items. Text is editable in MCL window classes when one of the window class's superclasses (usually its direct superclass) is the class `fred-mixin`. In standard Macintosh Common Lisp, this class is added to **Fred windows**, including Listeners, and to editable dialog items (called **Fred dialog items**).

The actual text being edited is stored in a **buffer.** Macintosh Common Lisp does not keep a file open while it is being edited. It merely reads the file into a buffer, lets you edit it, then reopens the file when you save the buffer.

A buffer is implemented like a more efficiently editable string. (Conceptually, it is a sequence of characters, each of which has an associated font.) The internal representation of a Fred buffer allows characters to be inserted or deleted quickly.

Operations are performed within buffers at places called **buffer marks.** Roughly, a buffer mark indicates any interesting place in the buffer, such as the place where text is inserted, the beginning of a selection range, or the point at which text becomes visible in the window. Buffer marks contain a position and a pointer to their owning buffer. A Fred buffer can be accessed only through its buffer marks.

The buffer is displayed in the window. A window's contents include the whole buffer, not only the part currently visible on screen (so, to take a trivial example, you can select or search the entire contents of a Fred window, not only the visible portion of the buffer).

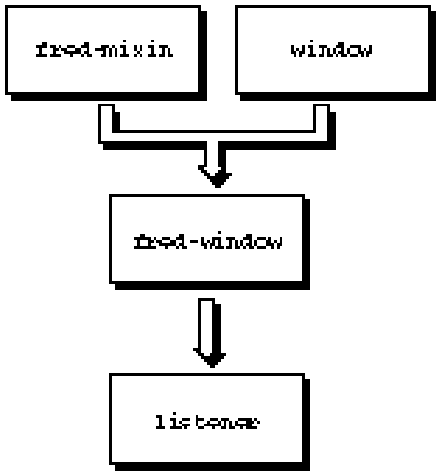
In general, higher-level editing operations, such as setting fonts, are implemented as methods applied to instances of `fred-mixin`, the class that governs the behavior of Fred windows and Fred dialog items. Low-level operations take a buffer mark as an argument.

Each of these concepts is discussed in more detail in the paragraphs that follow.

Fred windows

Fred windows are the windows in which editor buffers appear. The class `fred-window` is a subclass of `window` and a superclass of `listener`. Its superclasses are the class `fred-mixin`, which adds slots whose values define the general behavior of Fred, and `window`. (See Figure 14-1.)

n **Figure 14-1** The classes `fred-window` , `fred-mixin` , `window` , and `listener`



Fred windows are output streams and may be used in any situation that calls for a stream. Characters output to a Fred window stream are inserted at the insertion point, which corresponds to a buffer mark. Stream output to Fred windows is buffered and is not displayed until `fred-update` or `force-output` is called.

A Fred window has

- a buffer. The buffer is displayed in the window.
- an insertion point, indicated by a blinking vertical line, where typing is usually inserted in the window. The insertion point is a buffer mark.
- a display start position, indicating the beginning of the first line of the buffer displayed in the buffer's window. The display start position is also a buffer mark.
- a filename string.
- a horizontal and a vertical scroll bar.
- a minibuffer, a small independent display at the bottom of the window, used to display MCL messages.

Fred windows may also have

- a selection range, which is displayed in inverse video and is defined by a buffer mark. The buffer mark that defines the selection range is distinct from the insertion point, although most Fred window functions try to keep the insertion point at one end of the selection range.

Fred dialog items

Fred dialog items, a subclass of `simple-view`, include any dialog item with editable text. Their class is `fred-dialog-item`, whose superclasses include the class `fred-mixin` and an internal class whose superclasses are `key-handler-mixin` and `dialog-item`.

A Fred dialog item has

- a view size
- a view position
- a default font
- one or more colors
- an event handler
- a key handler
- text

It also knows whether or not it is enabled.

For a full description of dialog items, see the sections on dialog items in Chapter 6, "Dialog Items and Dialogs."

Buffers and buffer marks

A **buffer** is a sequence of characters much like a string. However, the implementation of buffers makes the insertion and deletion of characters much more efficient than with strings. The characteristics of buffers are inherited from `fred-mixin`.

A buffer has

- a set of buffer marks
- a modification counter, which is incremented any time the buffer is modified
- a property list

There is no data type for buffers; the representation of buffers is internal to the MCL implementation. Buffers keep track of their operations by using buffer marks and are accessed only through buffer marks.

A **buffer mark** contains a position and a pointer to its buffer. A buffer mark indicates a position in a buffer where some editing process might take place. For example, the position at which new characters can be entered into a buffer is indicated by a buffer mark. The positions of buffer marks are recalculated each time the buffer is modified. For example, when you type new characters or paste a selection into a buffer, the position of the buffer mark corresponding to the current insertion point changes, and so do the positions of buffer marks located after the newly inserted text.

Buffer marks are defined by the data type `buffer-mark` . An instance of `buffer-mark` contains

- a pointer to its owning buffer
- a position in its buffer, dynamically updated as the buffer changes

The following properties of a buffer mark can be determined:

- A direction, forward or backward. The direction determines what happens when a character is inserted at the position of the mark. Forward marks move forward, placing themselves after the new character; backward marks stay behind the new character. The insertion point is initially a forward mark but can be changed to a backward mark.
- Other information, such as contents and default insertion font.

Copying and deletion mechanism—The kill ring

For deletion and copying, Fred supports both an Emacs-style kill ring and the Macintosh Clipboard.

The **kill ring** is a list of blocks of text that have been deleted. Any command that deletes or copies text saves that text on the kill ring. There is only one kill ring, shared among all buffers; with it you can move or copy text from buffer to buffer. Cut, Copy, and Emacs commands such as Control-K (`kill-line`) move text onto the kill ring. Also saved on the kill ring is text deleted incidentally, for example, text that is deleted when you type or paste over a selection. Saving all deleted text onto the kill ring provides a level of safety not supported by the usual Macintosh Undo mechanism.

The Macintosh commands Cut, Copy, and Paste move text to and from the Clipboard. The Paste command ignores the kill ring, always pasting from the Clipboard.

The section “Working With the Kill Ring” describes tools for working with the kill ring.

MCL expressions relating to buffer marks

The following MCL expressions relate to buffer marks.

buffer-mark [*Class name*]

Description The `buffer-mark` class is the class of buffer marks.

The following functions govern the operation of buffers.

buffer-mark-p [*Function*]

Syntax `buffer-mark-p` *thing*

Description The `buffer-mark-p` function returns `t` if and only if *thing* is a buffer mark; otherwise, it returns `nil`. It has the same effect as `(typep` *thing* `'buffer-mark)`.

Argument *thing* Any Lisp object.

make-mark [*Function*]

Syntax `make-mark` *buffer-mark* `&optional` *position backward-p*

Description The `make-mark` function creates and installs a new mark in the same buffer as *buffer-mark*, with the specified position and direction. If given, *position* must be a mark or an integer. The new mark is returned.

Arguments

buffer-mark A buffer mark.

position A position in a buffer. It can be `nil` (the default), an integer offset from the beginning of the buffer, a mark, or `t`, meaning the end of the buffer. Its default value is `(buffer-position` *buffer-mark* `)`.

backward-p A Boolean value. If its value is `t`, a backward mark is created. The default value is `nil`.

set-mark

[*Function*]

Syntax `set-mark` *buffer-mark position*

Description The `set-mark` function sets the position of *buffer-mark* to *position* and returns the updated mark.

Arguments

buffer-mark A buffer mark.

position A position in a buffer. It can be an integer offset from the beginning of the buffer, a mark, or `t`, meaning the end of the buffer.

move-mark

[*Function*]

Syntax `move-mark` *buffer-mark* `&optional` *distance*

Description The `move-mark` function moves *buffer-mark* an amount specified by *distance* , which should be an integer. This function is equivalent to (`set-mark` *buffer-mark* (`+` (`buffer-position` *buffer-mark*) *distance*)).

Arguments

buffer-mark A buffer mark.

distance A positive or negative integer specifying the distance to move the mark. The default value is 1.

mark-backward-p

[*Function*]

Syntax `mark-backward-p` *buffer-mark*

Description The `mark-backward-p` function returns `t` if *buffer-mark* is a backward mark; otherwise, it returns `nil`.

Argument

buffer-mark A buffer mark.

same-buffer-p [*Function*]

Syntax `same-buffer-p` *buffer-mark1 buffer-mark2*

Description The `same-buffer-p` function returns `t` if *buffer-mark1* and *buffer-mark2* are buffer marks pointing to the same buffer.

Arguments

buffer-mark1	A buffer mark.
buffer-mark2	Another buffer mark.

make-buffer [*Function*]

Syntax `make-buffer &key :chunk-size :read-only :font`

Description The `make-buffer` function returns a buffer mark representing a new, empty buffer.

Arguments

`:chunk-size`	The length of each string making up a buffer, which is implemented as a linked list of strings. The default is `#x1000` (`4096`) for a `fred-window` and `128` for a `fred-dialog-item`. (The generic function `fred-chunk-size` returns the actual length.)
`:read-only`	The read/write status of the buffer, specifying whether information in the buffer can be modified. If the value of `:read-only` is true, any attempt to modify the buffer will cause an error to be signaled. The default value is `nil`.
`:font`	The display font for characters typed into the buffer. The default is the value of `*fred-default-font-spec*`.

Example

```
? (setq my-buffer (make-buffer))
#<BUFFER-MARK 0/0>
```

buffer-size [*Function*]

Syntax `buffer-size` *buffer-mark*

Description The `buffer-size` function returns the number of characters in *buffer-mark* .

Argument *buffer-mark* A buffer mark.

Example

Here is an example of the use of `buffer-size` . (The generic function `fred-buffer` returns the insertion point associated with a Fred window; it is documented later in this chapter in the section "Functions Relating to Fred Windows and Fred Dialog Items.")

```
? (setf my-window (make-instance 'fred-window))
#<FRED-WINDOW "New" #x4E15D9>
? (buffer-size (fred-buffer my-window))
0
```

buffer-modcnt [*Function*]

Syntax `buffer-modcnt` *buffer-mark*

Description The `buffer-modcnt` function returns the modification count of *buffer - mark* . The modification count is the number of times the buffer has been modified since it was created. By comparing the value returned by `buffer -modcnt` at different times, you can tell whether the buffer has been modified in the meantime.

Argument *buffer-mark* A buffer mark.

Example

This code shows how you might use `buffer-modcnt` to determine whether a buffer has been modified.

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

buffer-plist

[*Function*]

Syntax `buffer-plist` *buffer-mark*

Description The `buffer-plist` function returns the property list of the buffer containing *buffer-mark* . The system itself keeps certain information on buffer property lists, so you should not use `setf` with `buffer-plist`.

At present, Macintosh Common Lisp uses the buffer's property list for nothing except storing the value of `fred-package` .

Argument *buffer-mark* A buffer mark.

Example

```
? (buffer-plist my-buffer)
NIL
```

buffer-getprop

[*Function*]

Syntax `buffer-getprop` *buffer-mark key* `&optional` *default*

Description The `buffer-getprop` function looks up the *key* property on the property list (`buffer-plist` *buffer-mark*). It returns the value associated with *key* , if found; otherwise, it returns *default* .

Arguments

buffer-mark	A buffer mark.
key	A property on the property list associated with *buffer-mark* .
default	The default value to be returned.

Example

See the example under `buffer-putprop`.

buffer-putprop

[*Function*]

Syntax `buffer-putprop` *buffer-mark key value*

Description The `buffer-putprop` function gives *key* the value *value* on the property list (`buffer-plist` *buffer-mark*). The value *value* is returned.

Arguments

buffer-mark	A buffer mark.
key	The key to set in the property list.
value	The new value to associate with *key* .

Example

```
? (buffer-putprop my-buffer :font '("Times" 12))
("Times" 12)
? (buffer-getprop my-buffer :font)
("Times" 12)
? (buffer-plist my-buffer)
(:FONT ("Times" 12))
```

buffer-position

[*Function*]

Syntax `buffer-position` *buffer-mark* `&optional` *position*

Description The `buffer-position` function returns the position (number of characters from the start of *buffer-mark*) of *position* in *buffer-mark* . If *position* is `nil` (the default) or not supplied, the value of (`buffer-position` *buffer-mark*) is returned If *position* is an integer, `buffer-mark` checks that *position* is in the range of legal buffer positions, then returns *position* . If *position* is a mark in the same buffer as *buffer-mark* , its position is returned. Otherwise, an error is signaled.

Arguments

buffer-mark	A buffer mark.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer, a mark, or `t`, meaning the end of the buffer. Its default value is (`buffer-position` *buffer-mark*).

buffer-line

[*Function*]

Syntax `buffer-line` *buffer-mark* `&optional` *position*

Description The `buffer-line` function returns the line number of *buffer-mark* that contains *position* .

Arguments

buffer-mark A buffer mark.

position A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark* . Its default value is (`buffer-position` *buffer-mark*).

buffer-line-start

[*Function*]

Syntax `buffer-line-start` *buffer-mark* `&optional` *start count*

Description The `buffer-line-start` function returns two values. If there are enough lines in the buffer, `buffer-line-start` returns as the first value the position of the start of the *count*th line from the line containing `start` and, as the second value, `nil`. If there aren't enough lines, `buffer-line-start` returns as the first value the end of the range searched (the start of the buffer if *count* is negative, the end of the buffer if *count* is positive) and, as the second value, an integer specifying the number of lines of shortfall.

Arguments

buffer-mark A buffer mark.

start A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark* .

count The number of lines from *start* to search. A *count* of 0 means the start of the line containing *start* , a *count* of –1 means the start of the previous line, a *count* of 1 means the start of the next line, and so on. The default value is 0.

buffer-line-end

[*Function*]

Syntax `buffer-line-end` *buffer-mark* `&optional` *end count*

Description The `buffer-line-end` function returns two values. If there are enough lines in the buffer, `buffer-line-end` returns, as the first value, the position of the start of the *count*th line from the line containing *end* and, as the second value, `nil`. If there aren't enough lines, `buffer-line-end` returns the end of the range searched (the start of the buffer if *count* is negative, the end of the buffer if *count* is positive) and a second value specifying the number of lines of shortfall.

Arguments

buffer-mark A buffer mark.

end A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark* .

count The number of lines from *end* to search. A *count* of 0 means the end of the line containing *end*, a *count* of –1 means the end of the previous line, a *count* of 1 means the end of the next line, and so on. The default value is 0.

buffer-column

[*Function*]

Syntax `buffer-column` *buffer-mark* `&optional` *position*

Description The `buffer-column` function returns the number of characters between *position* and the start of the line that contains *position* .

Arguments

buffer-mark A buffer mark.

position A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark* . Its default value is (`buffer-position` *buffer-mark*).

lines-in-buffer [*Function*]

Syntax `lines-in-buffer` *buffer-mark*

Description The `lines-in-buffer` function returns the number of lines in the buffer.

This function works by counting the number of newline characters in the buffer; therefore it takes longer to run as the buffer grows in size.

Argument *buffer-mark* A buffer mark.

buffer-char [*Function*]

Syntax `buffer-char` *buffer-mark* `&optional` *position*

Description The `buffer-char` function returns the character at the specified position in *buffer-mark* .

Arguments

buffer-mark A buffer mark.

position A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark* . Its default value is (`buffer-position` *buffer-mark*).

buffer-char-replace [*Function*]

Syntax `buffer-char-replace` *buffer-mark char* `&optional` *position*

Description The `buffer-char-replace` function replaces the character at the specified position in *buffer-mark* with *char* . It returns the old character.

Arguments

buffer-mark A buffer mark.

char A character to insert in the buffer.

position A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark* . Its default value is (`buffer-position` *buffer-mark*).

buffer-insert [*Function*]

Syntax `buffer-insert` *buffer-mark string* `&optional` *position*

Description The `buffer-insert` function inserts *string* into *buffer-mark* at *position* .

Arguments

buffer-mark	A buffer mark.
string	Anything acceptable to the `string` function: that is, a string, symbol, or character.
position	A position in a buffer. It can be an integer offset from the beginning of the buffer or a mark in the same buffer as *buffer-mark* . Its default value is (`buffer-position` *buffer-mark*).

buffer-substring [*Function*]

Syntax `buffer-substring` *buffer-mark one-end* `&optional` *other-end*

Description The `buffer-substring` function returns a simple string of the characters in *buffer-mark* in the range described by the arguments.

The order of the *-end* arguments doesn't matter; they are interpreted in whatever order produces a meaningful result.

Arguments

buffer-mark	A buffer mark.
one-end	One end of the string to be returned.
other-end	The other end of the string. The default value is the position of *buffer-mark* .

buffer-insert-substring [*Function*]

Syntax `buffer-insert-substring` *buffer-mark string* `&optional` *start end position*

Description The `buffer-insert-substring` function inserts the substring of *string* specified by *start* and *end* into *buffer-mark* at *position* .

Arguments

buffer-mark	A buffer mark.
string	Anything acceptable to the `string` function: that is, a string, symbol, or character.
start	The starting position. The default value is 0.
end	The ending position. The default value is (`length` *string*).
position	An integer position, or a mark in the same buffer as *buffer-mark*. The default value is (`buffer-position` *buffer-mark*).

buffer-insert-with-style [*Function*]

Syntax

`buffer-insert-with-style` *buffer-mark string style* `&optional` *start*

Description

The `buffer-insert-with-style` function inserts *string* in *buffer-mark*. If *style* is given, the function sets the style of *buffer-mark* to *style*, beginning at *start*.

Arguments

buffer-mark	A buffer mark.
string	Anything acceptable to the `string` function: that is, a string, symbol, or character.
style	A font style.
start	The starting position. The default value is the position of *buffer-mark*.

buffer-current-sexp [*Function*]

Syntax

`buffer-current-sexp` *buffer-mark* `&optional` *position*

Description

The `buffer-current-sexp` function returns two values. The first is the s-expression in *buffer-mark* at *position*. Because this function actually reads the characters from the buffer, you may evaluate what it returns. It returns `nil` if there is no s-expression at *position*.

The second value returned is `t` if an s-expression was found at *position*, or `nil` if no s-expression was found at *position*.

Arguments

buffer-mark	A buffer mark.
position	An integer position, or a mark in the same buffer as *buffer-mark* . The default value is (`buffer-position` *buffer-mark*).

The definition of the current s-expression is determined according to the following rules:

- If *position* precedes an open parenthesis, the current s-expression is the text between that open parenthesis and its matching close parenthesis.

  position (. . . . . *current s-expression* . . . . .)

- If *position* follows a close parenthesis, the current s-expression is the text between that close parenthesis and its matching open parenthesis.

  (. . . . . *current s-expression* . . . . .) *position*

- If *position* precedes a quotation mark (that is, double quotes), the current s-expression is the text between that quotation mark and its matching quotation mark.

  position " . . . . . *current s-expression* . . . . . "

- If *position* follows a quotation mark, the current s-expression is the text between that quotation mark and its matching quotation mark.

  " . . . . . *current s-expression* . . . . . " *position*

- If *position* is immediately before, immediately after, or in the middle of a symbol, the current s-expression is the symbol.

- Otherwise there is no current s-expression.

`buffer-current-sexp-start` [*Function*]

Syntax `buffer-current-sexp-start` *buffer-mark* `&optional` *position*

Description The `buffer-current-sexp-start` function returns the position of the current s-expression, or `nil` if there is no current s-expression. The current s-expression is determined according to the rules described in the definition of `buffer-current-sexp` .

Arguments

buffer-mark	A buffer mark.
position	An integer position, or a mark in the same buffer as *buffer-mark* . The default value is (`buffer-position` *buffer-mark*).

buffer-current-sexp-bounds [*Function*]

Syntax `buffer-current-sexp-bounds` *buffer-mark* `&optional` *position*

Description The `buffer-current-sexp-bounds` function returns the starting and ending positions of the current s-expression; if there is no current s-expression, it returns `nil`. The current s-expression is determined according to the rules given for the function `buffer-current-sexp` earlier in this section.

Arguments

buffer-mark	A buffer mark.
position	An integer position, or a mark in the same buffer as *buffer-mark* . The default value is (`buffer-position` *buffer-mark*).

buffer-delete [*Function*]

Syntax `buffer-delete` *buffer-mark start* `&optional` *end*

Description The `buffer-delete` function deletes the characters in *buffer-mark* in the range described by the *start* and *end* arguments.

Arguments

buffer-mark	A buffer mark.
start	The start of the range to delete.
end	The end of the range to delete. The default value is (`buffer-position` *buffer-mark*).

buffer-downcase-region [*Function*]
buffer-upcase-region [*Function*]
buffer-capitalize-region [*Function*]

Syntax
`buffer-downcase-region` *buffer-mark start* `&optional` *end*
`buffer-upcase-region` *buffer-mark start* `&optional` *end*
`buffer-capitalize-region` *buffer-mark start* `&optional` *end*

Description These three functions convert the words between *start* and *end* to lowercase, uppercase, and initial capitals, respectively.

Arguments

buffer-mark	A buffer mark.
start	The start of the region to convert, expressed as an integer or buffer mark.
end	The end of the region to convert, expressed as an integer or buffer mark. The default value is (`buffer-position` *buffer-mark*).

`buffer-char-pos` [*Function*]

`buffer-not-char-pos` [*Function*]

Syntax

`buffer-char-pos` *buffer-mark char-or-string* `&key :start :end :from-end`

`buffer-not-char-pos` *buffer-mark char-or-string* `&key :start :end :from-end`

Description

The `buffer-char-pos` function returns the position of the first occurrence of a character that is an element of *char-or-string* in the buffer between `:start` and `:end`. An error is signaled if `:start` is not less than `:end`. The *char-or-string* argument may be a string or a character. The search is case insensitive; that is, the comparison is done using `char-equal`.

The `buffer-not-char-pos` function performs the same function, except that it returns the position of the first character in the buffer that is not an element of *char-or-string* .

Arguments

buffer-mark	A buffer mark.
char-or-string	A character or string of characters. The string is treated as a set of characters.
`:start`	The first position in the buffer to search. The default value is the cursor position.
`:end`	The last position in the buffer to search. The default value is 0 if `:from-end` is true or the buffer size if `from-end` is false.
`:from-end`	A value determining in which direction the search proceeds. If `:from-end` is true, the search proceeds from `:end` to `:start`. The default value is `nil`.

buffer-string-pos [*Function*]

Syntax `buffer-string-pos` *buffer-mark string* `&key :start :end :from-end`

Description The `buffer-string-pos` function returns the position of the first occurrence of *string* in the buffer between `:start` and `:end`. An error is signaled if `:start` is not less than `:end`. If `:from-end` is non-`nil`, the search proceeds backward. If *string* is not found, `nil` is returned; otherwise, the position of the first character of the string is returned. The search is case insensitive; that is, the comparison is done using `char-equal`.

Arguments

buffer-mark	A buffer mark.
string	A string.
`:start`	The first position in the buffer to search. The default value is the cursor position.
`:end`	The last position in the buffer to search. The default is 0 if `from-end` is true or the buffer size if `from-end` is false.
`:from-end`	A value determining in which direction the search proceeds. If `:from-end` is true, the search proceeds from `:end` to `:start`. The default is `nil`.

buffer-substring-p [*Function*]

Syntax `buffer-substring-p` *buffer-mark char-or-string* `&optional` *position*

Description The `buffer-substring-p` function returns `t` if *char-or-string* appears at the specified position in *buffer-mark*. The comparison is case insensitive. The *char-or-string* argument may be a string or a character.

Arguments

buffer-mark	A buffer mark.
char-or-string	A character or string of characters.
position	A position in the buffer. The default value is the position of *buffer-mark*.

buffer-word-bounds [*Function*]

Syntax `buffer-word-bounds` *buffer-mark* `&optional` *position*

Description The `buffer-word-bounds` function returns two values, the start and end of the word at *position* . If *position* is not in a word, both values are equal to (`buffer-position` *buffer-mark position*).

Arguments

buffer-mark A buffer mark.

position A position in the buffer. The default value is the position of *buffer-mark* .

buffer-fwd-sexp [*Function*]

Syntax `buffer-fwd-sexp` *buffer-mark* `&optional` *position end character ignore-#-comments*

Description The `buffer-fwd-sexp` function returns the position of the end of the s-expression starting at *position* . If the s-expression is not closed, the function returns `nil`.

Arguments

buffer-mark A buffer mark.

position A position in the buffer. The default value is the position of *buffer-mark* .

end The last position in the buffer to search.The default value is (`buffer-size` *buffer-mark*).

character A character. The value defaults to (`buffer-char` *buffer-mark position*).

ignore-#-comments
A value that determines whether to ignore initial sharp-sign (number-sign) comments. If the value of this is true, initial sharp-sign comments are skipped; if `nil`, the end of an initial sharp-sign comment is returned.

buffer-bwd-sexp [*Function*]

Syntax `buffer-bwd-sexp` *buffer-mark* `&optional` *position over-sharps*

Description The `buffer-bwd-sexp` function returns the position of the start of the s-expression ending at *position* , the value of which defaults to `(buffer-position` *buffer-mark* `)`. If the s-expression is not closed, the function returns `(max 0 (1- position))` .

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default value is the position of *buffer-mark* .
over-sharps	An argument specifying whether to consider a preceding reader macro, such as `#@`, as part of the s expression. If the value of this is true, reader macros are included; if `nil`, they are not.

buffer-skip-fwd-wsp&comments [*Function*]

Syntax `buffer-skip-fwd-wsp&comments` *buffer-mark start end*

Description The `buffer-skip-fwd-wsp&comments` function returns the first position in *buffer-mark* after *start* that is not white space or within a comment.

If *start* is not less than *end,* `nil` is returned.

Arguments

buffer-mark	A buffer mark.
start	The first position in the buffer to search.
end	The last position in the buffer to search.

buffer-insert-file [*Function*]

Syntax `buffer-insert-file` *buffer-mark pathname* `&optional` *position*

Description The `buffer-insert-file` function inserts the file specified by *pathname* into *buffer-mark* at *position* .

This function preserves and restores font information; see the next section, “Using Multiple Fonts.”

Arguments	*buffer-mark*	A buffer mark.
	pathname	The pathname of the file to insert.
	position	A position in the buffer. The default is `(buffer-position` *buffer-mark* `)`.

`buffer-write-file` [*Function*]

Syntax `buffer-write-file` *buffer-mark pathname* `&key :if-exists`

Description The `buffer-write-file` function outputs the contents of the buffer to the file specified by *pathname*.

This function preserves and restores font information; see the next section, "Using Multiple Fonts."

Arguments	*buffer-mark*	A buffer mark.
	pathname	The pathname of the file to insert.
	`:if-exists`	A keyword that specifies what to do if the file already exists. If the value of `:if-exists` is `:error`, an error is signaled. If it is `:supersede`, the file is deleted and a new file is written. If it is `:overwrite` and the file is unlocked, the data fork of the existing file is replaced with the contents of the buffer and the resource fork is modified by the font information for the buffer. If the file is locked, an error is signaled.

Using multiple fonts

Fred supports a limited multiple font capability, as outlined in Chapter 2, "Editing in Macintosh Common Lisp." In addition, you can use functions to manipulate fonts in buffers.

Font-spec information is stored with each buffer. Each character in the buffer is associated with a font spec. In addition, there is a current insertion font; characters inserted in the buffer appear in this font. (Note that the font information is actually stored as a series of ranges in the buffer; a separate font spec is not stored for each character.)

A limitation of using multiple fonts in Fred windows is that all the line heights in the buffer are the same. This line height is determined by the largest font used in the buffer. Once you add a large font, the line height is retained even if you remove all characters that appear in this font. You can restore the smaller line height by moving the text to a new buffer or by calling the function `buffer-remove-unused-fonts`.

Programming with multiple fonts introduces a new data structure, a style vector. A style vector can be applied to a series of characters in a buffer. For instance, you can use a style vector to specify that the first 10 characters after a specified position should be displayed in 12-point New York bold, the next 20 characters in 9-point Monaco, and the following 10 characters in 12-point Chicago outline. Style vectors do not automatically take note of their position: when you apply one, you have to specify a position using a buffer mark.

In addition to the functions described here, `buffer-write-file` and `buffer-insert-file` preserve and restore font information in a `'FRED'` resource.

Note that if you use another text editor to edit a file, font information may become misaligned with the text.

Functions for manipulating fonts and font styles

Use the following functions to manipulate fonts and font styles.

`buffer-char-font-spec` [*Function*]

Syntax `buffer-char-font-spec` *buffer-mark* `&optional` *position*

Description The `buffer-char-font-spec` function returns the font spec of the character at *position* in *buffer-mark* .

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default is (`buffer-position` *buffer-mark*).

buffer-current-font-spec [*Function*]

Syntax `buffer-current-font-spec` *buffer-mark*

Description The `buffer-current-font-spec` function returns the current font spec of *buffer-mark* . Any text added to the buffer is in this font.

Argument *buffer-mark* A buffer mark.

buffer-set-font-spec [*Function*]

Syntax `buffer-set-font-spec` *buffer-mark font-spec* `&optional` *start end*

Description The `buffer-set-font-spec` function sets the font spec of *buffer-mark* . If *start* is not given, `buffer-set-font-spec` sets the insertion font. Font specifications always merge with the current font.

Arguments

buffer-mark	A buffer mark.
font-spec	A font spec.
start	The start of the range in the buffer. This can be a mark or a number. The default is (`buffer-position` *buffer-mark*).
end	The end of the range in the buffer. The default is the end of the buffer.

buffer-replace-font-spec [*Function*]

Syntax `buffer-replace-font-spec` *buffer-mark old-spec new-spec*

Description The `buffer-replace-font-spec` function replaces the font specified by *old-spec* with the one specified by *new-spec* in the entire buffer and returns the font's index in the buffer's font list. If the font specified by *old-spec* is not in the buffer's font list, the function does nothing and returns `nil`.

Arguments

buffer-mark	A buffer mark.
old-spec	A font specification.
new-spec	A font specification.

Example

This function could be written as follows:

```
? (defun buffer-replace-font-spec (buf old-spec new-spec)
   (multiple-value-bind (old-ff old-ms) (font-codes old-spec)
    (multiple-value-bind (new-ff new-ms)
                         (font-codes new-spec old-ff old-ms)
      (buffer-replace-font-codes buf
         old-ff old-ms new-ff new-ms))))
```

buffer-remove-unused-fonts [*Function*]

Syntax `buffer-remove-unused-fonts` *buffer-mark*

Description The `buffer-remove-unused-fonts` function removes unused fonts from the buffer associated with *buffer-mark* .

Argument *buffer-mark* A buffer mark.

buffer-get-style [*Function*]

Syntax `buffer-get-style` *buffer-mark* `&optional` *start end*

Description The `buffer-get-style` function returns a style vector corresponding to the fonts, sizes, and styles used in the specified range in the buffer.

Arguments

buffer-mark	A buffer mark.
start	The start of the range in the buffer. This can be a mark or a number. The default is (`buffer-position` *buffer-mark*).
end	The end of the range in the buffer. The default is the end of the buffer.

buffer-set-style [*Function*]

Syntax `buffer-set-style` *buffer-mark style-vector start-position*

Description The `buffer-set-style` function sets the styles used in *buffer-mark*, beginning at *start-position*, according to *style-vector*.

Arguments

buffer-mark	A buffer mark.
style-vector	A style vector.
start-position	The beginning position in the buffer at which to insert the styles. The default is (`buffer-position` *buffer-mark*).

buffer-next-font-change [*Function*]

Syntax `buffer-next-font-change` *buffer-mark* `&optional` *position*

Description The `buffer-next-font-change` function scans the buffer of *buffer-mark* for the first font change following *position* and returns the position of the change. If there are no changes following *position*, `nil` is returned.

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default is (`buffer-position` *buffer-mark*).

buffer-previous-font-change [*Function*]

Syntax `buffer-previous-font-change` *buffer-mark* `&optional` *position*

Description The `buffer-previous-font-change` function scans the buffer of *buffer-mark* for the first font change before *position* and returns the position of the change. If there are no changes before *position*, `nil` is returned.

Arguments

buffer-mark	A buffer mark.
position	A position in the buffer. The default is (`buffer-position` *buffer-mark*).

buffer-set-font-codes [*Function*]

Syntax `buffer-set-font-codes` *buffer-mark ff ms* `&optional` *start end*

Description The `buffer-set-font-codes` function sets the font codes used in *buffer-mark*. If a range is specified by *start* and *end*, then the given range is changed. If no range is specified, then the current insertion font is changed.

Arguments

buffer-mark	A buffer mark.
ff	A font/face code. A font/face code is a 32-bit integer that combines the name of the font and its face (plain, bold, italic, and so on). See the section "Functions Relating to Font Codes" in Chapter 3, "Points and Fonts."
ms	A mode/size code. A mode/size code is a 32-bit integer that indicates the font mode (inclusive-or, exclusive-or, complemented, and so on) and the font size.
start	The initial position in the buffer to change. This should be a position or a mark. If *start* is not given, the current insertion font is changed. If *start* is given, then *end* must be given.
end	The final position in the buffer to be affected. This should be a buffer mark.

buffer-replace-font-codes [*Function*]

Syntax `buffer-replace-font-codes` *buffer-mark old-ff old-ms new-ff new-ms*

Description The `buffer-replace-font-codes` function replaces the font specified by *old-ff* and *old-ms* with the one specified by *new-ff* and *new-ms* in the owning buffer of *buffer-mark* and returns the font's index in the buffer's font list. If the font specified by *old-ff* and *old-ms* does not exist in the buffer, the function does nothing and returns `nil`.

Arguments

buffer-mark	A buffer mark.
old-ff	The old font/face code.
new-ff	The new font/face code.
old-ms	The old mode/size code.
new-ms	The new mode/size code.

Fred functions

High-level functions of the editor are defined on `fred-window` and `fred-dialog-item` through the class `fred-mixin`, which defines the general behavior of Fred.

Because `simple-view` is a subclass of `output-stream`, Fred windows are output streams. Characters output to a Fred window stream are inserted at the insertion point of the window. Stream output to Fred windows is buffered and is not displayed until `fred-update` or `force-output` is called.

Each Fred window has a buffer that is displayed in the window, a display start position that is a buffer mark indicating the first line of the buffer to be displayed in the Fred window, an insertion point that is indicated by a blinking vertical line, and a filename string indicating a file corresponding to the buffer displayed in the window. Fred dialog items also have these characteristics.

- Each Fred window must have a horizontal and vertical scroll bar and a minibuffer; most Fred dialog items do not have these.
- Fred windows may have selection ranges, which are highlighted.
- Fred dialog items (that is, all dialog items with editable text) have a size and position, a default font, one or more colors, an event handler, a key handler, and text.

The class `fred-mixin`

The class `fred-mixin` is a superclass of both `fred-window` and `fred-dialog-item`; it has no instances of its own.

`fred-mixin` [*Class name*]

Description The `fred-mixin` class mixes in Fred behavior.

This class does not have a method for `initialize-instance`. It adds the following initialization arguments used by its subclasses:

`:comtab` The command table to use with the buffer. The default is the value of `*comtab*`.

`:copy-styles-p`
An argument that determines whether to copy styles when copying. The default value is `nil`.

`:history-length`
The number of Fred commands that are remembered. Only commands that actually change text are remembered. The default value for Fred windows and Fred dialog items is `*fred-history-length*` ; for the Listener, it is `*listener-history-length*`.

Fred window and Fred dialog item functions

In addition to the functions outlined in this section, many Fred functions are associated with keystrokes. The names and actions of these functions are given in Chapter 2, “Editing in Macintosh Common Lisp.” They take one argument, either `fred-dialog-item` or `fred-window`.

Creating a Fred window

A `fred-window` is a subclass of the classes `fred-mixin` and `window`. A Fred window must contain at least an editing area (whose behavior is inherited from `fred-mixin`), two scroll bars, and a minibuffer.

`fred-window` [*Class name*]

Description The `fred-window` class is the class of Fred windows, based on `fred-mixin` and `window`.

initialize-instance [*Generic function*]

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

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

Arguments

fred-window A Fred window.

initargs A list of arguments used to initialize the Fred window. The following initialization arguments are available:

`:view-container`
The view's container. This value is set by `set-view-container`. Its initial value is `nil`.

`:view-font`
The font specification used by the view. The default is `("Geneva" 0 :PLAIN)`.

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

`:view-position`
A point, keyword, or list giving the initial position of the window. The default position is `#@(6 44)`.

`:view-size`
A point giving the initial size of the window. The default is `#@(502 150)`.

`:view-nick-name`
The nickname of the view. The default value is `nil`.

`:filename`
The name of the file to appear in the window. The default value is `nil`. If the file does not exist, an error is signaled.

`:wrap-p` An argument specifying whether text wraps in the window. The default value is `nil`.

`:view-subviews`
A list of subviews. Fred windows do not normally contain subviews.

`:window-title`
: A string specifying the title of the window. The title of a Fred window is computed from the pathname of the file displayed in it, if there is one; a window that is not displaying a file's contents has the title `New`.

`:window-show`
: An argument determining whether a window is shown when it is created. If this argument is true (the default), a window is shown when it is created. If `nil`, the window is created invisibly.

`:window-layer`
: An integer describing the layer in which the new window will be created. By default this is 0 (the front window). For details, see `set-window-layer`, defined in the section "MCL Functions for Programming Windows" in Chapter 5, "Views and Windows."

`:color-p`
: An argument specifying whether the window is a color window. If `nil` (the default), the window is based on the Macintosh window type. If non`nil`, the window is a color window.

`:window-type`
: A keyword describing the type of window to be created. The default is `:document-with-zoom`. This argument should be one of the following keywords:

```
:document
:document-with-grow
:document-with-zoom
:double-edge-box
:single-edge-box
:shadow-edge-box
:tool
```

`:copy-styles-p`
: An argument specifying whether to copy styles when copying. The default value is true.

`:procid`
: A number indicating the procID (window definition ID) of the window to be created. This is an alternative to specifying `:window-type` for programmers who want to use WDEFs with nonstandard procIDs.

`:comtab` The command table to use for editing in the buffer.

`:history-length`
The number of commands retained in the edit history. The default value is the value of `*fred-history-length*`, which is initially 20.

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

`:window-do-first-click`
A Boolean value determining whether the click that selects a window is also passed to `window-click-event-handler`. Its default value is `nil`.

`:close-box-p`
A Boolean value determining whether the window will have a close box. Close boxes aren't available on all windows.

`:wptr` For use by advanced programmers. An argument determining whether a new window is created or whether a previously existing window record is used. If the argument is not specified, `initialize-instance` calls `#_NewWindow` or `#_NewCWindow`. If the argument is specified, it should be a pointer to a window record on the Macintosh heap.

---

fred

[*Function*]

Syntax `fred &optional` *pathname new-window*

Description The `fred` function is a simpler way to create a Fred window. If *pathname* is given, `fred` attempts to open the file with that pathname.

Arguments

pathname A pathname, string, or stream associated with a file. If the file specified by *pathname* does not exist, Macintosh Common Lisp signals an error. If *pathname* is not given, Fred creates an empty Fred window.

new-window A Boolean value. If this value is `t`, Macintosh Common Lisp opens a new window, even if another window is already open to the specified file. Otherwise, Macintosh Common Lisp asks whether to open a new window or select the old one.

fred-dialog-item

[*Class name*]

Description

The `fred-dialog-item` class is the class of Fred dialog items. This class is based on `fred-mixin` and `basic-editable-text-dialog-item` (internal to `:CCL`).

Like any other dialog item, a `fred-dialog-item` can be the subview of any view.

initialize-instance

[*Generic function*]

Syntax

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

Description

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

Arguments

item A Fred dialog item.

initargs A list of arguments used to initialize the Fred dialog item. The following initialization arguments are available:

`:view-container`
The item's container. This value is set by `set-view-container`. Its initial value is `nil`.

`:view-font`
: The font in which text in the item appears. The default is `("Chicago" 12 :PLAIN)` .

`:view-position`
: The position in the dialog box where the item will be placed, in the local view coordinates. If this argument is not specified, the first available position large enough to hold the item is used. If no space is large enough, the dialog item is placed in the upper-left corner of the dialog. The default value is `nil`.

`:view-size`
: The size of the Fred dialog item. If not specified, this value is calculated so that the item fits in the view. If the specified value is too small, the item is clipped when it is drawn. The default value is `nil`.

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

`:allow-returns`
: An argument specifying whether to allow returns to be typed into the item. If `:allow-returns` is `nil` (the default), pressing the Return key while this item is the window's current key handler will invoke the window's default button, if it has one. A Return character can be inserted by pressing Shift-Return. This value is checked by the accessor `allow-returns-p` and changed by `set-allow-returns`.

`:allow-tabs`
: An argument specifying whether to allow tabs in the buffer. If `:allow-tabs` is `nil` (the default), pressing the Tab key while this item is the window's current key handler will select the next key handler. For example, pressing Tab in an editable text field will move the cursor to the next editable text field. A Tab character can be inserted by pressing Shift-Tab. This value is checked by the accessor `allow-tabs-p` and changed by `set-allow-tabs`.

`:copy-styles-p`
: An argument specifying whether to copy styles when copying. The default value is `nil`.

`:dialog-item-text`
: The default text to insert in the buffer. The default value is `""`, the empty string.

`:dialog-item-enabled-p`
: An argument specifying whether the dialog item is enabled; the default value is `t`.

`:part-color-list`
: A list of colors to which the parts of the Fred dialog item should be set. The default value is `nil`. The three possible keywords are `:frame`, the outline of the Fred dialog item; `:text`, its text; and `:body`, its body.

`:draw-outline`
: An argument specifying whether a boxed outline appears around the editable text. The default value is `t`.

`:buffer-chunk-size`
: The chunk size of the buffer. A buffer is conceptually a linked list of strings; the chunk size is the length of each of these strings. The default value for Fred dialog items is 128.

`:text-edit-sel-p`
: An argument specifying whether text can be selected. The default value is `t`.

`:comtab`
: The command table to use with the buffer. The default is the value of `*comtab*`.

Functions relating to Fred windows and Fred dialog items

The following functions manipulate Fred windows and Fred dialog items by specializing on the class `fred-mixin`.

view-mini-buffer [*Generic function*]

Syntax `view-mini-buffer` (*view* `fred-mixin`)

Description The `view-mini-buffer` generic function returns the minibuffer associated with *view* .

Argument *view* A Fred window or Fred dialog item.

Example

```
? (view-mini-buffer (make-instance 'fred-dialog-item))
#<MINI-BUFFER #x3AAF49>
```

fred-buffer [*Generic function*]

Syntax `fred-buffer` (*view* `fred-mixin`)

Description The `fred-buffer` generic function returns the buffer mark associated with the insertion point of*view* . The `window-null-event-handler` or `key-handler-idle` generic function displays the blinking vertical bar wherever this mark is located (unless the mark is off the screen, in which case no vertical bar is displayed).

Argument *view* A Fred window or Fred dialog item.

fred-chunk-size [*Generic function*]

Syntax `fred-chunk-size` (*view* `fred-mixin`)

Description The `fred-chunk-size` generic function returns the chunk size of *view,* that is, the size of each of the strings through which buffers are implemented.

Argument *view* A Fred window or Fred dialog item.

Example

```
? (fred-chunk-size (fred))
4096
```

fred-display-start-mark [*Generic function*]

Syntax `fred-display-start-mark` (*view* `fred-mixin`)

Description The `fred-display-start-mark` generic function returns the buffer mark of the first character visible in the window. By moving this mark, you affect which part of the buffer `fred-update` displays.

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

Argument *view* A Fred window or Fred dialog item.

set-fred-display-start-mark [*Generic function*]

Syntax `set-fred-display-start-mark` (*view* `fred-mixin`) *position* `&optional` *no-drawing*

Description The `set-fred-display-start-mark` generic function sets the buffer mark of the first character drawn in the window to *position*. If the value of *no-drawing* is `nil` (the default), the view is redrawn immediately. Otherwise the view is invalidated and will be redrawn the next time an `event-dispatch` occurs.

Arguments

view	A Fred window or Fred dialog item.
position	A position in the window (a mark or a number).
no-drawing	A Boolean value. The default value is `nil`.

show-cursor-p [*Variable*]

Description The *show-cursor-p* variable is bound to t by run-fred-command . It determines whether the insertion point will be made visible by run-fred-command after the command has been executed.

If the value of this variable is true, then the insertion point is made visible.

If the value of this variable is nil, *then* it is not made visible.

See the section “Fred Dispatch Sequence” later in this chapter.

window-show-cursor [*Generic function*]

Syntax window-show-cursor (*window* fred-mixin) &optional *position scrolling*

Description The window-show-cursor generic function performs an update on *view* , scrolling if necessary in order to make *position* visible.

Arguments

view	A Fred window or Fred dialog item.
position	A position to make visible in the window. The default is the insertion point.
scrolling	A Boolean value. If the value of *scrolling* is nil, no scrolling is performed.

fred-blink-position [*Generic function*]

Syntax fred-blink-position (*view* fred-mixin)

Description The fred-blink-position generic function returns the position of the parenthesis or quotation mark that matches a parenthesis or quotation mark at the insertion point. If there is no matching character, fred-blink-position returns nil.

Argument

view	A window.

fred-update

[*Generic function*]

Syntax `fred-update` (*view* `fred-mixin`)

Description The `fred-update` generic function updates the display of *view* , using the current values of the insertion point, default-position mark, selection region, and the contents of the buffer.

Note that by default, after every Fred keyboard command, Fred attempts to make the insertion point visible on the screen, repositioning the `fred-display-start-mark` if necessary. To disable this behavior for the duration of one Fred command, set the variable `*show-cursor-p*` to `nil`. Several built-in Fred commands (those that do not change the insertion point) disable this behavior.

The `fred-update` function is called automatically in several situations: in response to window update events from the Macintosh Window Manager, which occur due to resizing or uncovering a portion of the window; after the execution of every Fred keyboard command; in response to clicks in the window; and in response to `force-output` calls to the dialog item (views inherit from output streams).

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

Argument *view* A Fred window or Fred dialog item.

ed-insert-char

[*Generic function*]

Syntax `ed-insert-char` (*view* `fred-mixin`) *character*

Description The `ed-insert-char` generic function inserts *character* into the text of the Fred window or dialog item at the insertion point. The display is not automatically updated.

Arguments *view* A Fred window or Fred dialog item.
character Any character.

ed-insert-with-style [*Generic function*]

Syntax `ed-insert-with-style` (*view* `fred-mixin`) *string style* `&optional` *position*

Description The `ed-insert-with-style` generic function inserts *string* at *position* in *view*. If the value of `*paste-with-styles*` is true, it applies *style* over *string*. If the value of `*paste-with-styles*` is `nil`, it does not.

When you write functions that cause text to be pasted, use `ed-insert-with-style` to ensure consistency with the way Macintosh Common Lisp handles styles.

Arguments

view	A Fred window or Fred dialog item.
string	The string to insert in the buffer. This argument may also be `nil`, in which case nothing is inserted.
style	The style vector to map over the string after it is inserted in the buffer. This argument may also be `nil`, in which case the string is inserted in the current insertion font.
position	A position in the window (a mark or a number). The default is the window's insertion point.

fred-copy-styles-p [*Generic function*]

Syntax `fred-copy-styles-p` (*view* `fred-mixin`)

Description The `fred-copy-styles-p` generic function indicates whether or not `copy` from *view* copies styles. The default value is `nil`, meaning that it does not.

Argument

view	A Fred window or Fred dialog item.

fred-special-indent-alist [*Variable*]

Description The `*fred-special-indent-alist*` variable contains an association list of symbols that Fred should indent specially. The `car` of each pair is the symbol, and the `cdr` is the number of distinguished arguments when the symbol is used as a function, macro, or special form.

ed-current-symbol [*Function*]

Syntax `ed-current-symbol` *window* `&optional` *aux-find-symbol start end*

Description The `ed-current-symbol` function returns three values giving information about the text currently selected. If no text is selected, `ed-current-symbol` returns information on the text surrounding the cursor.

The first value is a symbol if the cursor is in an interned symbol, or if the selected text contains a symbol.

The second value is `t` if an interned symbol is found, or `nil` if an interned symbol is not found.

The third value is the character immediately before the selected text or symbol, or `nil` if the selection or symbol is at the start of the buffer.

This function attempts to locate the symbol in the package of *window*, or in the current package if the window doesn't have a package. It never has the effect of interning a new symbol.

Arguments

window A window.

aux-find-symbol A function that determines what symbol to return from the currently selected text. (It can be used, for example, to determine that the symbol is in the desired package.) It should return either the name of a function or `nil`.

start The beginning of the range to look at.

end The end of the range to look at.

ed-current-sexp

[*Generic function*]

Syntax `ed-current-sexp` (*view* `fred-mixin`) `&optional` *position dont-skip*

Description The `ed-current-sexp` generic function returns two values. If there is a current s-expression in *view* (that is, if there is an s-expression next to the insertion point, or an s-expression at *position*, or a selection), the function returns the s-expression and `t`. If there is no current s-expression, the function returns the two values `nil` and `nil`.

Arguments

view	A Fred window or Fred dialog item.
position	A position in the window (a mark or a number). The default is the window's cursor position.
dont-skip	An argument specifying whether to skip forward across reader macros. If true, the function skips across reader macros in deciding the current s-expression. If `nil`, it does not.

fred-point-position

[*Generic function*]

Syntax `fred-point-position` (*view* `fred-mixin`) *h* `&optional` *v*

Description The `fred-point-position` generic function returns the buffer position of the character nearest to the point specified by *h* and *v* in the local coordinates of the window containing the Fred dialog item. (See Chapter 3, "Points and Fonts," for a description of the point format.) This function assumes that the buffer has not been modified since the last call to `fred-update` .

Arguments

view	A Fred window or Fred dialog item.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to be an entire point in encoded form and is returned unchanged.

fred-hpos

[*Generic function*]

Syntax `fred-hpos` (*view* `fred-mixin`) `&optional` *position*

Description The `fred-hpos` generic function returns the horizontal position of the line containing *position* , in local window coordinates. The position is computed as the length (in pixels) of the line containing *position* , minus the amount of horizontal scrolling currently in effect in the window.

Arguments

view	A Fred window or Fred dialog item.
position	An integer position, or a mark in the window's buffer. The default value is the window's insertion point.

fred-vpos

[*Generic function*]

Syntax `fred-vpos` (*view* `fred-mixin`) `&optional` *position*

Description The `fred-vpos` generic function returns the vertical position of the line containing *position* , in local window coordinates. If *position* is not visible in the window, `-1` is returned.

Arguments

view	A Fred window or Fred dialog item.
position	An integer position, or a mark in the window's buffer. The default value is the window's insertion point.

fred-line-vpos

[*Generic function*]

Syntax `fred-line-vpos` (*view* `fred-mixin`) *line-number*

Description The `fred-line-vpos` generic function returns the vertical position of *line-number* in local window coordinates.

Arguments

view	A Fred window or Fred dialog item.
line-number	A line number.

fred-hscroll [*Generic function*]

Syntax `fred-hscroll` (*view* `fred-mixin`)

Description The `fred-hscroll` generic function returns the value of the desired amount of horizontal scroll in pixels in *view* .

Argument *view* A Fred window or Fred dialog item.

set-fred-hscroll [*Generic function*]

Syntax `set-fred-hscroll` (*view* `fred-mixin`) *hscroll*

Description The `set-fred-hscroll` generic function sets the value of the horizontal scroll in *view* to *hscroll* .

Arguments *view* A Fred window or Fred dialog item.

hscroll The desired amount of horizontal scroll in pixels.

selection-range [*Generic function*]

Syntax `selection-range` (*view* `fred-mixin`)

Description The `selection-range` generic function returns two values specifying the beginning and end of the currently selected text. If no text is selected, the function returns the insertion point as both values. You can use `eql` to test whether text is currently selected. If the two values are `eql`, no text is selected.

Text selected in the active window is highlighted.

Argument *view* A Fred window or Fred dialog item.

set-selection-range

[*Generic function*]

Syntax `set-selection-range` (*view* `fred-mixin`) `&optional` *position cursorpos*

Description The `set-selection-range` generic function sets the text currently selected to the buffer range between *position* and *cursorpos* and updates the window display. If *position* is equal to *cursorpos*, the selection range is made empty.

See also `select-all` in the section "Functions Implementing Standard Editing Processes" later in this chapter.

Arguments

view	A Fred window or Fred dialog item.
position	An integer position, or a mark in the window's buffer.
cursorpos	A position in the window buffer, by default the insertion point.

collapse-selection

[*Generic function*]

Syntax `collapse-selection` (*view* `fred-mixin`) *forward-p*

Description The `collapse-selection` generic function does nothing and returns `nil` if no text is selected. Otherwise, it deselects the selected text and returns `t`.

Arguments

view	A Fred window or Fred dialog item.
forward-p	An argument specifying the direction in which the cursor moves. If *forward-p* is `t`, the cursor moves forward to the end of the selected text; if it is `nil`, the cursor moves backward to the beginning of the selected text.

get-back-color

[*Generic function*]

Syntax `get-back-color` (*window* `fred-window`)

Description The `get-back-color` generic function returns the background color of *window*, encoded as an integer.

Argument

window	A Fred window.

get-fore-color

[*Generic function*]

Syntax `get-fore-color` (*window* `fred-window`)

Description The `get-fore-color` generic function returns the foreground color of *window* , encoded as an integer.

Argument *window* A Fred window.

view-font

[*Generic function*]

Syntax `view-font` (*view* `fred-mixin`)

Description The `view-font` generic function returns three values: the font spec of the current insertion font, the font spec of the character at the insertion point (or the first character in the selection), and a Boolean value. The Boolean value is `t` if the entire selection and the insertion font use the same font spec; otherwise, the Boolean value is `nil`.

Argument *view* A Fred window or Fred dialog item.

set-view-font

[*Generic function*]

Syntax `set-view-font` (*view* `fred-mixin`) *font-spec*

Description If text is selected, the `set-view-font` generic function merges the font specs of the characters in the selection with the given *font-spec* . If there are multiple font specs in the selection, they are all merged individually. If no text is selected, the current insertion font is merged with the given *font-spec* , and the result is used as the new insertion font.

Fred does not automatically set the insertion font when you move the insertion point. If the insertion font is `("Monaco" 9)` , typed characters appear in 9-point Monaco, even if you place the insertion point between characters displayed in Times Bold. When there is no selection, the insertion font must always be changed explicitly by a call to `set-view-font` . This behavior may change in future releases of Macintosh Common Lisp.

Arguments *view* A Fred window or Fred dialog item.

font-spec A font spec.

window-set-not-modified [*Generic function*]

Syntax `window-set-not-modified` (*view* `fred-mixin`)

Description The `window-set-not-modified` generic function is called by the window system when a Fred window is saved to a file. It sets the state of the window to be not modified and calls `fred-update`.

Argument *view* A Fred window or Fred dialog item.

window-filename [*Generic function*]

Syntax `window-filename` (*window* `fred-window`)

Description The `window-filename` generic function returns the pathname of the file associated with the Fred window. If no pathname is associated with the window, `window-filename` returns `nil`.

Files become associated with Fred windows when `set-window-filename` is called or if the window was created with an initialization argument for `:filename`.

Argument *window* A Fred window.

set-window-filename [*Generic function*]

Syntax `set-window-filename` (*window* `fred-window`) *new-name*

Description The `set-window-filename` generic function sets the filename associated with the Fred window. When the window contents are saved, they are saved to the new filename. If a file corresponding to the filename already exists, it is overwritten without warning when the window is next saved.

Arguments *window* A Fred window.
new-name A pathname or string giving the file to associate with the window.

fred-package

[*Generic function*]

Syntax `fred-package` (*view* `fred-mixin`)

Description The `fred-package` generic function returns the package associated with the window containing the item or `nil`. If `nil`, the window's package is always the current value of `*package*` .

Argument *view* A Fred window or Fred dialog item.

set-fred-package

[*Generic function*]

Syntax `set-fred-package` (*view* `fred-mixin`) *package*

Description The `set-fred-package` generic function sets the package associated with the window containing the Fred dialog item. The *package* argument may be a package, or it may be a string or symbol naming a package.

Arguments *view* A Fred window or Fred dialog item.
package A package indicator (that is, a string or a symbol naming a package, or a package object).

fred-margin

[*Generic function*]

Syntax `fred-margin` (*view* `fred-mixin`)

Description The `fred-margin` generic function returns the distance in pixels between the left edge of *view* and the left edge of the first character on each line.

Argument *view* A Fred window or Fred dialog item.

set-fred-margin

[*Generic function*]

Syntax `set-fred-margin` (*view* `fred-mixin`) *new-margin*

Description The `set-fred-margin` generic function sets the distance in pixels between the left edge of *view* and the left edge of the first character on each line to *new-margin* .

Arguments

view	A Fred window or Fred dialog item.
new-margin	A fixnum specifying the width of the margin in pixels. If *new-margin* is not a fixnum, an error is signaled.

fred-tabcount

[*Generic function*]

Syntax `fred-tabcount` (*window* `fred-window`)

Description The `fred-tabcount` generic function returns the number of spaces per tab in *window* .

Argument *window* A Fred window.

fred-wrap-p

[*Generic function*]

Syntax `fred-wrap-p` (*window* `fred-mixin`)

Description The `fred-wrap-p` generic function returns a Boolean value: `t` if lines wrap in *window* , `nil` if they do not.

Argument *window* A Fred window.

Example

The following code defines a Fred command that toggles whether text is wrapped in a window. The function `ed-refresh-screen` used in this example is defined in the file `assorted-fred-commands.lisp` in your MCL Examples folder.

```
(def-fred-command (:control :meta #\w)
  (lambda (w)
    (setf (fred-wrap-p w) (not (fred-wrap-p w)))
    (ed-refresh-screen w)))
```

window-save [*Generic function*]

Syntax `window-save` (*window* `fred-mixin`)

Description The `window-save` generic function saves the window to its disk file. If the window has no filename, `window-save-as` is called.

Argument *window* A Fred window.

window-save-as [*Generic function*]

Syntax `window-save-as` (*window* `fred-mixin`)

Description The `window-save-as` generic function calls the standard `SfPutFile` dialog box, allowing the user to choose a directory and input a filename, and saves the contents of the window to the filename.

Note that if the user clicks Cancel in this dialog box, Macintosh Common Lisp throws to `:cancel`. User code may wish to perform a `catch-cancel` to prevent a return to the top level. (The macro `catch-cancel` is documented in Chapter 6, "Dialog Items and Dialogs.")

Argument *window* A Fred window.

window-revert [*Generic function*]

Syntax `window-revert` (*window* `fred-mixin`) `&optional` *dont-prompt*

Description The `window-revert` generic function causes the window to revert to the last version saved.

Arguments *window* A Fred window.

dont-prompt A Boolean value. If the value is `nil` (the default), the user is asked to confirm the reversion before it is performed. If the value of this parameter is true, the reversion is performed without asking the user.

window-hardcopy

[*Generic function*]

Syntax `window-hardcopy` (*window* `fred-window`)`&optional` *show-dialog*

Description The `window-hardcopy` generic function sends the contents of the window or dialog item to the current printer. Before printing takes place, the user is prompted to specify various printer options.

To insert a hard page break in printouts, press Control-Q Control-L (the quoted form feed character). The character appears as a square box in MCL windows.

Arguments

window	A Fred window or Fred dialog item.
show-dialog	A Boolean value. If this value is true (the default), MCL presents a print job dialog to the user. Otherwise, MCL uses the values entered last time, or the default values if no print job dialog has been shown yet.

ed-beep

[*Function*]

Syntax `ed-beep &rest` *the-rest*

Description The `ed-beep` function sounds a beep. It returns `nil`.

Argument

the-rest	A rest argument; ignored.

Functions implementing standard editing processes

The functions that are found in the standard Macintosh editing menu are implemented as MCL generic functions. In addition, Macintosh Common Lisp provides a more extensive Undo facility.

cut [*Generic function*]

copy [*Generic function*]

paste [*Generic function*]

clear [*Generic function*]

undo [*Generic function*]

undo-more [*Generic function*]

select-all [*Generic function*]

Syntax

`cut` (*view* `fred-mixin`)
`copy` (*view* `fred-mixin`)
`paste` (*view* `fred-mixin`)
`clear` (*view* `fred-mixin`)
`undo` (*view* `fred-mixin`)
`undo-more` (*view* `fred-mixin`)
`select-all` (*view* `fred-mixin`)

Description

These generic functions are each specialized on the `fred-mixin` class (as well as on `window`; see Chapter 5, "Views and Windows").

The `cut` generic function deletes the currently selected text from the buffer and stores it in the Clipboard and the kill ring.

The `copy` generic function adds the currently selected text to the Clipboard and pushes it onto the kill ring. The selection is not removed from the window or dialog item.

The `paste` generic function replaces the currently selected text with the text in the Clipboard. If no text is selected, the text in the Clipboard is inserted at the insertion point.

The `clear` generic function deletes the currently selected text from the buffer without storing it in the Clipboard or the kill ring.

The `undo` generic function undoes the most recent edit if it can be undone.

The `undo-more` generic function undoes edits earlier in the edit history if they can be undone.

The `select-all` generic function makes the entire contents of the buffer the currently selected text.

Argument

view A Fred window or Fred dialog item.

window-can-do-operation [*Generic function*]

Syntax `window-can-do-operation` (*view* `fred-mixin`) *operation* `&optional` *menu-item*

Description The `window-can-do-operation` generic function is called to determine whether an Edit menu item should be enabled. It returns a Boolean value indicating whether *view* can perform *operation*. (This is a more general replacement for the older MCL function `window-can-undo-p` , which could check only for Undo.)

Arguments

view	A Fred window or Fred dialog item.
operation	A symbol indicating one of the standard editing operations: `cut`, `clear` , `copy`, `paste`, `select-all` , `undo`, or `undo-more` .
menu-item	The corresponding Edit menu item.

Example

The following code indicates that `*top-listener*` contains a selection that can be cut.

```
? (window-can-do-operation *top-listener* 'cut)
T
```

Multiple-level Undo

To support Undo, Fred buffers keep a history list of all changes that have been made since the buffer was created (or up to a user-definable limit imposed by `*fred-history-length*`). When an Undo command is issued, the most recent command on the history list is undone. Repeatedly issuing Undo commands undoes earlier and earlier changes, back to the initial state of the buffer or to the limit imposed by `*fred-history-length*` .

A single Fred command may involve several insertions and deletions. In the functions that relate to Undo, the argument *append-p* indicates that an insertion or a deletion is part of the same operation as the previous insertion or deletion.

Successive adjacent deletions or insertions, as well as multiple replacements via the Search dialog, are considered a single command.

This Undo history is maintained on a buffer-by-buffer basis. The number of commands saved for each Fred buffer is under user control.

Functions relating to Undo

The following functions support Undo in Macintosh Common Lisp. (See also the section "Undo Commands" in Chapter 2, "Editing in Macintosh Common Lisp.")

ed-delete-with-undo [*Generic function*]

Syntax `ed-delete-with-undo` (*view* `fred-mixin`) *start end* `&optional` *save-p reverse-p append-p*

Description The `ed-delete-with-undo` generic function deletes the range in *view* 's buffer specified by *start* and *end* and saves the deletion on the kill ring and in the history list of the buffer of *view* . It returns a `cons` with the string of the deleted range in the `car` and the style vector of the deleted range in the `cdr`.

The `ed-delete-with-undo` generic function works with the `*last-command*` variable (described in "MCL Expressions Associated With Keystrokes" later in this chapter) to concatenate successive deletions. If this function is called repeatedly, it concatenates the deleted text into a single item on the kill ring rather than creating several items on the kill ring. Deleting with `ed-delete-with-undo` sets the last command to `:kill`.

All Fred commands that delete text call `ed-delete-with-undo` .

Arguments

view	A Fred window or Fred dialog item.
start	The start of the range to delete. This argument may be an integer or a buffer mark.
end	The end of the range to delete. This argument may be an integer or a buffer mark.

save-p	An argument specifying what to do with deleted text. If the value of this argument is true (the default), the deleted text is added to the kill ring. If it is `nil`, the deleted text is added to the kill ring only if it is nontrivial. Nontrivial strings are those that contain both word-forming characters and word-delimiting characters. The justification is that text that is explicitly deleted should always be saved (therefore the value of *save-p* is true by default). Text killed incidentally, for instance, when the user types over selected text, is saved only when it is complicated.
reverse-p	An argument specifying the direction of the deletion. The value of this argument is true if the text is killed by a backward deletion.
append-p	An argument specifying whether this deletion is part of the same operation as the previous insertion, deletion, or replacement. If true, it is.

`ed-insert-with-undo` [*Function*]

Syntax `ed-insert-with-undo` *view string* `&optional` *position append-p*

Description The `ed-insert-with-undo` function inserts *string* in *view* at *position* and saves *string* in the history list of the buffer of *view*. If *string* is to be appended to a previous Undo command, *append-p* is true.

Arguments

view	A Fred window or Fred dialog item.
string	Either a string, or a `cons` of a string and a style.
position	A position in the view. The default is the insertion point.
append-p	An argument specifying whether this insertion is part of the same operation as the previous insertion, deletion, or replacement. If true, it is.

ed-replace-with-undo [*Function*]

Syntax `ed-replace-with-undo` *view start end string* `&optional` *append-p*

Description The `ed-replace-with-undo` function replaces the range of characters from *start* to *end* in *view* with *string* and saves the replaced range in the history list of the buffer of *view* .

Arguments

view	A Fred window or Fred dialog item.
start	The start of the range to replace. This argument may be an integer or a buffer mark.
end	The end of the range to replace. This argument may be an integer or a buffer mark.
string	Either a string, or a `cons` of a string and a style.
append-p	An argument specifying whether this replacement is part of the same operation as the previous insertion, deletion, or replacement. If true, it is.

set-fred-undo-string [*Function*]

Syntax `set-fred-undo-string` *fred-window string* `&optional` *undo-redo*

Description The `set-fred-undo-string` function sets the suffix of the Undo menu title to *string* in *fred-window*. For example, if *string* is `"Typing"` , the Undo menu item title becomes Undo Typing or Redo Typing.

Arguments

fred-window	A Fred window.
string	A string.
undo-redo	A value, either `:UNDO` or `:REDO` . If it is `:REDO` , then the name of the Undo menu item is Redo *string* (for example, Redo Typing). If it is `:UNDO` (the default), the name of the Undo menu item is Undo *string*.

setup-undo

[*Generic function*]

Syntax `setup-undo` (*view* `fred-mixin`) *function* `&optional` *string*

Description The `setup-undo` generic function allows Fred commands to support the Undo menu item. Any Fred action that can be undone in a way that is not supported by `ed-insert-with-undo` , `ed-replace-with-undo` , or `ed-delete-with-undo` should call `setup-undo` . The *function* argument should be a function to call when Undo is chosen. If given, *string* should be a short string to be used as the title of the menu item.

Arguments

view	A Fred window or Fred dialog item.
function	The function to call when the Undo menu item is chosen.
string	A string serving as the title of the Undo menu item. The default is “Undo”.

Examples

Here is an example of how to enable Undo and Redo. The function `insert-hello` inserts the string `"hello"` . It supports Undo and Redo.

```
? (defun insert-hello (window)
          (let* ((buf (fred-buffer window))
                 (start-pos (buffer-position buf)))
            (buffer-insert buf "hello" start-pos)
            (fred-update window)
            (setup-undo window
                        #'(lambda
                            ()
                           (buffer-delete buf start-pos)
                           (fred-update window)
                           (setup-undo window
                            #'(lambda ()
                               (insert-hello window)
                               (fred-update window))
                            "Redo Hello"))
             "Undo Hello")))
INSERT-HELLO
```

This example shows how to do the same thing more simply.

```
? (defun alternative-insert-hello (window)
          (ed-insert-with-undo window "hello")
          (set-fred-undo-string window "Hello")
          (fred-update window))
ALTERNATIVE-INSERT-HELLO
```

setup-undo-with-args

[*Generic function*]

Syntax `setup-undo-with-args` (*view* `fred-mixin`) *function arg* `&optional` *string*

Description The `setup-undo-with-args` generic function works like `setup-undo` except that the *function* must take two arguments; that is, it will be called via (`funcall` *function view arg*). The argument is frequently a position.

Arguments

view	A Fred window or Fred dialog item.
function	The function to call when the Undo menu item is chosen.
arg	An argument to pass the function.
string	A string serving as a modifier to the title of the Undo menu item. For example, if the string is `"Typing"` , the Undo menu item title is Undo Typing or Redo Typing.

Working with the kill ring

The kill ring is a circular list containing text that has been deleted from Fred windows or dialog items. Each item in the kill ring is a `cons` cell. The `car` of the `cons` contains a string. The `cdr` of the `cons` contains either a style vector or `nil` (style vectors are described in the section “Using Multiple Fonts” earlier in this chapter).

Functions for working with the kill ring

The following functions work with the kill ring.

ed-kill-selection [*Generic function*]

Syntax `ed-kill-selection` (*view* `fred-mixin`)

Description The `ed-kill-selection` generic function deletes the currently selected text by calling `ed-delete-with-undo` . The deleted text is saved only if it is nontrivial (see `ed-delete-with-undo` in the section "Functions Relating to Undo" earlier in this chapter for a definition of triviality).

Commands that insert text generally call this function before performing the insertion.

Argument *view* A Fred window or Fred dialog item.

add-to-killed-strings [*Function*]

Syntax `add-to-killed-strings` *string-style-cons*

Description The `add-to-killed-strings` function rotates the kill ring and pushes *string-style-cons* to the front of the kill ring.

Argument *string-style-cons*
A `cons` whose `car` is a string and whose `cdr` is a style vector or `nil`.

rotate-killed-strings [*Function*]

Syntax `rotate-killed-strings &optional` *count*

Description The `rotate-killed-strings` function rotates the kill ring. The third item becomes the second, the second item becomes the first, and the first becomes the last. (Remember, the kill ring is a circular list.)

Any empty items are automatically skipped.

Argument *count* An integer to be added to the normal number by which the kill ring is rotated. For example, if *count* is 0, the kill ring is rotated by 1; if it is 4, the kill ring is rotated by 5. The default value of *count* is 0.

Using the minibuffer

The minibuffer provides a convenient method for showing information to users of Fred windows. The information can be the result of a command, a progress indicator, a request for further information, or some combination of all these. All Fred windows display the window's package in the lower-left corner of the window. This is not considered part of the minibuffer.

Each instance of `fred-window` has its own minibuffer, an instance of the class `mini-buffer` that is accessed with `view-mini-buffer` . Minibuffers are output streams with some additional features.

The variable `*clear-mini-buffer*` specifies whether to clear the minibuffer after each Fred command.

Some of the following generic functions are associated with methods for Fred windows and some with methods for minibuffers.

Because minibuffers are streams, you can use them as the first argument to `format` . Sending a newline will clear the minibuffer.

Functions for working with the minibuffer

The following functions define minibuffers.

`mini-buffer` [*Class name*]

Description The `mini-buffer` class is the class of minibuffers, built on `output-stream` . A minibuffer displays information about its containing Fred window.

view-mini-buffer [*Generic function*]

Syntax `view-mini-buffer` (*view* `fred-mixin`)

Description The `view-mini-buffer` generic function returns the minibuffer of the window or dialog item.

Argument *view* A Fred window or Fred dialog item.

set-mini-buffer [*Generic function*]

Syntax `set-mini-buffer` (*view* `fred-mixin`) *string* `&rest` *format-args*

Description The `set-mini-buffer` generic function clears the text of the minibuffer, applies `#'format` to the minibuffer, *string*, and *format-args*, and then performs a minibuffer update to display the minibuffer.

The minibuffer shows only the last line printed. Sending a newline clears the minibuffer.

Arguments

view	A Fred window or Fred dialog item.
string	A format control string, suitable for passing as the second argument to `format`.
format-args	A set of *format* arguments, suitable for passing to `format` along with *format-string*.

mini-buffer-update [*Generic function*]

Syntax `mini-buffer-update` (*view* `fred-mixin`)

Description The `mini-buffer-update` generic function draws the contents of the minibuffer. This function is normally called whenever the window is updated. You need to call it explicitly whenever you print to a minibuffer and wish to show its contents.

Argument *view* A Fred window or Fred dialog item.

stream-column [*Generic function*]

Syntax `stream-column` (*stream* `mini-buffer`)

Description The `stream-column` generic function returns the length of the text displayed in the minibuffer.

Argument *stream* A minibuffer stream.

mini-buffer-string [*Generic function*]

Syntax `mini-buffer-string` (*minibuffer* `mini-buffer`)

Description The `mini-buffer-string` generic function returns the contents of the minibuffer as a string. This string is a vector with a fill pointer. The `stream-tyo` method on minibuffers works by pushing characters onto this string. Sending a newlineto the minibuffer sets the fill pointer to 0. The `stream-column` function returns the value of the fill pointer.

Argument *minibuffer* A minibuffer.

Defining Fred commands

Besides the standard Fred commands, described in Chapter 2, "Editing in Macintosh Common Lisp," you can program your own commands. You may wish to create your own tables of commands (discussed in the next section, "Fred Command Tables"). The following macro defines a Fred command in the currently active command table.

def-fred-command [*Macro*]

Syntax `def-fred-command` *keystroke function* `&optional` *doc-string*

Description The `def-fred-command` macro is equivalent to`(comtab-set-key *comtab* 'keystroke 'function doc-string)` . (See the next section, "Fred Command Tables," for more information.)

Arguments	*keystroke*	A keystroke code or keystroke name.
	function	A function to be called when *keystroke* is typed. Because this argument is not evaluated, it should usually be a symbol naming a function.
	doc-string	A documentation string describing the action of *keystroke* .

Example

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

Fred command tables

The following system is used to translate keystroke events into Fred actions.

Keystroke codes and keystroke names

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

In normal operation, keystrokes are handled by the active window. Fred treats every keystroke typed in a Fred window as a command. Associated with every possible keystroke is a Lisp function that implements the command. Some commands are simple; for example, pressing A is a command to insert A in the buffer. Some are more complex; for example, pressing Control-Meta-Shift-F is a command to select one s expression forward. Fred makes no distinction between these two kinds of commands. Indeed, you can easily redefine A to perform a complicated series of actions.

When you press a key in a Fred window, Fred first translates it into a *keystroke code* . The keystroke code contains a character and four flags called *meta, control, function,* and *shift* . The keystroke is encoded as a small integer, with the character code in bits 0 through 7, the meta flag in bit 8, the control flag in bit 9, the function flag in bit 10, and the shift flag in bit 11. See Table 14-1.

n **Table 14-1** Modifier bits in the keystroke code

Bit	Value	Keyword	Keyboard
8	#x100	:meta	Option
9	#x200	:control	Control or Command
10	#x400	:function	One of the 15 function keys
11	#x800	:shift	Shift

In Fred programming, keystrokes are usually described in terms of *keystroke names* rather than keystroke codes. A keystroke name is either a character or a list containing a character and zero or more modifier keywords. The modifier keywords are `:control`, `:shift`, `:meta`, and `:function`. Examples of legal keystroke names are `(:control #\a)`, `(:meta #\f)`, `(:function #\1)`, `#\a`, and `(:control :meta #\x)`.

The `:function` modifier is used to support the function keys on the Apple Extended Keyboard. The function keys are named using the characters `#\1` through `#\9` and `#\a` through `#\f`. For example, the F1 key has the keystroke name `(:function #\1)`, the F10 key has the name `(:function #\a)`, and the F15 key has the name `(:function #\f)`.

The functions `event-keystroke` and `keystroke-code` return the keystroke code.

The function `keystroke-code` can return any combination of character codes and modifier bits, but `event-keystroke` returns only a subset:

- The function bit is set only with character codes of 49–57 or 65–70 (the digits 1–9 and the letters A–F).
- The shift bit is set only with character codes representing non graphic characters or alphabetic characters combined with the Option or Control key.

You should note that the keystroke code for a letter with the shift bit set is not equal to the keystroke code for a shifted letter. That is:

```
? (keystroke-code '(#\A))
65
? (keystroke-code '(:shift #\A))
2113
```

Command tables

The binding between keystrokes and the functions they invoke is stored in a data structure called a *comtab* (short for *command table*). The global command table is stored in the variable `*comtab*`. Each window with `fred-mixin` may contain a local command table in a slot named `comtab`, the default value of which is `*comtab*`. In addition, each window may also contain a shadowing command table, which is initially `nil`.

Fred dispatch sequence

The `view-key-event-handler` method for `fred-mixin` performs the following sequence of events to process a keystroke.

When Fred receives a keyboard event, it binds the variable `*current-character*` to the character typed. It uses the function `event-keystroke` to translate the event to a keystroke code and binds `*current-keystroke*` to the keystroke code.

It then checks the variable `*fred-keystroke-hook*`, which can be a function, a command table, or `nil`. If it is a function, the function is run and is responsible for keystroke processing. If it is a command table, the keystroke is looked up in the command table. If it is `nil`, the keystroke is looked up in the `shadowing-comtab` or `comtab` of the Fred window or Fred dialog item. The keystroke look-up is performed by the generic function `keystroke-function`. If `*fred-keystroke-hook*` is `nil`, the keystroke is processed by the function `run-fred-command`.

When the function associated with the keystroke returns, Fred updates the display of the window on the screen, making sure the cursor is visible. (The function may set the variable `*show-cursor-p*` to `nil` to inhibit this.)

MCL expressions associated with keystrokes

The following functions control and report on the behavior of keystrokes.

event-keystroke [*Function*]

Syntax `event-keystroke` *message modifier*

Description The `event-keystroke` function takes the *message* and *modifier* fields of a Macintosh event record and returns a keystroke code. It sets the control bit if the Control key was pressed and the meta bit if the Option key was pressed. It sets the shift bit if the Shift key was pressed and the character either is not graphic or is alphabetic and the Control or Option key was also pressed. The Caps Lock key is ignored. The character portion of the keystroke code is set to the ASCII code in the *message* field, except when Option is pressed. If Option is pressed, the character portion is set to the character used to generate the keystroke (for instance, the code for Option-S is `#\s` rather than `#\ß`).

Fred calls this function when it receives a key-down event.

Arguments

message	The message field of an event record.
modifier	The modifier field of an event record.

keystroke-name [*Function*]

Syntax `keystroke-name` *keystroke-code*

Description The `keystroke-name` function returns the name of a keystroke code.

A keystroke name is either a character or a list, for example, `(:control :meta` *character* `)`, `(:function` *character* `)`, or `(:shift :meta` *character* `)`.

Argument

keystroke-code	Any valid keystroke code.

keystroke-code [*Function*]

Syntax `keystroke-code` *keystroke-name*

Description The `keystroke-code` function translates a keystroke name to a keystroke code.

Argument *keystroke-name*

A keystroke name. A keystroke name is either a character or a list, for example, `(:control :meta` *character* `)`, `(:function` *character* `)`, or `(:shift :meta` *character* `)`. The *keystroke-name* argument may also be a keystroke code (an integer), in which case it is simply returned.

Example

```
? (keystroke-code '(:shift :meta #\F))
2406
? (keystroke-name 2406)
(:SHIFT :META #\F)
```

keystroke-function

[*Generic function*]

Syntax `keystroke-function` (*view* `fred-mixin`) *keystroke* `&optional` *comtab*

Description The `keystroke-function` generic function performs the full Fred command look-up for *keystroke*. It always returns a function or a command table, never `nil` or another keystroke.

It first looks up the keystroke in *comtab*, or if *comtab* is unspecified or `nil`, in the shadowing command table of *view*, if there is one; otherwise it looks in the command table of *view*. If the definition is another keystroke, that keystroke is looked up. Circularity will be detected.

Arguments

view	A Fred window or Fred dialog item.
keystroke	A keystroke name or keystroke code.
comtab	A command table.

fred-keystroke-hook

[*Variable*]

Description The *fred-keystroke-hook* variable provides a hook into the Fred command dispatch process.

If this variable is a function, the function is called with one argument, the Fred window or dialog item, to do the keystroke processing. If it is a command table, the keystroke is looked up in the command table. If it is nil, the keystroke is looked up in the shadowing command table or command table of the Fred window or Fred dialog item.

The keystroke look-up is performed by the generic function keystroke-function .

last-command

[*Variable*]

Description The *last-command* variable is bound by run-fred-command to the value saved by the last command. Thus if a Fred command does not set the last command with set-fred-last-command , the value of *last-command* is nil when the next command runs.

This information is useful when one command needs to know what the previous one was. For example, repeatedly calling ed-yank-pop (Meta-Y) inserts successive strings from the kill ring into a window. For this to work, each call to ed-yank-pop needs to know whether the last Fred command was also ed-yank-pop .

The user should never set *last-command* explicitly; use set-fred-last-command instead.

fred-last-command

[*Generic function*]

Syntax fred-last-command (*view* fred-mixin)

Description The fred-last-command generic function returns the most recent Fred command.

Argument *view* A Fred window or Fred dialog item.

set-fred-last-command [*Generic function*]

Syntax `set-fred-last-command` (*view* `fred-mixin`) *new-last-command*

Description The generic function `set-fred-last-command` sets the last command of *view* to *new-last-command*.

Always use `set-fred-last-command` to set the value of `*last-command*`; do not set it directly.

Arguments

view A Fred window or Fred dialog item.

new-last-command A command to which to set `*last-command*`.

current-character [*Variable*]

Description The `*current-character*` variable is bound to the character of the current keystroke during the execution of Fred commands. This variable is used by functions such as `ed-self-insert`.

current-keystroke [*Variable*]

Description The `*current-keystroke*` variable is bound to the current keystroke during the execution of Fred commands.

ed-self-insert [*Generic function*]

Syntax `ed-self-insert` (*view* `fred-mixin`)

Description The `ed-self-insert` generic function inserts `*current-character*` into the window. You should call this function only from within Fred commands (at which time `*current-character*` is sure to be bound). The function `ed-self-insert` checks for a numeric prefix, such as Control-U, that tells it how many times to execute itself.

Argument

view A Fred window or Fred dialog item.

MCL expressions relating to command tables

The following MCL expressions are used to create and govern command tables.

comtab

[*Variable*]

Description The `*comtab*` variable is the global command table. You can modify this command table or set the variable to a new command table.

listener-comtab

[*Variable*]

Description The `*listener-comtab*` variable contains the initial form of `comtab` for the class `listener` . Whenever a new Listener is created, its command table is initially the value of this variable.

By modifying `*listener-comtab*` , you can change the behavior of the Listener without affecting other Fred windows.

Note that setting this variable to a new command table (as opposed to modifying the command table it is set to) affects only those Listeners created after the change.

control-x-comtab

[*Variable*]

Description The `*control-x-comtab*` variable contains the command table used by the Control-X keystroke.

make-comtab

[*Function*]

Syntax `make-comtab &optional` *default*

Description The `make-comtab` function returns a command table, with *default* as the means to process all keystrokes.

Argument *default* A default value. If *default* is a command table, when a keystroke is looked up in the new command table, Macintosh Common Lisp looks in *default*. If *default* is `nil`, it defaults to the value of `*comtab*`. If it is anything else, for example, a function, *default* is expected to process a keystroke.

---

copy-comtab [*Function*]

Syntax `copy-comtab &optional` *source-comtab*

Description The `copy-comtab` function returns a new command table that is initially functionally equivalent to *source-comtab*.

Argument *source-comtab* A command table or `nil`. If *source-comtab* is specified as `nil`, it returns a copy of the command table that was in use when Macintosh Common Lisp was launched. This is useful if you have somehow corrupted the current command table.

---

comtabp [*Function*]

Syntax `comtabp` *thing*

Description The `comtabp` function returns true if *thing* is a command table; otherwise, it returns `nil`.

Argument *thing* Any Lisp object.

---

comtab-set-key [*Function*]

Syntax `comtab-set-key` *comtab keystroke function* `&optional` *doc-string*

Description The `comtab-set-key` function sets the definition of *keystroke* to *function* within *comtab*.

Arguments

comtab	A command table.
keystroke	A keystroke.
function	A function to be called when *keystroke* is pressed. The function may be any of the following: n A symbol, a compiled function, or a lambda expression, indicating a function to call when *keystroke* is entered. n A command table, indicating that the keystroke is a prefix character, such as Control-X, that reads another character and looks it up in its own command table. n Another keystroke (name or code) to indicate that *keystroke* should do whatever the other keystroke would do. n The value `nil`, which causes the command table's default function to process the keystroke.
doc-string	A documentation string describing the action of *keystroke* .

Example

For example, the following form binds the F12 key to a command that prints the date in the top window:

```
? (comtab-set-key *comtab* '(:function #\c)  ;F12 key
                  #'(lambda
                      (d)
                     (multiple-value-bind
                      (second minutes hour date month year)
                      (get-decoded-time)
                      (declare (ignore second minutes hour))
                      (format d "~a/~a/~a"
                       month
                       date
                       (- year 1900))))
   "print the date in the top window")
#<COMTAB #x2F9E99>
```

comtab-get-key

[*Function*]

Syntax `comtab-get-key` *comtab keystroke*

Description The `comtab-get-key` function looks up the definition of *keystroke* in *comtab*. This function is the reverse of `comtab-set-key` .

Arguments

comtab	A command table. The value of *comtab* may be a symbol, a compiled function, a command table, another keystroke, or `nil`.
keystroke	A keystroke code or keystroke name.

Example

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

comtab-key-documentation [*Function*]

Syntax `comtab-key-documentation` *comtab keystroke*

Description The `comtab-key-documentation` function returns the documentation string associated with *keystroke* .

Arguments

comtab	A command table. The value of *comtab* may be a symbol, a compiled function, a command table, another keystroke, or `nil`.
keystroke	A keystroke code or keystroke name.

Example

```
? (comtab-key-documentation *comtab* '(:function #\c))
"print the date in the top window"
```

comtab-find-keys [*Function*]

Syntax `comtab-find-keys` *comtab function*

Description The `comtab-find-keys` function returns a list of all keystrokes bound to *function* in *comtab* .

Arguments

comtab	A command table. The value of *comtab* may be a symbol, a compiled function, a command table, another keystroke, or `nil`.
function	A function. The function symbol must be quoted (not #") .

Example

```
? (comtab-find-keys *comtab* 'ed-open-line)
(623)
```

Chapter 15 **Foreign Function Interface**

Contents

Using the Foreign Function Interface / 518
High-level Foreign Function Interface operations / 518
    Argument specifications / 523
    Result flags / 526
    Short example / 527
Low-level functions / 528
Calling Macintosh Common Lisp from foreign functions / 532
Extended example / 533

This chapter describes Macintosh Common Lisp's Foreign Function Interface (FFI), which permits calls from within Macintosh Common Lisp to functions written in C, Pascal, assembly language, and other languages. Such functions are called foreign functions. Foreign functions can in turn make calls back to Macintosh Common Lisp.

This chapter describes both high-level and low-level foreign function calls. To take advantage of the high-level functions described in this chapter, the foreign function code must be compiled into an MPW (Macintosh Programmer's Workshop) object file format. Any compiler that produces object files of the MPW format may be used. Machine code produced by other compilers or assemblers can be accessed only with the low-level function `ff-call`.

You should read this chapter if you plan to include calls to foreign functions within Macintosh Common Lisp.

You should also be familiar with MPW object files, which are documented in *MPW: Macintosh Programmer's Workshop Development Environment* .

Using the Foreign Function Interface

To use foreign functions from Macintosh Common Lisp, do the following:

- Write and compile the foreign functions using a compiler that produces MPW object files.
- Run Macintosh Common Lisp and load the Foreign Function Interface files.
- Load the MPW object files with the function `ff-load`.
- Define an interface for each foreign function you wish to call. (This is done with `defffun`, `deffcfun`, or `deffpfun`.)
- Call the foreign functions from Macintosh Common Lisp using MCL syntax.

A call from Macintosh Common Lisp to a foreign function looks exactly like a call to another MCL function. The MCL function that makes the call (and, for that matter, the programmer) doesn't even need to know that the function called was written in a different language.

u *Note:* In the current version of Macintosh Common Lisp, Object Pascal foreign functions are not called correctly. Functions produced by Fortran compilers may not be called correctly.

To use the Foreign Function Interface you must load the file `ff.fasl`, included in the MCL Library folder, or evaluate the expression

? **`(require "ff")`**

High-level Foreign Function Interface operations

The following high-level operations are used with the Foreign Function Interface. They can be used only on object files in the MPW object file format.

ff-load

[*Function*]

Syntax

```
ff-load files &key :entry-names :libraries :library -
entry-names :ffenv-name :replace
```

Description

The `ff-load` function loads the MPW object files specified by *files* and returns a foreign function environment.

The foreign function environment returned consists of code segments, a jump table, a static data area, and a collection of active entry point names. Dead code is removed so that only code and data reachable from the active entry points are included in the environment.

Each call to `ff-load` produces a distinct foreign function environment, with its own global space, function code, and so on. There is no sharing of code or data between environments produced by separate calls to `ff-load`. For example, if two different calls to `ff-load` require a library function `atoi`, they will each get their own copy of `atoi`. If they each refer to a global variable `errno`, they will get their own copy of `errno`. To share data and library code between routines, you must link the routines in a single call to `ff-load`.

Arguments

files A filename, a pathname, or a list of filenames and pathnames of MPW object files.

`:entry-names` A list of strings naming all the entry points in *files* that should be active. If `:entry-names` is not specified or its value is `nil`, all entry points in *files* are active. Note that these strings are case sensitive.

`:libraries` A list of additional object files to load. These differ from the files in *files* in that, by default, entry points in libraries are not considered active (so that code from libraries is not included in the link unless the code is needed by other functions).

`:library-entry-names` A list of active entry point names in libraries. This overrides the default for libraries of only including those entry points used by other functions.

`:ffenv-name`	A symbol. If `:ffenv-name` is given and its value is not `nil`, `ff-load` checks to see whether an environment with the given name is already loaded. If so, the action taken depends on the value of the `:replace` argument (described next). If not, the specified files are loaded and the resulting environment is given the name passed in this argument. This argument can be used to make `ff-load` behave somewhat like `require`.
`:replace`	If `:replace` is given and its value is not `nil`, the files are always loaded and any previously loaded environment of the same name (as specified by the `:ffenv-name` argument) is disposed of. (The previously existing environment is disposed of only if the loading is successful.)

Example

```
(setf (logical-pathname-translations "mpw")
      '(("clib;**;*.*" "hard disk:mpw:libraries:clibraries:**:*.*")
        ("lib;**;*.*"   "hard disk:mpw:libraries:libraries:**.*.*")
        ("**;*.*" "hard disk:mpw:**:*.*")))
(defparameter *c-libraries*
 '("mpw:clib;stdclib.o""mpw:lib;interface.o"))

(ff-load "c-hacks;utils.c.o"
         :ffenv-name  'c-utils
         :libraries *c-libraries*
         :library-entry-points
         '("atoi" "strcmp"))

(deffcfun (frob "frob") ...)   ;frob is defined in utils.c
(deffcfun (atoi "atoi") ...)
(deffcfun (strcmp "strcmp") ...)
```

For information on setting up logical pathnames, look in the file `ff-example.lisp` in the FF Examples subfolder of your Examples folder.

dispose-ffenv

[*Function*]

Syntax `dispose-ffenv` *ffenv*

Description The `dispose-ffenv` function disposes of the heap storage used by the environment *ffenv* . If you call a foreign function residing in an environment that has been disposed of, you will almost certainly crash.

Argument *ffenv* A foreign function environment, as returned by `ff-load` , or a symbol naming a foreign function environment.

Example

See the section “A Short Example” later in this chapter.

defffun

[*Macro*]

deffcfun

[*Macro*]

deffpfun

[*Macro*]

Syntax
`defffun` (*lisp-name entry-name* {*option* }*) ({*argspec* }*) {*result-flag* }*
`deffcfun` (*lisp-name entry-name* {*option* }*) ({*argspec* }*) {*result-flag* }*
`deffpfun` (*lisp-name entry-name* {*option* }*) ({*argspec* }*) {*result-flag* }*

Description These macros help you define an MCL interface to a foreign function. You describe the arguments the function takes and the result it returns, and `defffun` defines an MCL function that performs appropriate coercions and type checks on the arguments and calls the foreign function. The `deffcfun` and `deffpfun` macros are identical to `defffun` except that they set the `:language` option.

Arguments

lisp-name The name of the MCL function that is defined by the macro. This must be a symbol.

entry-name The name of an active entry point in a foreign function environment. It should be a string. Entry point names are case sensitive. If *entry-name* exists in more than one loaded environment, the environment used is undefined.

option The entry name may be followed by options that further describe the foreign function. The options to `deffun`, `deffcfun`, and `deffpfun` provide information on the syntax and calling sequence of the function being defined. These options are `:language`, `:check-args`, and `:reverse-args`.

`:language`
The value of `:language` indicates the language used to define the foreign function, which in turn regulates defaults for other options. This option is currently used in the macroexpansion of `deffcfun` and `deffpfun`. Future extensions will support other language types.

`:check-args`
If the value of `:check-args` is non-`nil` (the default), the function performs run-time checks to ensure that the actual argument types match the declared expectations. If its value is `nil`, this type checking is skipped. This option may be overridden by individual *argspecs*.

`:reverse-args`
If the value of `:reverse-args` is non-`nil`, the arguments are pushed on the stack in reverse order from that specified in the *argspec* list. This is the default if the language is C.

argspec The description of a single argument to the foreign function. Argument specifications have the general form (*lisp-type* {*flag* }*). They are described in detail in the next section, "Argument Specifications."

result-flag The value returned by the function. Result flags are described in the section "Result Flags" later in this chapter.

Example

See the section "Short Example" later in this chapter.

Argument specifications

The argument specifications of foreign function calls give information about each argument the foreign function will receive, including the MCL type to expect and a series of flags. The flags give information on the foreign type, the argument-passing method, and the necessity for argument checking.

lisp-type Any valid MCL type specifier. It declares that the corresponding argument to the MCL function will be of that type. If argument checking is requested, a `check-type` form is included in the MCL function. In addition, *lisp-type* is used to select the argument-passing convention and foreign type, if these are not explicitly specified. If *lisp-type* is a symbol (not a list) and there are no *flags*, the parentheses around *argspec* may be omitted.

flags Specify the format (foreign type) in which the corresponding argument should be passed, the method for passing, and the necessity for performing a type check of the argument. The following flags describe the format (foreign type) of the argument to be passed to the foreign function. They are mutually exclusive; that is, you may choose only one of the possible values.

- `:long` The foreign function is expecting a longword value. The MCL argument should be an integer or a character (in which case its `char-code` is used). It can also be a four-character string or a symbol.
- `:word` The foreign function is expecting a word value. The MCL argument should be a fixnum. The low 16 bits of the value constitute the foreign argument.
- `:double` The foreign function is expecting a floating-point value in the machine double format (8 bytes). The MCL argument should be of type `double-float`.
- `:extended` The foreign function will receive a floating-point value in SANE extended format (10 bytes). The MCL argument should be of type `double-float`. The `:extended` flag is the default floating-point type for C and Pascal.
- `:ptr` The foreign function will receive a longword value. The argument should be an object of type `macptr`. (See Chapter 16, “Low-Level Operating System Interface,” for a description of this MCL data type.)

`:Lisp-ref`

The foreign function is expecting a pointer to an MCL value. A temporary location is reserved (for the duration of the call) in nonrelocatable memory, a pointer to the MCL argument is placed in that location, and the address of the location is passed to the foreign function. The foreign function can access the MCL data using double indirection, just as it would a Macintosh handle. The contents of the location are updated whenever the MCL data is relocated.

`:cstring`

The foreign function is expecting a null-terminated string. The MCL argument should be an MCL string. The foreign function must not modify any locations beyond the end of the string. The string may be of any length.

`(:cstring` *size* `)`

The foreign function is expecting a null-terminated string in a buffer of *size* bytes (not including the null). The MCL argument should be a string of *size* characters or less. The foreign function must not modify any locations beyond the end of the buffer.

`:pstring`

The foreign function is expecting a Pascal string (a string preceded by a count byte). The MCL argument should be a string. The foreign function must not modify any locations beyond the end of the string. If the argument is longer than 255 characters, an error is signaled.

`(:pstring` *size* `)`

The foreign function is expecting a string preceded by a length byte in a buffer of *size* bytes (not including the length byte). The MCL argument should be a string *size* characters long or less. The foreign function must not modify any locations beyond the end of the buffer.

If no foreign format flag is specified, a default is chosen based on the *lisp-type* and the `:language` option, according to Table 15-1. If *lisp-type* is not listed in the table, there is no default (and the foreign type must be specified).

Table 15-1 Foreign type defaults

MCL type	In C	In Pascal
Integer	`:long`	`:word`
Character	`:long`	`:word`
String	`:cstring`	`:pstring`
Float	`:extended`	`:extended`

The following *flags* are used to specify the argument-passing method. They are mutually exclusive; that is, you may choose only one of the possible values.

`:by-value`
: The value of the argument is passed to the foreign function (pushed on the stack). This method may not be used to pass strings (that is, the argument format must not be `:cstring` or `:pstring`).

`:by-address`
: The address of a location containing a copy of the argument is passed to the foreign function (pushed on the stack). If the foreign function modifies the argument, the changes will not be visible to Macintosh Common Lisp. Use `:by-reference` if you want Macintosh Common Lisp to see changes.

`:by-reference`
: The address of a location containing a copy of the argument is passed to the foreign function (pushed on the stack). In addition, arrangements are made so that upon return from the foreign function, any changes in the copy are reflected in the MCL argument itself. Thus foreign functions may destructively modify MCL data structures. Only floating point values and strings may be passed by reference (that is, the argument format must be `:cstring`, `:pstring`, `:double`, or `:extended`). When a string is passed by reference and the foreign function changes the size of the string, an error is signaled unless the MCL string is adjustable.

When no passing method is specified, the default is to pass `:long`, `:word`, `:ptr`, and `:Lisp-ref` arguments by value and others by address. In addition, if the language is C, `:extended` arguments are passed by value.

The following flags specify whether type checking should be performed on *lisp-type*. Using one of these flags lets you override the `:check-args` option for the function as a whole. The two flags are mutually exclusive.

`:check-arg`
: The argument is checked at run time to ensure it is of type *lisp-type*.

`:no-check-arg`
: The argument type is not checked at run time.

Result flags

The value returned by the foreign function is described by *result-flags*. These flags describe the type and location of the returned value.

The type of the returned value is described by one of the following keywords. The type determines how the result is coerced before it is passed back to Macintosh Common Lisp.

`:long` The result is interpreted as a 32-bit signed integer and returned as a signed MCL integer.

`:full-long`
: The result is equivalent to `:long`; maintained for backward compatibility.

`:word` The returned value is a 16-bit word. It is interpreted as a signed integer and returned as a fixnum.

`:double` The foreign result is a double float. It is coerced to an MCL double float.

`:extended` The foreign result is a float in extended format. It is coerced to an MCL double float.

`:float` This is a synonym for `:extended`.

`:char` The foreign result is a character. The low 8 bits of the returned value are interpreted as a character code and returned as an MCL character.

`:ptr` The returned value is a 32-bit integer of type `macptr`.

`:novalue` The foreign function returns no value. The MCL function returns `nil`. This is the default if the language is Pascal.

The location of the returned value is described by one of the following keywords:

`:d0`–`:d7`	The foreign function returns the value in the specified data register as a signed MCL integer. The default is `:d0` if the language is `:c`.
`:a0`–`:a4`	The value in the specified address register is returned as a macptr. (See Chapter 16, "Low-Level Operating System Interface," for a description of macptrs.).
`:stack`	The foreign function returns the value on top of the stack as a macptr.

Short example

Assume that the file `test.c` contains the following C function:

```
#include <ctype.h>
#include <memory.h>

digitval (ch)
  char ch;
{  if isdigit(ch) return ch - '0';
  else return -1;
            }
```

After compiling this file in MPW, you can use it from within Macintosh Common Lisp as follows:

```
? (ff-load "test.c.o" :ffenv-name 'test
                      :libraries *c-libraries*)
#<A FF-ENV>
? (deffcfun (digit-value "digitval")
            (character) :long)
DIGIT-VALUE
? (digit-value #\7)
7
? (digit-value #\A)
-1
```

Low-level functions

The following low-level functions are used to implement the higher-level functions described earlier in this chapter. You may want to use the low-level functions if you require increased speed or flexibility. The `ff-call` function is faster than higher-level functions because it is open-coded and the arguments are not coerced or checked for types. This gives more flexibility at the cost of dependability.

If you use these functions, it is your responsibility to pass the right types; if you pass the wrong types, the system will probably crash.

`ff-call` [*Function*]

Syntax `ff-call` *pointer* `{` *type-keyword argument* `}*` `[:return-block` *pointer* | *return-value-keyword* `]`

Description The `ff-call` function transfers control to the address *pointer*, passing arguments according to the *type-keyword* /*argument* pairs.

The `ff-call` function returns a value according to `:return-block` or *return-value-keyword*. If the `:return-block` keyword is present, `ff-call` returns `nil`. If it is not present, `ff-call` returns the value appropriate for *return-value-keyword*.

This function is useful if you have `'CODE'` resources with entry points at known offsets. Keep in mind, however, that no type checking is performed on the arguments and that the system will probably crash if bad arguments are passed.

The stack-based arguments are pushed on the stack in the order in which they appear in the call.

If the returned value is to be taken from the stack, room is left on the stack for this value before any stack-based arguments are pushed. The foreign function is entered at *pointer* with the return address on the top of the stack. The function does not have to obey strict stack discipline, but it must never lower the stack beyond its arguments (in other words, it should never pop anything more than its arguments, and it should not modify any data on the stack beyond the arguments).

Arguments

pointer A pointer to the entry point of the foreign function to be called.

type-keyword Indicates what type of coercion to perform on the subsequent argument and where to place the argument. The possible values of *type-keyword* are the following:

`:word` The MCL argument should be a fixnum. The low 16 bits of the argument are pushed on the stack.

`:long` The MCL argument should be an integer or a character, in which case its `char-code` is used. It can also be a four-character string or a symbol. It is coerced to a longword and the result is pushed on the stack. (See the section "32-Bit Immediate Quantities" in Chapter 17, "Higher-Level Operating System Interface.")

`:ptr` An object of type `macptr`. Its associated address is pushed on the stack. (See Chapter 16, "Low-Level Operating System Interface," for a description of macptrs.)

`:d0-:d7` The MCL argument should be an integer or character, in which case its `char-code` is used. It can also be a four-character string or a symbol. It is coerced to a longword and the result is placed in the indicated data register. The same constraints apply as in `:long`, described previously.

`:a0-:a4` A macptr. Its associated address is placed in the indicated address register.

`:a5` A macptr. Its associated address is placed in the A5 register and in the Macintosh low memory global `CurrentA5` for the duration of the call.

argument An argument.

`:return-block`

A keyword. If this keyword appears in a call to `ff-call`, it should be followed by a pointer to a block of memory where the returned values are placed. This mechanism lets a foreign function return values from multiple registers and positions on the stack. If this keyword is present, `ff-call` returns `nil`.

return-value-keyword

Indicates the type of value the function returns if `:return-block` is not present. If the value of *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

`:word`	A 16-bit result is read from the top of the stack, sign extended, and returned as a signed MCL integer.
`:long`	A 32-bit result is read from the top of the stack and returned as a signed MCL integer.
`:ptr`	A 32-bit result is read from the top of the stack and returned as an object of type `macptr`.
`:d0-:d7`	The value in the indicated data register is returned as a signed MCL integer.
`:a0-:a4`	The value in the indicated data register is returned as a macptr. (See Chapter 16, "Low-Level Operating System Interface," for a description of macptrs.)
`:novalue`	No value is returned from the function. The `ff-call` function returns `nil`.

ff-lookup-entry

[*Function*]

Syntax `ff-lookup-entry` *entry-name*

Description The `ff-lookup-entry` function returns two values describing the entry point named *entry-name* . The entry point must be an active entry point in some previously loaded environment. Note that entry point names are case sensitive. The first value returned is a pointer to the entry point. The second value is the A5 pointer for the environment where the entry point was found. If *entry-name* does not exist, `nil` is returned. If *entry-name* exists in more than one loaded environment, the specific environment returned is undefined.

Using the `ff-lookup-entry` function is a relatively slow operation. You normally call it once at load time and store the results in some easily accessible place, rather than calling it every time you need to reference the entry point. Note that the values returned by `ff-lookup-entry` are pointers and thus cannot be maintained across dumps and restarts. When restarting a dumped image, you must reinitialize all entry information by calling `ff-lookup-entry` again.

Argument *entry-name* A string giving the name of an entry point in a foreign function environment. The string is case sensitive.

Examples

Here are two examples.

```
;frob is a foreign function:
? (multiple-value-bind (entry a5)
                       (ff-lookup-entry "frob")
                       (ff-call entry :a5 a5))
;FrobCount is a static integer variable:
? (format T "There are ~D frobs."
          (%get-long
           (ff-lookup-entry "FrobCount")))
```

%word-to-int

[*Function*]

Syntax `%word-to-int` *fixnum*

Description The `%word-to-int` function sign-extends the low 16 bits of *fixnum* to a full integer value.

Argument *fixnum* A fixnum.

Example

```
? (= (%word-to-int #xFFFF) -1)
T
? (= (%word-to-int 1) 1)
T
```

%copy-float

[*Function*]

Syntax `%copy-float` *float*

Description The `%copy-float` function returns a copy of *float* , that is, a newly consed floating-point number that is `eql` but not `eq` to *float* . The argument can be an MCL `double-float` or a macptr to a memory location that contains a 64-bit floating-point number in machine format (see 68881 or SANE documentation for details).

Argument *float* A floating-point number.

Calling Macintosh Common Lisp from foreign functions

A foreign function may call an MCL function, receive a returned value, and do further processing before returning to Macintosh Common Lisp. An MCL function that is to be called by a foreign function must be defined with one of the following macros. The macros create a pointer, which can then be passed to the foreign function.

The `defccallable` macro is used to define MCL functions that can be called from the C Toolbox. The `defpascal` macro is used to define MCL functions that can be called from the Macintosh Toolbox or from user-written Pascal code. Both of these macros put an MCL function in the function cell of a symbol and a pointer to the C or Pascal entry point in the value cell of a symbol.

The `defpascal` and `defccallable` macros are described in Chapter 16, "Low-Level Operating System Interface."

The following example uses `defccallable`. It uses a C function, `addthree`, that takes two arguments: an integer and a pointer to an MCL function. The C function adds 1 to its first argument, then calls the MCL function pointed to by its second argument. This MCL function is passed the incremented first argument.

The MCL function increments its argument and returns it, whereupon the C function increments it again and returns the value. Here control passes from Macintosh Common Lisp to C to Macintosh Common Lisp to C and finally back to Macintosh Common Lisp.

The `addthree` function is accurately named only if it is passed a pointer to an MCL function that takes a `fixnum` argument, increments it, and returns the incremented value.

Here is the C function:

```
int addthree (i, Lispfn)
                int i, (*Lispfn) ();
                {
                i = i + 1;
                i = (*Lispfn) (i);
                return i + 1;
                }
```

The MCL `defccallable` macro sets the value of its first argument to the entry point for the function that it defines. Here is the MCL macro:

```
? (defccallable add-one
            (:long i :long))
            (+ i 1))
ADD-ONE
```

The pointer to the MCL function is not an MCL data type, so use the type specifier.

```
? (deffcfun (add-three "addthree")
            ((fixnum :long) (t :ptr))
            :long)
ADD-THREE
```

The pointer to the MCL function is stored in the value cell of `add-one`, so you don't need to quote the symbol or use the special form `function`.

```
? (add-three 5 add-one)
8
```

Extended example

The files `ff-example.c`, `ff-example.Lisp`, and `ff-example.test` in the FF Examples folder in your Examples folder contain an expression-by-expression example of how to use the Foreign Function Interface with C.

You may find it convenient to switch between MPW and Macintosh Common Lisp while doing this. (Under Macintosh Operating System version 6, you must use MultiFinder. MultiFinder functionality is built into system software version 7.)

Perform the following steps.

- Boot MPW.
- Edit your foreign language code. The examples use a set of C functions defined in the file `example.c` on the Foreign Function Interface disk.
- Compile your code. Use the MPW Build facility to make sure you get all of the right library files.
- Test your code in MPW. This stage isn't strictly necessary but will ensure that you pass MCL-proven working code. If you don't test your code in MPW, it isn't necessary to link it. Macintosh Common Lisp needs only the object files.
- Start Macintosh Common Lisp and load the Foreign Function Interface.

  MCL code for loading the foreign function files and defining an MCL interface to the functions is given in the file `example.lisp` in the FF Examples subfolder of your MCL Examples folder.
- Call the foreign functions from Macintosh Common Lisp.

Examples for testing the code are contained in the file `example.test` in the FF Examples subfolder.

Chapter 16 Low-Level Operating System Interface

Contents

Interfacing to Macintosh data structures / 537
Upgrading from previous versions of Macintosh Common Lisp / 538
Sharing data between Macintosh Common Lisp and the Macintosh Operating System / 539
Macptrs / 539
Memory management / 541
Stack blocks / 542
Accessing memory / 543
Miscellaneous routines / 554
Strings, pointers, and handles / 554
Pascal VAR arguments / 559
The Pascal null pointer / 559
Callbacks to Lisp from the ROM / 559

This chapter discusses basic information necessary for accessing the Macintosh Toolbox and Macintosh Operating System, and describes the lowest level of access that is supported.

You should read this chapter if you are accessing Macintosh data structures at a low level. If you are using higher-level accessing, such as through records and traps, you may wish to read this chapter for background information.

Access by means of records and traps is discussed in Chapter 17, “Higher-Level Operating System Interface.” The highest level of access, through standard predefined MCL functions, is discussed throughout this manual. An example of converting trap calls to predefined MCL functions is documented in Appendix E, “QuickDraw Graphics.”

When making any operating system call, you should be familiar with its description in *Inside Macintosh* .

The discussions in this chapter assume some familiarity with *Inside Macintosh* .

Interfacing to Macintosh data structures

Macintosh Common Lisp provides several higher levels of interface to the Macintosh Operating System. Macintosh Common Lisp can access Macintosh data structures in any of several ways.

- At the highest level, you can gain access to the Macintosh Toolbox and Macintosh Operating System through standard predefined MCL classes, such as `view` and `window`, and the methods that apply to them. These are easy and safe to use, insulating you from both syntactic and semantic errors associated with Macintosh data structures.
- Through Macintosh Common Lisp, you also have access to the majority of the traps and to all of the records and constants defined in the MPW Pascal interfaces. Discussed in Chapter 17, "Higher-Level Operating System Interface," these trap interface files are included for the first time with Macintosh Common Lisp version 2.

  These trap interfaces are simple to use, and since Macintosh Common Lisp now performs error checking at compile time on their arguments, they are much more robust than the traps mechanism previously provided. By using these interface files, you can insulate yourself from a large class of syntactic errors associated with Macintosh data structures.
- You can define your own traps, records, and record types in Macintosh Common Lisp, which you can call just as you would the standard interface files. Macintosh Common Lisp performs compile-time error checking on their arguments as well.
- At a lower level, you can use the Lisp macros `stack-trap`, `register-trap`, and `%gen-trap`, discussed in Chapter 17, "Higher-Level Operating System Interface." These macros are less robust, since the two most efficient of them do no error checking.

  These macros should be used with care, since improper calls to the Macintosh Operating System can cause a system crash or destroy the integrity of your file system. (This is true of any call to the operating system, not only within Macintosh Common Lisp. In any environment, you should call the operating system cautiously.)

Upgrading from previous versions of Macintosh Common Lisp

Please note the following changes if you are upgrading code from a previous version of Macintosh Common Lisp:

- In previous versions of Macintosh Common Lisp, Lisp objects were constrained in their range of addresses, and it was possible to determine from a thing's address whether it was a Lisp object and even what its data type was. Because Macintosh Common Lisp now addresses a full 32 bits, Lisp objects may be stored in many more locations, and it is not feasible to tell the data type of an object from its address.
- Because it is meaningful to distinguish between a 32-bit value that is used as an address and one that is used for another purpose, Macintosh Common Lisp now includes a new Lisp data type, `macptr`. Any MCL object with a 32-bit value that represents a pointer to a Macintosh data structure must now be of type `macptr`. To distinguish them from Macintosh pointers that are not MCL objects, these objects are called macptrs. (See the section "Macptrs" later in this chapter.)
- All pointers to Macintosh data structures are macptrs.
- You cannot pass any other Lisp object, including `nil`, to a function requiring a macptr, and macptrs may never point to Lisp objects—only to the address of a Macintosh data structure.
- The Pascal null pointer is now `(%null-ptr)`, a macptr equivalent to `(%int-to-ptr 0)`. It is no longer equivalent to Macintosh Common Lisp `nil`.
- The function `eql`, not `eq`, compares two macptrs for equality.
- The functions `%put-ptr`, `%put-long`, and other functions that read and write to memory locations were previously defined to return the pointer being used. In Macintosh Common Lisp version 2 they return `nil`.
- MCL data types that may be translated to Macintosh 32-bit longwords now include four-character strings and symbols with four-character print names.

Sharing data between Macintosh Common Lisp and the Macintosh Operating System

Macintosh Common Lisp manipulates two distinct sets of data: Macintosh data, such as windows, patterns, and regions, and Lisp data, such as lists, symbols, and objects. Lisp data and Macintosh data are stored in different places and in different formats. Macintosh data is stored on the application heap, and Lisp data is stored on the Lisp heap. These two heaps operate independently. Each piece of data belongs on either one heap or the other.

Some Lisp data contains pointers to Macintosh data. For example, window objects (Lisp data) contain Macintosh pointers to Macintosh window records (on the Macintosh heap).

In general, Macintosh data is needed only for communication with Macintosh Operating System calls. Before Macintosh Common Lisp 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.

Macptrs

A macptr (an object of data type `macptr`) represents a 32-bit address.

Macptrs are generated in the following ways:

- from a call to a trap, such as `#_newHandle` or `#_newPtr`, that allocates a new pointer or handle
- from a call to `%get-ptr`, where you are referencing some memory location relative to some Macintosh pointer
- as the result of a call to `make-record`
- as the result of a call to `%int-to-ptr`
- as the binding of the temporary variables in a `%stack-block` or `rlet`

They are required in the following circumstances:

- as the first argument to `%get-` and `%put-` functions
- as the value of any parameter to a trap passed in an address register
- as the value of any parameter to a trap that requires a pointer, record, handle, or array

You cannot pass any other Lisp object to a function requiring a macptr. In particular, `nil` cannot be passed as a pointer to a Macintosh data structure; you cannot call the ROM passing Lisp `nil`. Instead, you pass the macptr `(%null-ptr)`.

Two macptrs to the same address are `eql` but not necessarily `eq`. That is, the two pointers themselves may not be the same; the address they reference is the same. If both `x` and `y` point to `(%int-to-ptr 0)`, then

```
? (eq x y)
```
...undetermined...
```
? (eql x y)
T
```

The address to which a macptr points is not changed by garbage collection.

In general, performing an operation such as `%int-to-ptr` or `%get-ptr` results in the allocation of a new Lisp macptr object. However, the MCL compiler avoids allocating macptrs whenever possible.

Here is an example in which a macptr is not allocated.

```
? (defun peek-long (addr)
      "Returns the contents of the longword at ADDR."
      (%get-long (%int-to-ptr addr)))
PEEK-LONG
```

Since the result of `(%int-to-ptr addr)` is used directly by `%get-long`, the compiler does not need to allocate a macptr.

By taking advantage of the following in your code, you can ensure that macptrs created incidentally are not allocated:

- Addresses that are consumed directly by primitive operations are not encapsulated in macptrs. Of the MCL low-level functions for reading and writing to memory locations, most avoid allocating macptrs. (See the sections “Accessing Memory” and “Strings, Pointers, and Handles” later in this chapter.)
- Macptrs can be explicitly stack-allocated. When appropriate, you may use `dynamic-extent` declarations to indicate that the compiler may safely stack-allocate macptrs used in local contexts.
- Macptrs can be destructively modified with `%setf-macptr` (see the following section).

%setf-macptr [*Function*]

Syntax `%setf-macptr` *macptr pointer*

Description The `%setf-macptr` function destructively modifies *macptr* so that it references the address referenced by *pointer*. The compiler open-codes this function.

Arguments

macptr	A macptr.
pointer	A macptr.

Memory management

Macintosh Common Lisp works in cooperation with the Macintosh Memory Manager. Thus you can use the traps `#_NewPtr` and `#_NewHandle` to allocate blocks of memory on the application heap. The Macintosh and Lisp heaps are dynamically resized to satisfy memory requests. This resizing sometimes triggers a garbage collection.

Important You are responsible for releasing memory allocated on the Macintosh heap. (You can do so with `#_DisposPtr` and `#_DisposHandle`.) The contents of this memory are not subject to garbage collection even if all pointers to the memory are lost.

The `#_NewPtr` and `#_NewHandle` traps are automatically called by `make-record`, described in Chapter 17, “Higher-Level Operating System Interface.”
The `#_DisposPtr` and `#_DisposHandle` traps are automatically called by `dispose-record`.

Stack blocks

When you need a small amount of memory for *temporary* storage, it is often more convenient and more efficient to bypass the Macintosh Memory Manager. The `%stack-block` macro allows programs to allocate blocks of memory on the stack. Be very careful using `%stack-block`. When you exit the `%stack-block` form, all the memory allocated is reclaimed and so any remaining pointers to the stack block become invalid. The `%stack-block` form should be used only for well-defined temporary storage, for example, to set up rectangles or I/O parameter blocks to be passed to traps, or to store *var* arguments temporarily.

`%stack-block` [*Macro*]

`%vstack-block` [*Macro*]

Syntax

`%stack-block ({(`*symbol size*`)}`$^{+}$`) {`*form*`}`*
`%vstack-block (`*symbol size*`) {`*form*`}`*

Description

For each *symbol/size* pair, the `%stack-block` macro allocates a block of storage *size* bytes long and binds *symbol* to a macptr to the block.

The action of these forms is semantically equivalent to doing a `#_NewPtr`/`#_DisposPtr` pair for each variable but is much more efficient. The bindings and the storage created by `%stack-block` have dynamic extent; they become invalid when the form is exited from.

If a storage block of indefinite extent is needed, `make-record` should be used instead; see Chapter 17, "Higher-Level Operating System Interface."

If there is not enough room on the stack to allocate the requested memory, an error is signaled. The forms are executed in the resulting environment.

The `%stack-block` macro is usually not used directly to stack allocate Macintosh records. Instead, use the `rlet` macro described in Chapter 17, "Higher-Level Operating System Interface."

(The `%vstack-block` macro is now semantically equivalent to `%stack-block` and is provided for backward compatibility only.)

Arguments	*symbol*	Any symbol. This symbol is bound to the stack block for the duration of the call.
	size	The maximum total size for an individual stack block form is 32K bytes. Every *size* must evaluate to a positive fixnum.
	form	Zero or more forms, which are evaluated as the body. Declarations may appear at the head of the body.

Example

In this example, an 8-byte block is allocated on the stack. The memory is filled with the coordinates of a rectangle. A pointer to the block—and two additional words are then passed to the trap `#_FrameRoundRect`. When the window is redrawn, a rectangle with rounded corners appears at the given coordinates:

```
? (setq my-window (make-instance 'window))
#<WINDOW "Untitled" #x46E929>
? (defmethod view-draw-contents ((w (eql my-window)))
    (let ((oval-width 12)
          (oval-height 8))
      (%stack-block ((my-rect 8))
        (%put-word my-rect 12 0)
        (%put-word my-rect 40 2)
        (%put-word my-rect 32 4)
        (%put-word my-rect 80 6)
        (#_FrameRoundRect my-rect
                          oval-width oval-height))))
#<STANDARD-METHOD VIEW-DRAW-CONTENTS ((EQL #<WINDOW "Untitled"
    #x46E929>))>
? (view-focus-and-draw-contents my-window)
NIL
```

Accessing memory

Once memory for a structure has been allocated, programs need methods for directly reading from and writing to the memory. Macintosh Common Lisp provides the following low-level functions for reading and writing to memory locations. (The macros `rref` and `rset` may be used. See Chapter 17, "Higher-Level Operating System Interface," for details.) Most calls to these functions are compiled inline for efficiency.

Each of the following functions takes *offset*, a fixnum, as an optional argument. No type-checking is performed on *offset*. It is sign-extended to 32 bits.

- *Note:* No variable error checking is performed on any of the following. Since their purpose is to let you read and write to the memory in unforeseen ways, they aren't designed to prevent serious programming errors, and it is possible to read and write to memory locations in ways that seriously affect your computer. For example, on a 68000 (but not a 68020) microprocessor, accessing a word at an odd memory address results in a fatal error. Writing to a nonexistent memory address results in a bus access error, and so on.

%get-signed-byte [*Function*]

Syntax `%get-signed-byte` *macptr* `&optional` *offset*

Description The `%get-signed-byte` function gets the byte (8 bits) at *macptr* + *offset* and returns it as a signed Lisp integer in the range –128 through 127. The compiler open-codes this function.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

%get-unsigned-byte [*Function*]

%get-byte [*Function*]

Syntax
`%get-unsigned-byte` *macptr* `&optional` *offset*
`%get-byte` *macptr* `&optional` *offset*

Description These equivalent functions get the byte (8 bits) at *macptr* + *offset* and return it as an unsigned Lisp integer in the range 0 through 255. The compiler open-codes these functions.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

%hget-byte

[*Function*]

Syntax `%hget-byte` *handle* `&optional` *offset*

Description The `%hget-byte` function accesses a handle, gets the byte (8 bits) at *handle* + *offset* , and returns it as an unsigned Lisp integer in the range 0 through 255. The compiler open-codes this function.

The expression `(%hget-byte` *handle offset* `)` is equivalent to `(%get-byte (%get-ptr` *handle* `)` *offset* `)`.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

%hget-signed-byte

[*Function*]

Syntax `%hget-signed-byte` *handle* `&optional` *offset*

Description The `%hget-byte` function accesses a handle, gets the byte (8 bits) at *handle* + *offset*, and returns it as a signed Lisp integer in the range 128 through 127. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

%get-signed-word

[*Function*]

Syntax `%get-signed-word` *macptr* `&optional` *offset*

Description The `%get-signed-word` function gets the word (16 bits) at *macptr* + *offset* , sign-extends it, and returns it as a signed Lisp integer in the range – 32,768 through 32,767. The compiler open-codes this function.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr* .

%get-unsigned-word [*Function*]

%get-word [*Function*]

Syntax `%get-unsigned-word` *macptr* `&optional` *offset*
`%get-word` *macptr* `&optional` *offset*

Description These equivalent functions get the word (16 bits) at *macptr* + *offset* and return it as an unsigned Lisp integer in the range 0 through 65,535. The compiler open-codes these functions.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

%hget-word [*Function*]

Syntax `%hget-word` *handle* `&optional` *offset*

Description The `%hget-word` function accesses a handle, gets the word (16 bits) at *handle* + *offset*, and returns it as an unsigned Lisp integer in the range 0 through 65,535. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

%hget-signed-word [*Function*]

Syntax `%hget-signed-word` *handle* `&optional` *offset*

Description The `%hget-signed-word` function accesses a handle, gets the word (16 bits) at *handle* + *offset*, sign-extends it, and returns it as a signed Lisp integer in the range –32,768 through 32,767. The compiler open codes this function.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

%get-long [*Function*]

%get-signed-long [*Function*]

Syntax `%get-long` *macptr* `&optional` *offset*
`%get-signed-long` *macptr* `&optional` *offset*

Description These equivalent functions get the macptr at *macptr* + *offset* and return a signed Lisp integer in the range – 2,147,483,648 through 2,147,483,647. The compiler open-codes these functions.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

%hget-long [*Function*]

%hget-signed-long [*Function*]

Syntax `%hget-long` *handle* `&optional` *offset*
`%hget-signed-long` *handle* `&optional` *offset*

Description These functions access a handle, get the macptr at *handle* + *offset*, and return a signed Lisp integer in the range –2,147,483,648 through 2,147,483,647. The compiler open-codes these functions.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

%get-unsigned-long [*Function*]

Syntax `%get-unsigned-long` *macptr* `&optional` *offset*

Description The `%get-unsigned-long` function gets the macptr at *macptr* + *offset* and returns an unsigned Lisp integer in the range 0 through 4,294,967,295. The compiler open-codes this function.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

%hget-unsigned-long [*Function*]

Syntax `%hget-unsigned-long` *macptr* `&optional` *offset*

Description The `%hget-unsigned-long` function gets the macptr at *macptr* + *offset* and returns an unsigned Lisp integer in the range 0 through 4,294,967,295. The compiler open-codes this function.

Arguments

macptr	A handle.
offset	Window.

%get-ptr [*Function*]

Syntax `%get-ptr` *macptr* `&optional` *offset*

Description The `%get-ptr` function returns the macptr at *macptr* +*offset* . The compiler open-codes this function. The resulting macptr is heap consed unless it is used by another open-coded low-level primitive or used as the initial binding of a variable that is declared to have dynamic extent.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr* .

%hget-ptr [*Function*]

Syntax `%hget-ptr` *handle* `&optional` *offset*

Description The `%hget-ptr` function accesses a handle and returns the macptr at *handle* +*offset* . The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
offset	Window.

%get-string [*Function*]

Syntax `%get-string` *macptr* `&optional` *offset*

Description The `%get-string` function gets the Pascal string at *macptr* + *offset* and returns it as a Lisp string. This function is not open-coded by the compiler. If *macptr* points to a handle on the Macintosh heap, the handle is dereferenced to access the string.

(Pascal strings are formatted differently from Lisp strings. The Macintosh Toolbox uses Pascal strings.)

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

%get-cstring [*Function*]

Syntax `%get-cstring` *macptr* `&optional` *offset end*

Description The `%get-cstring` function gets a *string* at *macptr* + *offset* and returns it as a Lisp string. This function is not open-coded by the compiler. If *macptr* points to a handle on the Macintosh heap, the handle is dereferenced to access the string.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*. The default is 0.
end	The end of the string to be gotten. The default is *offset*.

%get-ostype [*Function*]

Syntax `%get-ostype` *macptr* `&optional` *offset*

Description The `%get-ostype` function gets the 4 bytes at *macptr* + *offset* and returns them as an os-type, a keyword of four characters. (See *Inside Macintosh* for details on os-types.) It returns `nil`.

Arguments

macptr	A macptr.
offset	A fixnum used to offset the address specified by *macptr*.

%put-byte

[*Function*]

Syntax `%put-byte` *macptr data* `&optional` *offset*

Description The `%put-byte` function stores the low 8 bits of *data* at *macptr* + *offset* . It returns `nil`. The compiler open-codes this function.

Arguments

macptr	A macptr.
data	A fixnum. The low 8 bits are used. This argument is not type-checked.
offset	A fixnum used to offset the address specified by *macptr* .

%hput-byte

[*Function*]

Syntax `%hput-byte` *handle data* `&optional` *offset*

Description The `%hput-byte` function accesses a handle and stores the low 8 bits of *data* at *handle* + *offset* . It returns `nil`. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
data	A fixnum. The low 8 bits are used.
offset	A fixnum used to offset the address specified by *macptr* .

%put-word

[*Function*]

Syntax `%put-word` *macptr data* `&optional` *offset*

Description The `%put-word` function stores the low 16 bits of *data* at *macptr* + *offset* . It returns `nil`. The compiler open-codes this function.

Arguments

macptr	A macptr.
data	A fixnum. The low 16 bits are used. This argument is not type-checked.
offset	A fixnum used to offset the address specified by *macptr* .

%hput-word

[*Function*]

Syntax `%hput-word` *handle data* `&optional` *offset*

Description The `%hput-word` function accesses a handle and stores the low 16 bits of *data* at *handle* + *offset* . It returns `nil`. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
data	A fixnum. The low 16 bits are used.
offset	A fixnum used to offset the address specified by *handle* .

%put-long

[*Function*]

%put-full-long

[*Function*]

Syntax
`%put-long` *macptr data* `&optional` *offset*
`%put-full-long` *macptr data* `&optional` *offset*

Description These functions store the integer value of *data* at *macptr* + *offset* . (The function `%put-full-long` is included for backward compatibility; both functions allow full 32-bit accuracy.) The compiler open-codes these functions.

Arguments

macptr	A macptr.
data	Any Lisp object that can be coerced to a 32-bit immediate quantity.
offset	A fixnum used to offset the address specified by *macptr* .

%hput-long

[*Function*]

Syntax `%hput-long` *handle data* `&optional` *offset*

Description The `%hput-long` function accesses a handle and stores the integer value of *data* at *handle* + *offset* . The compiler open-codes this function.

Arguments

handle	A dereferenced handle. This argument must be a macptr.
data	Any Lisp object that can be coerced to a 32-bit immediate quantity.
offset	A fixnum used to offset the address specified by *handle* .

%put-ptr

[*Function*]

Syntax `%put-ptr` *macptr data* `&optional` *offset*

Description The `%put-ptr` function stores *data* as a macptr at *macptr* + *offset* and returns `nil`. The compiler open-codes this function.

Arguments

macptr	A macptr.
data	Another macptr.
offset	A fixnum used to offset the address specified by *macptr* .

%hput-ptr

[*Function*]

Syntax `%hput-ptr` *handle data* `&optional` *offset*

Description The `%hput-ptr` function accesses a handle, stores *data* as a macptr at *handle* + *offset* , and returns `nil`. The compiler open-codes this function.

Arguments

handle	A handle. This argument must be a macptr.
data	Another macptr.
offset	A fixnum used to offset the address specified by *handle* .

%put-string

[*Function*]

Syntax `%put-string` *macptr string* `&optional` *offset maxsize*

Description The `%put-string` function stores *string* as a Pascal string starting at *macptr* + *offset* . The compiler does not open-code this function.

Arguments

macptr	A macptr.
string	A string.
offset	A fixnum used to offset the address specified by *macptr*. The default is 0.
maxsize	The maximum size of the string. The default is 255.

%put-cstring

[*Function*]

Syntax `%put-cstring` *macptr string* `&optional` *offset maxsize*

Description The `%put-cstring` function stores *string* starting at *macptr* + *offset*. The compiler open-codes this function.

Arguments

macptr	A macptr.
string	A string.
offset	A fixnum used to offset the address specified by *macptr*.
maxsize	The maximum allowable size of the string. If the length of *string* is larger than *maxsize*, an error is signaled.

%put-ostype

[*Function*]

Syntax `%put-ostype` *macptr string* `&optional` *offset*

Description The `%put-ostype` function stores *string* as 4 bytes at *macptr* + *offset*. The argument *string* may be anything that can be coerced to a 32-bit value—in other words, `%put-ostype` is another name for `%put-long`. The compiler open-codes this function.

Arguments

macptr	A macptr.
string	A string with a length of four characters, used to specify the os-type, or a symbol with a four-character `symbol-name`.
offset	A fixnum used to offset the address specified by *macptr*.

Miscellaneous routines

The following routines provide additional tools and information for communicating between Macintosh Common Lisp and the Macintosh Operating System. These include a set of functions for performing various types of data conversion and handle dereferencing, information on Pascal VAR arguments and Boolean values, and the definitions of Lisp functions that can be called by ROM routines.

Strings, pointers, and handles

The following utilities are provided for using strings, pointers, and handles.

with-pstrs [*Macro*]
with-cstrs [*Macro*]
with-returned-pstrs [*Macro*]

Syntax

`with-pstrs` ({(*symbol string* `&optional` *start end*)}$^+$) {*form*}$^+$
`with-cstrs` ({(*symbol string* `&optional` *start end*)}$^+$) {*form*}$^+$
`with-returned-pstrs` ({(*symbol string* `&optional` *start end*)}$^+$)
{*form*}$^+$

Description

The `with-pstrs` macro allocates memory for each *string*, stores *string* in this memory in the Pascal string format, and binds the corresponding *symbol* to a pointer to the memory.

The `with-pstrs` macro saves the trouble of allocating memory and filling it with individual characters every time a Macintosh-accessible string is needed.

The `with-cstrs` macro allocates memory for each *string*, stores *string* in this memory, and binds the corresponding *symbol* to a pointer to the memory.

The `with-returned-pstrs` macro allocates 256 bytes for each string (rather than trying to optimize the amount of memory allocated). This guarantees that the returned string does not overwrite other segments of memory.

Traps that use the strings as VAR arguments for returning values should be called through the macro `with-returned-pstrs` .

Arguments

symbol	A symbol that is bound to the Pascal string for the duration of the macro.
string	A string with a maximum length of 255 characters.
start	The position at which to begin reading characters from the string.
end	The position at which to stop reading characters from the string.
form	Zero or more forms that make up the body of the macro.

`macptrp` [*Function*]

Syntax `macptrp` *thing*

Description The `macptrp` function returns `t` if and only if *thing* is a macptr; otherwise, it returns `nil`.

Argument

thing	Any Lisp data object.

`pointerp` [*Function*]

Syntax `pointerp` *macptr* `&optional` *errorp*

Description The `pointerp` function returns `t` if and only if the address that *macptr* contains is in RAM; otherwise, it returns `nil`. If *macptr* is not a macptr, an error is signaled.

Arguments

macptr	A macptr.
errorp	An argument specifying how to treat errors. If *errorp* is true, `pointerp` signals an error if *macptr* is not a macptr. Otherwise it returns `nil`.

zone-pointerp [*Function*]

Syntax `zone-pointerp` *thing*

Description The `zone-pointerp` function returns `t` if *thing* is a pointer to a nonrelocatable system or application-heap-zone memory block; otherwise, it returns `nil`.

The test is performed heuristically by determining if *thing* points to a Macintosh heap zone, and if so, determining if the longword before the address pointed at by *thing* is equal to the heap zone. A zone pointer is different from a generic pointer because the former points to a memory block that was allocated using `#_NewPtr`, and because the various Memory Manager pointer traps, such as `#_DisposPtr`, may be used on it.

For more information on the structure of zone pointers, see the information on the Memory Manager in *Inside Macintosh*.

Argument *thing* Any Lisp data object.

handlep [*Function*]

Syntax `handlep` *thing*

Description The `handlep` function returns `t` if *thing* is a Macintosh handle; otherwise, it returns `nil`.

This is determined heuristically by checking to see whether *thing* points to the system or application heap zone, and if so, indirecting through *thing* to determine if the longword prior to the address plus the zone pointer are equal to *thing*.

For more information on the structure of handles, see the information on the Memory Manager in *Inside Macintosh*.

Argument *thing* Any Lisp data object.

with-dereferenced-handles [*Macro*]

Syntax `with-dereferenced-handles ({(`*variable handle*`)}+) {`*form*`}+`

Description The `with-dereferenced-handles` macro executes *forms* with each *variable* bound to the locked, dereferenced *handle*.

Only previously unlocked handles are locked. Upon exit, only handles that were unlocked on entry are unlocked. (This prevents a bug that occurs when programming the Macintosh computer with other languages.) Unlocking of handles is protected against with `unwind-protect`, so the handles are guaranteed to be left in the same state before and after the call to `with-dereferenced-handles`, even if termination is abnormal.

Arguments

variable	A Lisp variable.
handle	A Macintosh handle.
form	A set of forms that are evaluated sequentially in the environment in which the handles are dereferenced and bound to variables.

with-pointers [*Macro*]

Syntax `with-pointers ({(`*variable pointer-or-handle*`)}+) {`*form*`}+`

Description The `with-pointers` macro binds *variable* to the pointer for each *pointer-or-handle* that is a pointer and binds *variable* to the locked, dereferenced handle for each *pointer-or-handle* that is a handle.

When binding handles, `with-pointers` acts just as `with-dereferenced-handles` does.

The `with-pointers` macro is useful if you are unsure whether *pointer-or-handle* will be a pointer or a handle. It signals an error if *pointer-or-handle* is neither a pointer nor a handle.

Arguments

variable	A Lisp variable.
pointer-or-handle	A Macintosh pointer or handle.
form	One or more forms that are evaluated in the resulting environment.

%inc-ptr [*Function*]

Syntax `%inc-ptr` *pointe* `&optional` *number*

Description The `%inc-ptr` function increments (or decrements) *pointer* by adding *number* to it and returns a new pointer. The compiler open-codes this function.

Arguments

pointer	A macptr.
number	An integer. The default is 1.

%incf-ptr [*Macro*]

Syntax `%incf-ptr` *pointer number*

Description The `%incf-ptr` function destructively modifies *pointer* by adding *number* to it and returns the modified *pointer* . The compiler open-codes this function.

Arguments

pointer	A mactptr.
number	An integer.

%ptr-to-int [*Function*]

Syntax `%ptr-to-int` *pointer*

Description The `%ptr-to-int` function returns an integer coerced from *pointer* , that is, the numerical address *pointer* points to. The compiler open-codes this function. This function may put a bignum onto the stack.

Argument

pointer	A macptr.

%int-to-ptr [*Function*]

Syntax `%int-to-ptr` *integer*

Description The `%int-to-ptr` function returns a macptr coerced from *integer* , that is, a pointer to the numerical address *integer* . The compiler open-codes this function.

Argument

integer	An integer.

Pascal VAR arguments

Pascal VAR arguments are passed by reference rather than by value; that is, you pass the function of a pointer to a piece of data rather than to the data itself. The called function may then affect the data and in this way communicate information to the caller. Implementing VAR arguments in Macintosh Common Lisp is very easy. Just allocate memory of the appropriate size for the piece of data (either on the stack or on the Macintosh heap, depending on how long you want to use that piece of data) and pass the macptr as the VAR argument.

The Pascal null pointer

The following two MCL expressions relate to the Pascal null pointer.

%null-ptr [*Macro*]

Syntax `%null-ptr`

Description The result of `(%null-ptr)` is equivalent to the Pascal null pointer and to `(%int-to-ptr 0)`. In Macintosh Common Lisp version 1.3, Lisp `nil` was the same as `(%int-to-ptr 0)`, but this is no longer true.

%null-ptr-p [*Function*]

Syntax `%null-ptr-p` *pointer*

Description The `%null-ptr-p` function returns `t` if *pointer* is a Pascal null pointer.

Argument *pointer* A pointer.

Callbacks to Lisp from the ROM

The following macros define Lisp functions that can be passed to the ROM or other C or Pascal code. This lets the other code make callbacks to Lisp.

defpascal [*Macro*]

defccallable [*Macro*]

Syntax

`defpascal` *name* (`{`*type parameter* `}`* [*return-type*]) [*doc-string*]
`{`*form*`}`*

`defpascal` *name* (`:reg` *parameter*) `{`*form*`}`*

`defccallable` *name* (`{`*type parameter* `}`* [*return-type*])
[*doc-string*] `{`*form*`}`*

Description

The syntax of these macros is similar to that of `defun`, except that the lambda list contains alternating types and parameter names and ends with a type specifier for the value returned by the procedure. The `&optional`, `&keyword`, and `&rest` keywords are not permitted, because they are not supported by the Pascal- or C-calling sequences.

The value cell of *name* is set to a pointer to the procedure. This pointer may be passed to traps or to C or Pascal code that expects a pointer to a function.

Arguments

name	A symbol to name the function.
type	The type of the corresponding parameter. Arguments passed to the function are taken from the stack and coerced according to this type-specifier keyword. The following values are legal:
`:word`	The argument is a 16-bit fixnum.
`:long`	The argument is a signed integer.
`:ptr`	The argument is a macptr.
parameter	The name of the parameter.
return-type	The type of the value returned by the function. The following values are legal:
`:word`	The returned value is a 16-bit fixnum.
`:long`	The returned value is interpreted as a 32-bit signed integer and returned as a signed MCL integer.
`:ptr`	The returned value is a macptr.
`:void`	The procedure does not return a value. (Omitting *return-type* is equivalent to a *return-type* of `:void`.)
doc-string	A documentation string.
form	The body of the function; zero or more forms that are evaluated as an implicit `progn` procedure.

Example

The following example is a simplified version of the `scroll-bar-proc` procedure from the file `scroll-bar-dialog-items.lisp` in the Library folder. The trap `#_TrackControl` is documented in *Inside Macintosh*.

```
? (defpascal my-track-proc (:ptr my-control
                            :word partCode
                            :void)
    (#_SetCtlValue my-control
     (+ (#_GetCtlValue my-control)
        (case partCode
          (20 -1)                       ;scroll back one line
          (21 1)                       ;scroll forward one line
          (22 (- *page-height*))        ;scroll back one page
          (23 *page-height*)))))        ;scroll forward one page
```

It could be used as follows:

```
(if
  (#_TrackControl the-control mouse-point my-track-proc)
    (view-draw-contents w))
```

In the second calling sequence for `defpascal`, only a single parameter is given. When the function is called, the values of the CPU registers are copied into a record. The values of the record can be set and accessed with `rref` and `rset` (for details, see Chapter 17, “Higher-Level Operating System Interface”). The value returned by the function is the pointer to the record.

An example of the use of `defccallable` is given in Chapter 15, “Foreign Function Interface.”

Chapter 17 Higher-Level Operating System Interface

Contents

Traps and records / 565
- Changes from previous versions of Macintosh Common Lisp / 565

Traps / 566

Loading trap definitions / 567

Defining a trap / 569
- Examples of calling traps / 570
- Determining record types and record field types / 571

Records / 574
- Installing record definitions / 575
- The structure of records / 575
- Defining record types / 576
- Variant fields / 579

Creating records / 580
- Creating temporary records with `rlet` / 580
- Creating records with indefinite extent / 582

Using records / 584

Getting information about records / 593

Trap calls using `stack-trap` and `register-trap` / 594
- Low-level stack trap calls / 595
- Low-level register trap calls / 597
- Lisp macros for calling Macintosh traps / 598

Notes on trap calls / 603
- 32-bit immediate quantities / 603
- Boolean values: Pascal true and false / 604

This chapter discusses how to make calls to the Macintosh Operating System that take place through traps and records, at a higher level than those discussed in Chapter 16, "Low-Level Operating System Interface."

You should read this chapter if you are using records or traps within Macintosh Common Lisp or if you are using Macintosh Common Lisp to program applications on the Macintosh. Traps discussed in this chapter are those documented in *Inside Macintosh* , so this chapter assumes some familiarity with *Inside Macintosh* .

If you need to make low-level calls to the operating system, you should also read Chapter 16, “Low-Level Operating System Interface.”

You must keep some caveats in mind when sharing data between Macintosh Common Lisp and the Macintosh Operating System. These are described in the first three sections of Chapter 16, “Low-Level Operating System Interface.” You should be familiar with the information in those sections before reading the rest of this chapter.

Traps and records

Traps call procedures in the Macintosh Operating System. (They are defined and discussed in *Inside Macintosh*.) Traps often require the allocation of **records,** areas of Macintosh memory usually stored as Pascal records. For example, the procedure to draw and fill an oval in a window requires calling a trap—calling a Macintosh Operating System procedure that knows how to draw and fill an oval. That trap in turn requires an area of memory, a record, to store the rectangle that defines the filled oval.

MCL functions and programs work efficiently with traps and records. You can easily access and alter data structures created at run time (such as windows and event records) as well as Macintosh resources.

The traps mechanism has been simplified and made more robust in this version of Macintosh Common Lisp. Macintosh Common Lisp now has interface files containing information about all Macintosh Operating System traps, constants, records, and mactypes. If the value of `*autoload-traps*` is true, information is automatically loaded from the relevant interface file when the trap is called. This functionality is described next.

Changes from previous versions of Macintosh Common Lisp

Previous versions of Macintosh Common Lisp included a source code file, `traps.lisp`, which defined the MCL traps interface. Previous versions of Macintosh Common Lisp had no information about traps, constants, and so on. If you passed the trap the wrong arguments, it did no error checking.

This file is still available for backward compatibility. New code should use the new implementation of traps, described immediately following.

Macintosh Common Lisp version 2 has changed the implementation of Macintosh Operating System calls by making information about them available in interface files. Every trap, constant, record, and record field type is now described in one of these files (located in the Interfaces folder within the Library folder). When one of these operating system calls is referred to and the value of `*autoload-traps*` is true, information is automatically read from the relevant interface file when the MCL reader reads a trap-call form. Macintosh Common Lisp knows what arguments to expect and can do error checking on argument types. You simply give Macintosh Common Lisp the correct arguments for the trap or record, as described in its definition in *Inside Macintosh*.

MCL interface files currently define all calls to the Macintosh Operating System for over 2000 records, traps, record field types, and constants used with the Macintosh computer.

Names of record formats and trap calls in Macintosh Common Lisp correspond to those in MPW and *Inside Macintosh* . In Macintosh Common Lisp, however, calls to the operating system are indicated by the reader macros number sign–underscore `#_` or number sign–dollar sign `#$`. (See the section "Loading Trap Definitions" later in this chapter.)

You can define your own record formats with `defrecord` and your own trap calls with `deftrap`. These macros are also described in this chapter.

Traps

On the Macintosh, the Toolbox and operating system reside in ROM. However, to allow flexibility for future development, application code must be kept free of specific ROM addresses, so all references to Toolbox and operating system routines must be made indirectly through a **trap dispatch table.** To issue a call in assembly language to the Toolbox or operating system, you use a **trap macro** defined in a set of macro files.

When you assemble your program, the macro generates a **trap word.** Instruction words beginning with `$A` do not correspond to valid machine-language instructions. Instead they augment the MC68000 microprocessor's native instruction set with additional operations specific to the Macintosh computer.

An attempt to execute any instruction word beginning with `$A` causes a trap to the trap dispatcher, which determines what operation the trap word stands for, looks up the address of the corresponding routine in the trap dispatch table, and jumps to the routine.

There are three ways to call traps in Macintosh Common Lisp:

- through the Interface files, defining trap calls for virtually all Macintosh Operating System calls in ROM (Function definitions that implement calls marked "not in ROM" are provided in the file `interfaces.lisp` in the MCL Library folder.)
- through the macro `deftrap`, which provides a high-level mechanism for trap calling
- through the functions `stack-trap`, `register-trap`, and `%gen-trap`, which provide low-level trap calling

u *Note:* The old functions `register-trap` and `stack-trap` are provided for backward compatibility and may be removed from future versions of Macintosh Common Lisp. However, they continue to work and are documented in the section "Trap Calls Using `stack-trap` and `register-trap`." In addition, since Macintosh traps may now use values in both stacks and registers, or values in multiple registers, the new function `%gen-trap` has been provided. It allows for more general parameter-passing and value-return combinations.

Loading trap definitions

In the Interfaces folder, a subfolder of Library, are source files giving definitions of over 2000 Macintosh traps described in *Inside Macintosh*.

If you reference a known Macintosh trap and the value of `*autoload-traps*` is true (the default), the trap is automatically loaded.

If you want to load an entire interface file, use the function `require-interface`:

```
? (require-interface 'quickdraw)
"QUICKDRAW"
```

If you use an unusual selection of traps, you can create your own interface files, containing only those traps you use.

u *Note:* The autoloading mechanism uses the index files in the folder `ccl:library;interfaces;index;` to find trap and constant definitions. If you modify any of the interface files, you must evaluate the form `(reindex-interfaces)` to update the index files.

The following two reader macros apply to traps.

#_symbol

[*Reader macro*]

Syntax *#_symbol*

Description If the value of `*autoload-traps*` is true, the `#_` reader macro tries to load the trap definition of *symbol* from the appropriate interface file and interns *_symbol* in the `traps` package. For example, `#_NewPtr` loads `_NewPtr` and interns the symbol `_NewPtr` in the `traps` package.

#$symbol

[*Reader macro*]

Syntax #$*symbol*

Description If the value of `*autoload-traps*` is true, the `#$` reader macro tries to load a constant definition for *symbol* from the proper interface file. It also interns $*symbol* in the `traps` package. For example, `#$WMgrPort` loads the constant `$WMgrPort` and interns the symbol `$WMgrPort` in the `traps` package.

Since `#_` and `#$` are reader macros, they are evaluated only at read time. Therefore, if you include autoloaded symbols in a macroexpansion, you must use `require-trap` or `require-trap-constant` .

require-trap

[*Macro*]

Syntax `require-trap` *trap-name* `&rest` *rest*

Description The `require-trap` macro autoloads a trap whether called at read time or at macroexpand time.

Arguments

trap-name	A trap name.
rest	A list of other arguments.

Example

```
? (defmacro draw-string (string start end)
    (let ((pstr (gensym)))
      `(with-pstr (,pstr ,string ,start ,end)
         (require-trap #_DrawString ,pstr))))
DRAW-STRING
```

require-trap-constant

[*Macro*]

Syntax `require-trap-constant` *trap-name*

Description The `require-trap-constant` macro autoloads a constant from the interface file, whether called at read time or at macroexpand time.

Argument *trap-name* The name of a trap constant.

reindex-interfaces [*Function*]

Syntax `reindex-interfaces`

Description The `reindex-interfaces` function updates the interface index files.

Defining a trap

The macro `deftrap` defines the behavior of a trap. Arguments can be explicitly typed so as to override the definition; for example, an argument may be passed as two words instead of as a single longword.

After performing compile-time type checking on the arguments, high-level traps expand into `stack-trap` , `register-trap` , or `%gen-trap` (discussed later in this chapter in the section “Lisp Macros for Calling Macintosh Traps”).

deftrap [*Macro*]

Syntax `deftrap` {*trap-name* } ([({*arg* }{*mactype* })]*) ({*return-place* }{*mactype* }) | `nil`) {*implementation-form* }+)

Description The `deftrap` macro makes *trap-name* into a symbol for a trap and defines the behavior of the trap.

Arguments

trap-name The name of the trap.

arg Any symbol whose keyword is not a record field type.

Type checking on the arguments is performed at compile time. Run-time type checking on the Lisp data types of the arguments is also performed if an optimize declaration of `(safety 3)` is in effect. Run-time type checking is always performed on pointer and long arguments.

mactype A record field type. (See the section “Defining Record Types” later in this chapter.)

return-place An argument specifying where the results of the trap call are returned. The value of *return-place* is `:stack`, a register, or `nil`, which indicates that no value is returned.

implementation-form Any collection of code, with the proviso that one subform of *implementation-form* must be of the form `({`*trap-kind*`}{`*trap-number*`}{`*call-arg*`}*)`.

trap-kind One of: `:trap`, `:stack-trap`, `:register-trap`, or `:no-trap`. When *trap-kind* is `:stack-trap`, *call-arg* can be left off, in which case the *args* of the `deftrap` are used instead. If you specify `:trap`, `deftrap` will generate the correct kind of trap.

trap-number A fixnum, or, in the case of `:no-trap`, a form to be executed in lieu of a trap call.

call-arg If present, either a symbol or `{`*register-key*`}{`*value*`}`. If there is no *call-arg*, `([({`*arg*`}{`*mactype*`})]*)` is used. If *call-arg* is a symbol, it must be a symbol from `([({`*arg*`}{`*mactype*`})]*)`. Otherwise, any number of register keys and values may appear.

Example

The macro `deftrap` is used extensively in the files in the Interfaces subfolder of your MCL Library folder.

Examples of calling traps

This section gives several examples of calling traps. All of these traps make ephemeral records with `rlet`, which is used to create a record for the duration of a procedure call.

The following code creates a new window and draws inside it, using Macintosh Operating System traps that require records.

First, create the window. (Using `defparameter` ensures that `*w*` is reevaluated whenever it is recompiled.)

```
? (defparameter *w* (make-instance 'window))
```

Within Lisp, call the `PaintRoundRect` procedure to draw and fill a rectangle inside the window. This *Inside Macintosh* definition of this procedure is as follows:

```
PROCEDURE PaintRoundRect (r: Rect; ovalWidth, ovalHeight: INTEGER);
```

The data type of the first argument, `r`, is `:rect` (see *Inside Macintosh*), so you must define a record with `rlet` or `make-record` . Then call the trap from within Macintosh Common Lisp using `#_PaintRoundRect` , which corresponds to the `PaintRoundRect` procedure.

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

A call to the trap `PtToAngle` (see *Inside Macintosh*) gets a result by passing an argument by reference:

```
? (rlet ((angle :integer)
         (r :rect :topleft #@(100 50)
                          :bottomright #@(120 70)))
        (with-focused-view *w*
          (#_Framerect r)
          (#_moveto 110 60)
          (#_lineto 150 20)
          (#_PtToAngle r #@(150 20) angle)
          (%get-word angle)))
```

This trap call creates a record to hold the result of the call to `StuffHex` (see *Inside Macintosh*), which translates a Pascal string into binary data. It creates another record to call the rectangle required by `FillOval` , and finally it draws the oval with the pattern in the window:

```
? (rlet ((pat :pattern))
    (with-pstrs ((hex-string "1020408102040801"))
      (#_stuffhex pat hex-string)
      (rlet ((r :rect :topleft #@(200 20)
                      :bottomright #@(250 100)))
        (with-focused-view *w*
          (#_filloval r pat)))))
```

Determining record types and record field types

When you are creating a trap, you must know the data types of the arguments. You can determine them by consulting the trap definition in *Inside Macintosh*. The data type of all arguments is shown at the end of each chapter.

You can also use the functions `find-mactype` and `find-record-descriptor` or Table 17-1.

Table 17-1 lists the MCL equivalents of the most frequently used Pascal types.

n **Table 17-1** Pascal types and their equivalent MCL types

Pascal type	Lisp type
array	`macptr`
boolean	`t` or `nil`
byte	`fixnum`
char	Lisp character (`#\`*char*)
handle	`macptr`
integer	`fixnum`
longword	integer
os-type	`fixnum`, string, or symbol usually a symbol, such as `':|STR |`
point	`fixnum` or point made with `#@(h v)`
pointer	`macptr`
string	`macptr` made by `with-pstrs`
Anything passed by reference (VAR; can be made with `rlet` or `%stack-block`)	`macptr`
Other records, i.e., a record made with `make-record` or `rlet`	`macptr`

`find-record-descriptor` [*Function*]

Syntax `find-record-descriptor` *record-type* `&optional` *errorp autoload*

Description The `find-record-descriptor` function returns the record type of *record-type*.

Arguments

record-type	A record type.
errorp	An argument determining whether an error is signaled if the value of this argument is true and *record-type* is not a valid record type. The default value is true.
autoload	An argument determining whether to load *record-type*. The default is true.

find-mactype [*Function*]

Syntax `find-mactype` *mactype* `&optional` *errorp autoload*

Description The `find-mactype` function returns the record field type of *mactype* .

Arguments

mactype	A record field type.
errorp	An argument determining whether an error is signaled if the value of this argument is true and *mactype* is not a valid record type. The default value is true.
autoload	An argument determining whether to load*mactype* . The default is true.

record-length [*Macro*]

Syntax `record-length` *record-type*

Description The `record-length` macro returns an integer representing the length of *record-type* .

Argument

record-type	A record type.

record-fields [*Function*]

Syntax `record-fields` *record-type*

Description The `record-fields` function returns a list of the fields in *record-type* .

Argument

record-type	A record type.

record-info

[*Function*]

Syntax `record-info` *record-type* `&optional` *error-p*

Description The `record-info` function returns a list of the offset, type, and length of each field in *record-type*.

Arguments

record-type A record type.

error-p An argument specifying the behavior of the function if *record-type* does not name a record type. If this parameter is specified and true, `record-info` signals an error; otherwise it returns `nil`.

field-info

[*Function*]

Syntax `field-info` *record-type field-name*

Description The `field-info` function returns the offset, type, and length of the field *field-name* of *record-type*.

Arguments

record-type A record type.

field-name A field name.

Records

Records can be viewed from two perspectives: how they are stored and used and how they are passed by Lisp.

Records keep track of blocks of Macintosh memory within Macintosh Common Lisp. As stored and used, a **record** is a contiguous structured block of memory of a specific size, stored on the stack or Macintosh heap. As passed around by Lisp, a record is a simple pointer to Macintosh memory, with no formatting or length information.

To use a record, a program must provide a *record type*. This record type tells the system how the data at the other end of the pointer should be interpreted. Your program must keep track of the types of all the records you create.

Records have no explicit type, so you can map over a single block of memory in several different ways, as if it were several different types of record. This is convenient, for example, in the case of a window pointer, whose first section is a GrafPort record. The system also allows you to use pointers returned by Macintosh traps as records.

In the following discussion, the word *record* can mean either a block of memory or a pointer to memory, depending on the context. For example, when you allocate a new record with `make-record`, a block of memory is allocated on the heap, and a pointer to the block is returned. For the sake of brevity, this process is described in this way: a record is allocated and returned.

Installing record definitions

In the Interfaces folder, a subfolder of Library, are source files giving definitions of many of the Macintosh records described in *Inside Macintosh*.

If you reference a known Macintosh record and the value of `*autoload-traps*` is true, the record definition is automatically loaded.

Other records may be defined with the macro `defrecord`.

The structure of records

A record has an associated set of **fields** that refer to different portions of the memory block. A record **definition** is a template that defines the fields for a specific type of record.

Each field has a name, a type, and a byte offset into the record. Field names are used to access portions of a record symbolically. Field types are used to determine the size of each field and the way the information in the field is encoded and decoded (for example, a field may itself be a record and therefore contain subfields). Field offsets indicate the position of the field inside the record.

Here is an example of a record definition. It has a name, `foo`, and two fields, `str` and `array`. The field `str` is a string of length 255 and the field `array` is an array of 100 integers:

```
(defrecord (foo :handle)
  (str (:string 255))
  (array (:array :integer 100)))
```

You can access the same portion of a record in different ways by using variant fields. See the section “Variant Fields” later in this chapter.

Defining record types

Many standard record types are already defined in the Interface files. However, you can also define your own record types with `defrecord`.

`defrecord` [*Macro*]

Syntax `defrecord` *record-name* `&rest` *slot-descriptions*

Description The `defrecord` macro defines a new record type.

Arguments

record-name Either a symbol that will be used to name the type of record, or a list whose `car` is the symbol used to name the record and whose `cadr` is a keyword that specifies the default type of storage used for the record. See the note at the end of this definition on overriding default record storage.

The package of the symbol used to name the record is ignored; all record names are converted to the keyword package.

The keyword should be either `:pointer` or `:handle`. If the keyword is `:pointer`, records of this record type are allocated and accessed as pointers. If the keyword is `:handle`, they are allocated and accessed as handles. If no keyword is given, the record type is assumed to be `:pointer`.

slot-descriptions

One or more slot descriptions. A standard slot description is a list of the form `(` *name type* `{`*option*`}*)`.

name The name used to access the field in the record. The name cannot be `variant`, which has special meaning (see the next section, “Variant Fields”).

type The type of data the field contains; *type* is used to determine the field's length. The *type* must be one of the predefined types (see Table 17-2), a previously defined record type, an array, or a list whose `car` is the symbol `:string` and whose `cadr` is a fixnum from `1` to `255`, which is used to specify the length of the string.

An array type is of the form `(:array` *type dimensions* `+)`, where *type* is defined as in this example and *dimensions* are constant fixnums:

```
(defrecord (foo :handle)
  (str (:string 255))
  (array (:array :integer 100)))
```

option Zero or more of the following:

`:offset` *number*
A fixnum to offset the slot from the beginning of the record.

`:include` *t-or-nil*
A keyword, which can be used only when the slot type is another record. It indicates whether the fields of the other record can be accessed directly, as if they were fields of the record being defined. If the value of this keyword is `nil` (the default), they cannot be accessed directly. If the value is true, they can. In the first slot description, the value of `:include` is automatically true.

Examples

Here are two examples of the syntax of `defrecord`.

```
? (defrecord PenState
    (:pnLoc  point)
    (:pnSize point)
    (:pnMode integer)
    (:pnPat  pattern))
```

This call, one of the calls in the Interface files, creates a record type called `PenState` with four slots.

```
? (defrecord foo
    (field1 :integer :default 42)
    (field2 (array :longint 10))
    (field3 (array :byte 5 5))
    (field4 (some-record-type :handle))
    (field5 (:string 255)))
```

This call creates a new record type called `foo`. The fourth field of this record type is stored as a handle. Records stored as handles are less likely to cause fragmentation of the Macintosh heap, but you must be careful when using them.

Important The Macintosh ROM is very strict about whether records are passed to it by handle or by pointer. It is recommended that you explicitly specify the storage type by using the `:storage` keyword in calls to `make-record` and `rref` or by using the `href` or `pref` macros.

MCL records correspond exactly to MPW Pascal packed records, except that Boolean fields always take up a full byte. Fields that are 2 or more bytes long always begin at word boundaries (that is, at even memory locations). Fields that are 1 byte long are padded to 2 bytes if necessary. See *Inside Macintosh* for more details on field size.

Table 17-2 lists the predefined record field types and their lengths.

Table 17-2 Predefined record field types and their lengths

Lisp type	Length in bytes	Equivalent to
`array`	4	`pointer`
`boolean`	1	
`byte`	1	`unsigned-byte`
`signed-byte`	1	
`unsigned-byte`	1	`byte`
`character`	1	
`handle`	4	
`integer`	2	`unsigned-integer`
`signed-integer`	2	
`unsigned-integer`	2	`integer`
`long`	4	`signed-long`
`signed-long`	4	
`unsigned-long`	4	
`longint`	4	`signed-long`

(continued)

Table 17-2 Predefined record field types and their lengths (continued)

Lisp type	Length in bytes	Equivalent to
`signed-longint`	4	`signed-long`
`unsigned-longint`	4	`unsigned-long`
`ostype`	4	
`point`	4	
`pointer`	4	
`short`	2	`unsigned-integer`
`signed-short`	2	`signed-integer`
`unsigned-short`	2	`unsigned-integer`
`string`	4	
`word`	2	`unsigned-integer`
`signed-word`	2	`signed-integer`
`unsigned-word`	2	`unsigned-integer`

Variant fields

You can use **variant fields** to access the same portion of a record in different ways. Variant fields allow an area of a record to be mapped to different sets of fields. For example, you can use variant fields to access one part of a record as a single longword or as 4 bytes. Variant fields (like records in general) are useful mnemonic aids and short cuts. The size of a variant field is equal to the total size of the largest set of fields in the variant portion.

If a field description contains variant fields, it will have the form

```
(:variant ({ variant-field1 }+)
          ({ variant-field2 }+)
          ...
          ({ variant-fieldN }+))
```

The section of the record described by the variant may be accessed through *N* sets of fields. The size of the variant field is equal to the size of the largest set of fields.

You can specify an `:origin` keyword argument for a field. The keyword `:origin` simply sets the offset counter.

The following code indicates that a `rect` record may be accessed either as two points or as four coordinates:

```
(defrecord Rect
           (:variant ((top :integer)
                      (left :integer))
                     ((topleft :point :origin 0)))
           (:variant ((bottom :integer)
                      (right :integer))
                     ((bottomright :point :origin 4))))
```

A variant field list can itself use variants.

Creating records

Records may be created temporarily, for example, within a function, using `rlet`, or with indefinite extent, using `make-record`. Records with indefinite extent must be disposed of explicitly. Temporary records are much more efficient.

Creating temporary records with `rlet`

The macro `rlet` is used when memory needs to be allocated temporarily.

`rlet` [*Macro*]

Syntax `rlet ({`*symbol record-type init-forms* `*}+) {`*form*`}*`

Description The `rlet` macro creates a temporary record on the stack and evaluates *form*. The value of the last *form* is returned. The `rlet` macro is the most efficient way to create temporary records.

The records are stored on the stack and are therefore ephemeral. When the evaluation of *forms* is done, all the records vanish irretrievably and should not be referenced. (This macro is similar to `%stack-block`, to which `rlet` macroexpands.)

The `rlet` macro has the same general form as the `let` macro. For every *symbol* a new binding is created. Space is allocated on the stack for a *record-type* record, and the record is initialized according to the *init-form*, which should be pairs of *field-keyword* and *value*. A pointer to the new record is the value of the corresponding *symbol*. The *forms* are evaluated in the resulting environment, and the value of the last *form* is returned.

If the value of `*autoload-traps*` is true, `rlet` automatically loads the definition of the record.

Arguments

symbol	A symbol.
record-type	A record type.
init-forms	Zero or more initial value forms.
forms	Zero or more forms.

Examples

This example of `rlet` allocates space on the stack for a record of type `:rect`, initializes it with its `:topleft` and `:bottomright` values, and evaluates `(#_framerect r)`. It draws a rectangle into the current GrafPort, the window `foo`:

```
? (setq foo (make-instance 'window))
#<WINDOW "Untitled", #x352819>
? (with-focused-view foo
    (rlet ((r :rect
              :topleft #@(10 10)
              :bottomright #@(100 100)))
      (#_framerect r)))
NIL
```

The binding is ephemeral; `r` no longer has a binding after the value of the last form is returned:

```
? r
> Error: Unbound variable: R
> While executing: SYMBOL-VALUE
```

Here is an example of the expansion of `rlet`.

```
(rlet ((r :rect :topleft #@(10 10) :bottomright #@(100 100)))
  (#_framerect r))
```

macroexpands to

```
(%stack-block ((r 8))
  (ccl::%put-point r 655370 0)
  (ccl::%put-point r 6553700 4)
  (traps:_framerect r))
```

and then to

```
(let* ((r (ccl::%new-ptr 8)))
  (declare (dynamic-extent r))
  (ccl::%put-point r 655370 0)
  (ccl::%put-point r 6553700 4)
  (traps:_framerect r))
```

If you use the `rlet` macro to allocate a record with `:storage :handle`, it acts as though you overrode the allocation to `:storage :pointer`.

Important If you override the default storage with `:storage :pointer`, you should use the pointer-specific macros `%ref` and `%set` to access the record (or be careful to always specify `:storage :pointer` in `rref` and `rset`). Doing otherwise may cause a crash.

The *record-type* may also be a record field type. In that case, `rlet` allocates enough storage for one of the specified record fields. For example, the call

```
(rlet ((p :point))...)
```

allocates enough storage to hold a point (that is, 4 bytes). When you allocate storage using record field types, you cannot specify the initial contents.

Creating records with indefinite extent

When you want to return a record from a function, you must create a record that has indefinite extent. The memory this record takes up is not subject to automatic garbage collection; it uses space in the Macintosh heap until it is explicitly disposed of.

Some records, such as windows, must be created and initialized by specific Toolbox routines. Such records should be created not by using `make-record` but by using the appropriate Toolbox traps.

The following functions control records with indefinite extent.

make-record [*Macro*]

Syntax `make-record` *record-name* `&rest` *initforms*

Description The `make-record` macro allocates space in the Macintosh heap for a record*record-name* and returns a pointer or a handle to it. This record has indefinite extent (as opposed to blocks allocated with `%stack-block` or `rlet`).

Arguments

record-name Either a symbol that will be used to name the type of record, or a list whose`car` is the symbol used to name the record and whose`cadr` is a keyword that specifies the default type of storage used for the record. The keyword should be one of the following:

`:storage` A keyword used to override the default storage method used by the record. If specified, this should be `:pointer` or `:handle`.

Important It is recommended that you always specify the storage explicitly, rather than relying on the default. A crash is very likely if a handle is used as a pointer or vice versa.

`:clear` A keyword determining how*record-name* is cleared. If the value of `:clear` is true, clears the entire record, using an efficient operating system call to allocate and clear the pointer or handle simultaneously.

`:length` A fixnum. Used to override the default size used by the record. There is no error checking to see whether `:length` is long enough.

initforms A list of keywords and values used to initialize the record.

dispose-record [*Macro*]

Syntax `dispose-record` *record* `&optional` *storage-type*

Description The `dispose-record` macro disposes of *record* and returns `nil`.

If *record* contains pointers to other records, `dispose-record` does not automatically dispose of these other records, since other pointers to those records may exist.

Any Macintosh Toolbox data structure that is allocated using a special trap (such as regions and controls) also needs to be deallocated using a special trap, rather than by `dispose-record`. In general, if you did not use `make-record` to create the data structure, you should not dispose of it with `dispose-record`.

The `dispose-record` macro has an effect only if *record* is a pointer or a handle to a Macintosh Memory Manager block.

Arguments

record	A record.
storage-type	The type of the record being disposed. This may be either a record type or storage (`:pointer` or `:handle`). Supplying this argument allows the macro to expand into more efficient code.

Using records

The following macros and functions control the use of records.

href [*Macro*]

Syntax `href` *handle accessor*

Description The `href` macro returns the contents of the specified field of *handle*. This macro is the most efficient way to access the fields of a record.

If the value of `*autoload-traps*` is true, `href` automatically loads the record definition.

An error is signaled at macroexpansion time if an attempt is made to get a handle to a record and the surrounding record is stored as a pointer.

The `href` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try experimenting with `href` expansions to see how this macro works.

The macro `href` may be combined with `setf` to modify a field of a record.

Arguments

handle A handle to a record.

accessor A form that describes which record type and field to access; *accessor* has the form *record-type{.field}*$^{+}$. An array reference has the form (*record-type {.field}*$^{+}$ *indices* `*`), for example:

```
(defrecord (foo :handle)
   (str (:string 255))
   (array (:array :integer 100)))

(href f (foo.array 42))
```

record-type A previously defined record type.

field A field in a record of type *record-type*. If *field* is also a record type, its fields may be accessed by appending an additional period and field name. If that field is a record type, the process can continue. There can be any number of *fields*. Every one but the last must be a record type; the last one may be a record type, but is not required to be.

In cases where the first field of *record-type* is also a record, then that field's fields may be referred to as if they were direct fields of *record-type*. For example, a QuickDraw GrafPort is the first field of a `windowRecord` record, so the GrafPort's `portrect` can be abbreviated from `windowRecord.grafport.portrect` to `windowRecord.portrect`.

If the final *field* of *accessor* is not a record, then the actual field value is returned. If the final *field* of *accessor* is itself a record, then a pointer to that record is returned.

If you do not specify enough array indices, Macintosh Common Lisp returns a pointer to the location in memory where the subarray would begin.

---

pref

[*Macro*]

Syntax `pref` *pointer accessor*

Description The `pref` macro returns the contents of the specified field of *pointer*. This macro is the most efficient way to access the fields of a record.

If the value of `*autoload-traps*` is true, `pref` automatically loads the record definition.

An error is signaled at macroexpansion time if the surrounding record is stored as a handle.

The `pref` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try experimenting with `pref` expansions to see how this macro works.

The macro `pref` may be combined with `setf` to modify a field of a record.

Arguments

pointer A pointer to a record.

accessor A form that describes which record type and field to access; *accessor* has the form *record-type{.field}* +. An array reference has the form (*record-type {.field}* + *indices* `*`), for example:

```
(defrecord (foo :pointer)
   (str (:string 255))
   (array (:array :integer 100)))

(pref f (foo.array 42))
```

record-type A previously defined record type.

field A field in a record of type *record-type*. If *field* is also a record type, its fields may be accessed by appending an additional period and field name. If that field is a record type, the process can continue. There can be any number of *fields*. Every one but the last must be a record type; the last one may be a record type, but is not required to be.

In cases where the first field of *record-type* is also a record, then that field's fields may be referred to as if they were direct fields of *record-type*. For example, a QuickDraw GrafPort is the first field of a `windowRecord` record, so the GrafPort's `portrect` can be abbreviated from `windowRecord.grafport.portrect` to `windowRecord.portrect`.

If the final *field* of *accessor* is not a record, then the actual field value is returned. If the final *field* of *accessor* is itself a record, then a pointer to that record is returned.

If you do not specify enough array indices, Macintosh Common Lisp returns a pointer to the location in memory where the subarray would begin.

`rref` [*Macro*]

Syntax `rref` *record accessor* `&key :storage`

Description The `rref` macro returns the contents of the specified field of *record*. This macro is the most efficient way to access the fields of a record.

If the value of `*autoload-traps*` is true, `rref` automatically loads the record definition.

An error is signaled at macroexpansion time if an attempt is made to get a pointer to a record and the surrounding record is stored as a handle. If this is desired, use a `with-dereferenced-handles` form around the call and specify `:storage` to be `:pointer`. Such a pointer to a record within a handle is valid for the duration of `with-dereferenced-handles`.

The `rref` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try experimenting with `rref` expansions to see how this macro works.

Combined with `setf`, `rref` is equivalent to `rset`.

Arguments

record A pointer or handle to a record.

accessor A form that describes which record type and field to access; *accessor* has the form *record-type{.field}* +. An array reference has the form (*record-type {.field}* + *indices* `*`), for example:

```
(defrecord (foo :handle)
  (str (:string 255))
  (array (:array :integer 100)))

(rref f (foo.array 42))
```

record-type A previously defined record type.

field A field in a record of type *record-type*. If *field* is also a record type, its fields may be accessed by appending an additional period and field name. If that field is a record type, the process can continue. There can be any number of *fields*. Every one but the last must be a record type; the last one may be a record type, but is not required to be.

In cases where the first field of *record-type* is also a record, then that field's fields may be referred to as if they were direct fields of *record-type*. For example, a QuickDraw GrafPort is the first field of a `windowRecord` record, so the GrafPort's `portrect` can be abbreviated from `windowRecord.grafport.portrect` to `windowRecord.portrect`.

If the final *field* of *accessor* is not a record, then the actual field value is returned. If the final *field* of *accessor* is itself a record, then a pointer to that record is returned.

If you do not specify enough array indices, Macintosh Common Lisp returns a pointer to the location in memory where the subarray would begin.

`:storage` The storage method used for the record. It should be `:pointer` or `:handle`. If omitted, the default storage type for the specified record type is used. It is recommended that you always explicitly specify the storage type.

Important The storage type used with `make-record` when a record is created must be the same as the storage type specified by any calls to `rref` or `rset` for that record. A crash is very likely if a handle is referenced as a pointer or vice versa.

Examples

Here are some examples of using `rref`:

```
(rref my-rect :rect.top)
(rref wptr :windowRecord.portrect.bottomright)
(rref tePtr :terec.viewrect.left :storage :pointer)
(rref my-control :control.controlvalue)
```

rset

[*Macro*]

Syntax `rset` *record accessor value* `&key :storage`

Description The `rset` macro sets the value of a field in a record.

If the value of `*autoload-traps*` is true, `rset` automatically loads the record definition.

The `rset` macro is very efficient. It expands into a simple call to a low-level memory-accessing function that is in turn compiled inline. Try expanding `rset` to see how it works.

Combined with `setf`, `rref` is equivalent to `rset`.

Arguments

record A pointer or handle to a record.

accessor A form that describes which record type and field to set. For a complete description of record accessors, see `rref`.

value The new value to store in the field. If the final *field* of *accessor* is a record, *value* must also be a record (either a handle or a pointer) that is copied into the appropriate field of *record*.

Attempting to put a new value into an underspecified array reference gets a run-time error.

`:storage` The storage method used for the record. It should be either `:pointer` or `:handle`. If omitted, the default storage type for the specified record type is used. It is recommended that you always explicitly specify the storage type.

Important The storage type used with `make-record` when a record is created must be the same as the storage type specified by any calls to `rref` or `rset` for that record. A crash is very likely if a handle is referenced as a pointer or vice versa.

Examples

```
(rset wptr :window.portrect.topleft #@(100 200))
(rset my-rect :rect.left -10)
(rset teptr :terec.viewrect.top 50 :storage :pointer)
(rset my-control :control.controlvalue 200)
```

raref [*Macro*]

Syntax `raref` *record array-descriptor* `&rest` *indices*

Description The `raref` macro accesses an array inside a record. If `raref` is not passed enough indices, it returns a pointer to the place in memory indicated by the indices you have passed. This macro operates on that place in memory and returns the contents of the specified field of the specified record.

Arguments

record A pointer to a record.

array-descriptor A description of the type and dimensions of the array, of the form (*type* {*dimension* }*).

indices A set of indices to a location in memory.

Example

Assume that you have a record type `foo`.

```
(defrecord foo
    (field1 :integer :default 42)
    (field2 (array :longint 10))
    (field3 (array :byte 5 5))
    (field4 (some-record-type :handle))
    (field5 (:string 255)))
```

The following three expressions are equivalent:

```
(rref ptr (foo.field3 2 4))
(raref (rref ptr foo.field3) (:byte 5 5) 2 4)
(raref (rref ptr (foo.field3 2)) (:byte 5) 4)
```

rarset

[*Macro*]

Syntax `rarset` *record value array-descriptor* `&rest` *indices*

Description The `rarset` macro sets the value of an array index inside a record and returns the new contents.

Arguments

record	A pointer to a record.
value	The new value of the contents of the specified field.
array-descriptor	A description of the type and dimensions of the array, of the form (*type* {*dimension* }*).
indices	A set of indices to a location in memory.

clear-record

[*Macro*]

Syntax `clear-record` *record* `&optional` *storage-type*

Description The `clear-record` macro clears *record* and returns `nil`.

Arguments

record	A record.
storage-type	The type of the record being cleared. This may be either a record type or storage (`:pointer` or `:handle`). Supplying this argument allows the macro to expand into more efficient code. If specified, the storage must match the storage given to or defaulted by `make-record`; otherwise, it may cause a crash.

copy-record

[*Macro*]

Syntax `copy-record` *source-record* `&optional` *record-type dest-record*

Description The `copy-record` macro copies all of the fields of *source-record* (which should be of type *record-type*) into *dest-record* and returns *dest-record*.

Arguments

source-record A record of type *record-type*.

record-type Any previously defined record type.

If the default storage for *source-record* has been overridden, you should not specify *record-type* as well. If you specify *record-type*, then *source-record* and *dest-record* must have been allocated with the default storage. No error checking is performed.

If *dest-record* requires a value but *record-type* does not, give *record-type* the value `nil`.

dest-record A record of type *record-type*, if *record-type* is supplied; if not, a record of the same storage type as *source-record*.

If *dest-record* is not specified, a new one is allocated. If *dest-record* is specified, `copy-record` correctly copies the contents of *source-record* into *dest-record*, even when the storage types of *source-record* and *dest-record* are different.

get-record-field

[*Function*]

Syntax `get-record-field` *record record-type field-name*

Description The `get-record-field` function returns the value of the *field-name* field of *record*. The `get-record-field` function is much less efficient than the `rref` macro. It should be used only when a function (rather than a macro) is needed, for example, when the record type or field name is not known at compile time.

Unlike the macros listed in this chapter, `get-record-field` needs access to record definitions at run time.

Arguments

record A record.

record-type Any record type, given as a keyword.

field-name The field of the record to be accessed, given as a keyword.

set-record-field [*Function*]

Syntax `set-record-field` *record record-type field-name value*

Description The `set-record-field` function sets the value of the *field-name* field of *record* and returns `nil`. The `set-record-field` function is much less efficient than the `rset` macro. It should be used only when a function (rather than a macro) is needed, for example, when the record type or field name is not known at compile time.

Unlike the macros listed in this chapter, `set-record-field` needs access to record definitions at run time.

Arguments

record	A record.
record-type	Any record type, given as a keyword.
field-name	The field of the record to be accessed, given as a keyword.
value	The new value to be placed in the field.

Getting information about records

The following functions give information on records. They are provided primarily for use during development and debugging of programs that use records.

record-types [*Variable*]

Description The `*record-types*` variable contains a list of all the types of records that are currently defined in the MCL environment.

mactypes [*Variable*]

Description The `*mactypes*` variable contains a list of all the field types for use in records. (Note that this list does not include record types themselves, which can also be used as field types.)

print-record

[*Function*]

Syntax `print-record` *record record-type* `&optional` *currlevel*

Description The `print-record` function prints the values of the fields of *record* of type *record-type* . No values are returned.

The `print-record` function uses the values of `*print-length*` and `*print-level*` .

Arguments

record	A record.
record-type	Any record type, given as a keyword.
currlevel	The current print level. The default is 0.

handle-locked-p

[*Function*]

Syntax `handle-locked-p` *handle*

Description The `handle-locked-p` function returns `t` if *handle* is locked, `nil` if it is not.

Trap calls using `stack-trap` and `register-trap`

Most older Macintosh traps accept arguments either on the stack or in registers, but not in both places. Some newer traps accept arguments through both. In general, the operating system traps are register based and the Toolbox traps are stack based, but there are exceptions.

Within a single trap call, some arguments are passed as immediate values and some are passed by reference (that is, a pointer to the value is passed). In general, data 4 bytes long or less is passed by value, and data longer than 4 bytes is passed by reference. Check *Inside Macintosh* for the calling sequences of particular traps. Arguments passed by reference (Pascal VAR parameters) may be modified by the trap.

For each trap `TrapName` , the file `traps.lisp` in the Library folder defines a constant `_TrapName` (equal to the trap number) and a macro `TrapName` that expands into the trap call and is recognized by the compiler. You can use `stack-trap`, `register-trap` , and `%gen-trap` to call any trap.

Low-level stack trap calls

Here is the general format of a stack trap call.

_TrapName [*Macro*]

Syntax *_TrapName* {*type-keyword argument* }* [*return-value-keyword*]

Description The arguments are evaluated, coerced according to *type-keyword* , and passed as the arguments to the trap.

Arguments

_TrapName The name of the trap.

type-keyword A keyword signifying the type of coercion to be performed on the corresponding *argument*. Possible *type-keywords* are `:word`, `:long`, `:ostype` , `:ptr`, and `:d0`. The keywords operate on the subsequent *argument* according to the following list.

- `:word` This keyword causes *argument* to be passed as a 16-bit word. Arguments passed as words should be fixnums; it is an error if *argument* has more than 16 significant bits.
- `:long` This keyword causes *argument* to be passed as a 32-bit longword, and is equivalent to `:ostype` , next.
- `:ostype` This keyword causes *argument*, as a four-character string or symbol with a four-character print name, to be passed as a 32-bit value (8 bits for each of the characters); os-types are used as identifiers by the Resource Manager, Scrap Manager, and other parts of the Macintosh Operating System.
- `:ptr` This keyword causes *argument* to be passed as a 32-bit macptr.
- `:d0` This keyword causes *argument* to be passed in the `d0` register.
- `:boolean` This keyword causes *argument* to be passed as a Boolean value.

argument An argument to be passed to the stack. As noted previously, *argument* should evaluate to 32-bit values or to macptrs to data on the Macintosh heap or the stack.

return-value-keyword

Indicates the type of value returned by the trap. If *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

`:word`	A 16-bit result is read from the stack, sign-extended, and returned as a signed Lisp integer.
`:long`	A 32-bit result is read from the stack and returned as a signed Lisp integer.
`:ptr`	A 32-bit result is returned from the stack as a macptr.
`:novalue`	The Macintosh trap does not return a value. The Lisp call returns `nil`.
`:boolean`	A Boolean value is returned from the stack.

Examples

This form calls the trap `FrameRoundRect` with three arguments: a pointer and two words. The value `nil` is returned.

```
? (let ((oval-width 12)
        (oval-height 8))
    (%stack-block ((my-rect 8))                    ;rectangles are 8 bytes
                  (%put-word my-rect 12 0)         ;top=12
                  (%put-word my-rect 40 2)         ;left=40
                  (%put-word my-rect 32 4)         ;bottom=32
                  (%put-word my-rect 80 6)         ;right=80
                  (_FrameRoundRect:ptr my-rect
                                       :word oval-width
                                       :word oval-height)))
```

The following two forms are equivalent, although the first is much easier to read:

```
? (_GetResource :ostype "STR#"    ;ASCII "STR#"
                :word 15          ;resource number 15
                :ptr)             ;return value is pointer

? (_GetResource :word #x5223      ;ASCII "R#"
                :word #x5354      ;ASCII "ST"
                :word 15          ;resource number 15
                :ptr)             ;return value is pointer
```

Note that in the second form, the components of the resource type are pushed in reverse order so that they will be in the correct order on the stack.

Low-level register trap calls

Here is the general format of a register trap call.

_TrapName [*Macro*]

Syntax *_TrapName* [:check-error]{ *register-keyword argument* }* [*return-register-keyword*]

Description The arguments are evaluated, coerced according to *register-keyword* , and passed as the arguments to the trap.

Arguments

_TrapName The name of the trap.

:check-error
A symbol. If this symbol appears as the first argument to a register trap, then Macintosh Common Lisp signals an error if the trap returns a negative value in register d0. Before using :check–error , make sure that the trap in question uses this error-signaling protocol.

register-keyword
A keyword that specifies the register that holds the subsequent *argument*.

argument An argument to be passed to the register. Arguments are evaluated in left-to-right order and placed in registers according to the *register-keywords* . Arguments to be placed in data registers should be 32-bit values. Arguments placed in address registers are not coerced in any way and should be macptrs.

return–register-keyword
A keyword that specifies which register will hold the value returned by the trap (if any). Recognized register keywords are :a0 through :a6 and :d0 through :d7. If *return-register-keyword* is not supplied, then nil is returned. Values returned from a data register are returned as 32-bit values. Values are returned from an address register as macptrs. There is no facility for returning multiple values from register traps.

Example

In this example, upon completion `my-pointer` holds a pointer to a 2000-byte block of memory on the current Macintosh heap. If the memory cannot be allocated, a Lisp error is signaled.

```
(setq my-pointer (_newptr :check-error
                          :d0 2000
                          :a0))
```

Lisp macros for calling Macintosh traps

The forms `stack-trap`, `register-trap`, and `%gen-trap` provide a generalized mechanism for calling Macintosh traps. They are used in the trap definition files.

`stack-trap` [*Macro*]

Syntax `stack-trap` *trap-number* {*trap-keyword argument* }* [*return-value-keyword*]

Description The `stack-trap` macro expands into an efficient low-level system call to the stack.

Arguments

trap-number A value that evaluates to a 68000 A-trap instruction. These can be found in *Inside Macintosh*. If *trap-number* is specified as a compile-time constant expression, the trap call can be open-coded by the compiler.

type-keyword A value that signifies the type of coercion to be performed on the corresponding *argument*. The keywords operate on the subsequent *argument* according to the following list.

`:word` A keyword that causes *argument* to be passed as a 16-bit word. Arguments passed as words should be fixnums; it is an error if *argument* has more than 16 significant bits.

`:long` A keyword that causes *argument* to be passed as a 32-bit longword.

`:ostype` A keyword that causes *argument*, a four-character string or a symbol with a four-character print name, to be passed as a 32-bit value (8 bits for each of the characters); ostypes are used as identifiers by the Resource Manager, Scrap Manager, and other parts of the Macintosh Operating System.

`:ptr` A keyword that causes *argument* to be passed as a macptr.

`:d0` A keyword that causes *argument* to be passed in the D0 register.

`:boolean` A keyword that causes *argument* to be passed as a Boolean value.

argument An argument to be passed to the stack. As noted previously, *argument* should evaluate to 32-bit values or to macptrs to data on the Macintosh heap or the stack.

`:trap-modifier-bits` *bitmask*
A sequence. Anywhere that a *type-keyword* /*argument* pair may appear, the sequence `:trap-modifier-bits` *bitmask* is also allowed. The `:trap-modifier-bits` specifier causes the associated bitmask to be logical-ored with the value of `_TrapName` and the resulting value used as the trap number. The placement of any `:trap-modifier-bits` forms in the argument list is not significant.

return-value-keyword
Indicates the type of value returned by the trap. If *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

`:word` A keyword that causes a16-bit result to be read from the stack, sign-extended, and returned as a signed Lisp integer.

`:long` A keyword that causes a 32-bit result to be read from the stack and returned as a signed Lisp integer.

`:ptr` A keyword that causes a 32-bit result to be returned from the stack as a macptr.

`:novalue` A keyword that causes no value to be returned from the Macintosh trap. The Lisp call returns `nil`.

`:boolean` A Boolean value is returned.

register-trap [*Macro*]

Syntax `register-trap [:check-error ]` *trap-number* `{`*register-keyword argument* `}* [`*return-register-keyword* `]`

Description The `register-trap` macro expands into an efficient low-level system call to the register.

Arguments

`:check-error`
Signals an error if the trap returns a negative value in register `d0`.

trap-number Evaluates to a 68000 A-trap instruction. These instructions can be found in *Inside Macintosh*. If *trap-number* is specified as a compile-time constant expression, the trap call can be open-coded by the compiler.

register-keyword
Specifies the register that holds the subsequent *argument*. Recognized register keywords are: `a0` through `:a6` and `:d0` through `:d7`.

argument An argument to a register call. Arguments are evaluated in left-to-right order and placed in registers. Arguments to be placed in data registers should be 32-bit values. Arguments placed in address registers should be macptrs.

`:trap-modifier-bits` *bitmask*
Anywhere that a *register-keyword* / *argument* pair may appear, the sequence: `trap-modifier-bits` *bitmask* is also allowed. The `:trap-modifier-bits` specifier causes the associated bitmask to be logical-ored with the value of `_TrapName` and the resulting value used as the trap number. The placement of any: `trap-modifier-bits` forms in the argument list is not significant.

return-register-keyword
Specifies which register holds the value returned by the trap (if any). Recognized register keywords are `:a0` through `:a6` and `:d0` through `:d7`.

Examples

Here is an example of the use of a register-based trap call with `:trap-modifier-bits`. The trap instruction `_NewPtr` has #x0200 logical-ored into it, causing the ROM to zero out the contents of the newly created pointer.

```
? (defconstant #$trap-clear-bitmask #x0200)
#$TRAP-CLEAR-BITMASK
? (_NewPtr :trap-modifier-bits #$trap-clear-bitmask
           :d0 10 :a0)
#<A Mac Zone Pointer Size 10 #x65F9A>
```

The following forms are equivalent to the previous form:

```
(register-trap _newptr :trap-modifier-bits 512 :d0 10 :a0)
(register-trap (logior _NewPtr 512) :d0 10 :a0))
```

Placement is not significant; this call is equivalent to any of the preceding ones.

```
(_NewPtr :d0 10 :trap-modifier-bits #$trap-clear-bitmask :a0)
```

%gen-trap

[*Function*]

Syntax `%gen-trap` *trap* {*type-keyword argument* }* [`:return-block` *pointer* | *return-value-keyword*]

Description The `%gen-trap` function executes a low-level system call to a trap with parameters on the stack or in registers.

It returns a value according to `:return-block` or *return-value-keyword* . If the `:return-block` keyword is present, `%gen-trap` returns `nil`. If it is not present, `%gen-trap` returns the value appropriate for *return-value-keyword* .

The compiler open-codes this function.

Arguments

trap A trap call to a trap word in the range `$A000` - `$AFFF`.

type-keyword A keyword that indicates what type of coercion to perform on the subsequent argument and where to place the argument. The possible values of *type-keyword* are the following *arguments*:

`:word` The *argument* parameter (which should be a fixnum) is truncated to 16 bits, and the resulting word value is pushed on the stack.

`:long` An object coercible to a 32-bit immediate quantity. The resulting longword value is pushed on the stack.

`:ptr` The *argument* parameter is pushed on the stack as a macptr.

`:d0-:d7` An object coercible to a 32-bit immediate quantity. The resulting longword value is placed in the indicated data register.

`:a0-:a4` The *argument* parameter should be a macptr. Its address is put in the indicated address register.

`:boolean` This keyword causes *argument* to be passed as a Boolean value.

The stack-based arguments are pushed on the stack in the order in which they appear in the call.

`:return-block`

If this keyword appears in a trap call, it should be followed by a pointer to a block of memory where the returned values are placed. This mechanism lets a trap return values from multiple registers and positions on the stack. If this keyword is present, `%gen-trap` returns `nil`.

pointer A pointer.

return-value-keyword

Indicates the type of value the function returns if `:return-block` is not present. If the value of *return-value-keyword* is not supplied, `:novalue` is assumed. The following keywords are recognized:

`:word`	A 16-bit result is read from the top of the stack, sign-extended, and returned as a signed Lisp integer.
`:long`	A 32-bit result is read from the top of the stack and returned as a signed Lisp integer.
`:ptr`	A 32-bit result is read from the top of the stack and returned as a macptr.
`:d0-:d7`	The value of the indicated data register is returned as a signed Lisp integer.
`:a0-:a4`	The value of the indicated address register is returned as a macptr.
`:boolean`	A Boolean value is returned.
`:novalue`	No value is returned and the trap call returns `nil`.

Notes on trap calls

The following sections discuss 32-bit immediate quantities and Boolean true and false values.

32-bit immediate quantities

When interfacing to the Macintosh ROMs, it is necessary to be able to specify 32-bit immediate quantities. These quantities are used to represent numerical values. When viewed as a sequence of 4 consecutive bytes, they are sometimes used to denote os-types.

The following Lisp data types may be mapped to 32-bit values in contexts where a trap or memory-access primitive requires a 32-bit immediate quantity:

- Any fixnum. Fixnums use only the low 29 bits.
- Any other integer. The 32 least significant bits of the integer are used. It is an error to pass an integer larger than 32 bits, but the error is not detected.
- A character of type `base-character`. The 32-bit value of the character code is used as the value.
- A string whose length is four characters. Macintosh Common Lisp views such a string as a sequence of 4 bytes; this allows specifying a 32-bit os-type.
- A symbol whose print name is a string whose length is four characters.

The constants `nil` and `t` are no longer acceptable as arguments to functions (such as `%put-long`) that require a 32-bit value.

Boolean values—Pascal true and false

Pascal parses Boolean values as words (2 bytes) with bit 8 set. Macintosh Common Lisp automatically converts between this representation and the MCL values `nil` and `t`. Thus,

```
(stack-trap _button :boolean)
```

is equivalent to

```
(logbitp 8 (stack-trap _button :word))
```

and

```
(stack-trap foo :boolean x)
```

is equivalent to

```
(stack-trap foo :word (if x -1 0))
```

Appendix A **Implementation Notes**

Contents

The Metaobject Protocol / 606
- Metaobject classes defined in Macintosh Common Lisp / 606
- Unsupported metaobject classes / 607
- Unsupported Introspective MOP functions / 608
- MCL functions relating to the Metaobject Protocol / 608

MCL class hierarchy / 621

Types and tag values / 621

Reader macros undefined in Common Lisp / 623

Numeric argument undefined in Common Lisp / 624

Numbers / 624

Characters / 626

Arrays / 626

Packages / 628

Additional printing variables / 629

Memory management / 630

Function swapping / 630

Garbage collection / 631
- Ephemeral garbage collection / 632
  - Generations / 632
  - Memory management unit support / 632
- Full garbage collection / 636

Evaluation / 637

Compilation / 638
- Tail recursion elimination / 638
- Self-referential calls / 638
- Compiler policy objects / 638

Miscellaneous MCL expressions / 644

This appendix describes details of the implementation of Common Lisp by Macintosh Common Lisp. It includes information on cases that are ambiguous in Common Lisp and provides technical information on memory management, the compiler, and other aspects of the Macintosh Common Lisp system.

The Metaobject Protocol

Macintosh Common Lisp version 2 implements CLOS as documented in the second edition of *Common Lisp: The Language*. It also contains some informational functions that are part of the Metaobject Protocol (MOP) as described in *The Art of the Metaobject Protocol* by Gregor Kiczales and others (MIT Press, 1991).

Metaobject classes defined in Macintosh Common Lisp

Table A-1 shows the class structure of the metaobject classes defined in Macintosh Common Lisp version 2. All the metaobject classes are instances of `standard-class` except `generic-function` and `standard-generic-function`, which are instances of `funcallable-standard-class`. They are not documented in *Common Lisp: The Language*, but some of them are documented in *The Art of the Metaobject Protocol*.

Table A-1 Structure of metaobject classes defined in Macintosh Common Lisp version 2

Class	Direct superclasses
`standard-object`	`t`
`structure-object`	`t`
`metaobject`	`standard-object`
`method-combination`	`metaobject`
`long-method-combination`	`method-combination`
`short-method-combination`	`method-combination`
`standard-method-combination`	`method-combination`
`method`	`metaobject`
`standard-method`	`method`
`standard-accessor-method`	`standard-method`
`standard-writer-method`	`standard-accessor-method`
`standard-reader-method`	`standard-accessor-method`

(continued)

n **Table A-1** Structure of metaobject classes defined in Macintosh Common Lisp version 2 (continued)

Class	Direct superclasses
`generic-function`	`metaobject` `ccl::funcallable-standard-object`
`standard-generic-function`	`generic-function`
`specializer`	`metaobject`
`class`	`specializer`
`ccl::compile-time-class`	`class`
`structure-class`	`class`
`built-in-class`	`class`
`ccl::std-class`	`class`
`funcallable-standard-class`	`std-class`
`standard-class`	`std-class`

During compilation, if a class definition is encountered for a previously unknown class, an instance of the class named `ccl::compile-time-class` is added to the compilation environment. This instance is a stub only. The Common Lisp generic function `class-name` returns its name and `find-class` finds it if given the compile-time environment as its third argument, but none of the other MOP functions returns any kind of useful information. For example, `class-precedence-list` signals an error when called with an instance of `ccl::compile-time-class`. This way of handling `defclass` at compile time is very likely to change in future versions of Macintosh Common Lisp.

The class named `ccl::std-class` is an implementation detail that may change in future versions of Macintosh Common Lisp; hence its name is not exported. It is included in the above table for completeness.

Unsupported metaobject classes

The following metaobject classes do not exist in Macintosh Common Lisp version 2.0:

```
eql-specializer
forward-referenced-class
slot-definition
```

```
standard-slot-definition
standard-direct-slot-definition
standard-effective-slot-definition
```

Unsupported Introspective MOP functions

The following functions, which are part of the de facto Introspective MOP standard, are not supported by Macintosh Common Lisp verson 2.0:

```
class-default-initargs
class-direct-default-initargs
generic-function-argument-precedence-order
generic-function-declarations
generic-function-initial-methods
generic-function-lambda-list
method-lambda-list
slot-boundp-using-class
slot-definition-class
slot-definition-allocation
slot-definition-initargs
slot-definition-initform
slot-definition-initfunction
slot-definition-name
slot-definition-readers
slot-definition-type
slot-definition-writers
slot-exists-p-using-class
slot-makunbound-using-class
slot-value-using-class
```

MCL functions relating to the Metaobject Protocol

The following MOP functions are supported in Macintosh Common Lisp.

class-direct-subclasses [*Generic function*]

Syntax `class-direct-subclasses` (*class* `class`)

Description The `class-direct-subclasses` generic function returns a list of the direct subclasses of the given class, that is, all classes that mention this class in their `defclass` forms.

Argument *class* A class.

Example

```
? (defclass foo () ())
#<STANDARD-CLASS FOO>
? (defclass bratch (foo) ())
#<STANDARD-CLASS BRATCH>
? (defclass gronk (foo) ())
#<STANDARD-CLASS GRONK>
? (class-direct-subclasses (find-class 'foo))
(#<STANDARD-CLASS GRONK> #<STANDARD-CLASS BRATCH>)
? ? (class-direct-subclasses (find-class 'standard-object))
(#<STANDARD-CLASS FOO>
 #<STANDARD-CLASS INSPECTOR::ERROR-FRAME>
 #<STANDARD-CLASS INSPECTOR::UNDO-VIEW-MIXIN>
 #<STANDARD-CLASS INSPECTOR::BOTTOM-LINE-MIXIN>
 #<STANDARD-CLASS INSPECTOR::CACHE-ENTRY>
 #<STANDARD-CLASS INSPECTOR::BASICS-FIRST-MIXIN>
 #<STANDARD-CLASS INSPECTOR::OBJECT-FIRST-MIXIN>
 #<STANDARD-CLASS INSPECTOR::UNBOUND-MARKER>
 #<STANDARD-CLASS INSPECTOR::INSPECTOR> #<STANDARD-CLASS MENUBAR>
 #<STANDARD-CLASS KEY-HANDLER-MIXIN> #<STANDARD-CLASS APPLICATION>
 #<STANDARD-CLASS CONDITION> #<STANDARD-CLASS SCRAP-HANDLER>
 #<STANDARD-CLASS CCL::LISP-WDEF-MIXIN>
 #<STANDARD-CLASS CCL::INSTANCE-INITIALIZE-MIXIN>
 #<STANDARD-CLASS FUNCALLABLE-STANDARD-OBJECT>
 #<STANDARD-CLASS METAOBJECT>)
```

The file `grapher.lisp` in your MCL Examples folder contains a good example of the use of`class-direct-subclasses` .

class-direct-superclasses [*Generic function*]

Syntax `class-direct-superclasses` (*class* `class`)

Description The `class-direct-superclasses` generic function returns a list of the direct superclasses of the given class, that is, all classes that are specified in the class's `defclass` form.

Argument *class* A class.

Example

```
? (defclass my-io-stream (input-stream output-stream) ())
#<STANDARD-CLASS MY-IO-STREAM>
? (class-direct-superclasses *)
(#<STANDARD-CLASS INPUT-STREAM> #<STANDARD-CLASS OUTPUT-STREAM>)
```

class-precedence-list [*Generic function*]

Syntax `class-precedence-list` (*class* `class`)
`class-precedence-list` (*class* `standard-class`)

Description The `class-precedence-list` generic function returns the class precedence list for the given class. This list is used by `compute-applicable-methods` to determine the order of precedence of methods specialized on the class.

Argument *class* A class.

Example

```
? (defclass foo () ())
#<STANDARD-CLASS FOO>
? (class-precedence-list *)
(#<STANDARD-CLASS FOO> #<STANDARD-CLASS STANDARD-OBJECT> #<BUILT-IN-
CLASS T>)
? (defclass bar () ())
#<STANDARD-CLASS BAR>
? (class-precedence-list *)
(#<STANDARD-CLASS BAR> #<STANDARD-CLASS STANDARD-OBJECT> #<BUILT-IN-
CLASS T>)
? (defclass gronk (foo bar) ())
#<STANDARD-CLASS GRONK>
? (class-precedence-list *)
(#<STANDARD-CLASS GRONK> #<STANDARD-CLASS FOO> #<STANDARD-CLASS BAR>
 #<STANDARD-CLASS STANDARD-OBJECT> #<BUILT-IN-CLASS T>)
```

class-prototype

[*Generic function*]

Syntax `class-prototype` (*class* `ccl::std-class`)
`class-prototype` (*class* `structure-class`)

Description The `class-prototype` generic function returns a prototype instance of the given class. The contents of the instance are undefined, though it has the same number of instance slots as an instance created with `make-instance` (or a structure creator function), and all class slots are accessible.

Argument *class* A class.

Example

In this example, `y` is bound only because of `:allocation :class` .

```
? (defclass foo ()
            ((x :initform 1 :accessor foo-x)
             (y :allocation :class
                :initform 2 :accessor foo-y)))
#<STANDARD-CLASS FOO>
? (foo-y (class-prototype (find-class 'foo)))
2
```

class-direct-instance-slots

[*Generic function*]

Syntax `class-direct-instance-slots` (*class* `ccl::std-class`)

Description The `class-direct-instance-slots` generic function returns a list of slot definition objects describing the instance slots that were declared in the class's `defclass` forms. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name` .

Argument *class* A class.

Example

See the example in the definition of `slot-definition-name` later in this appendix.

class-direct-class-slots [*Generic function*]

Syntax `class-direct-class-slots` (*class* `ccl::std-class`)

Description The `class-direct-class-slots` generic function returns a list of slot definition objects describing the class slots that were declared in the class's `defclass` forms. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name` .

Argument *class* A class.

class-instance-slots [*Generic function*]

Syntax `class-instance-slots` (*class* `ccl::std-class`)

Description The `class-instance-slots` generic function returns a list of slot definition objects describing all the instance slots, direct and inherited, that were declared in the `defclass` for the class. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name` .

Argument *class* A class.

class-class-slots [*Generic function*]

Syntax `class-class-slots` (*class* `ccl::std-class`)

Description The `class-class-slots` generic function returns a list of slot definition objects describing all the class slots, direct and inherited, that were declared in the `defclass` for the class. MCL slot definitions are represented as lists. The only supported accessor for a slot definition object is `slot-definition-name` .

Argument *class* A class.

Example

```
? (defclass foo ()
    ((x :accessor foo-x
        :initarg :x
        :initform 1)
     (y :allocation :class
        :accessor foo-y
        :initarg :y
        :initform 2)))
#<STANDARD-CLASS FOO>
? (defclass bar (foo)
    ((m :accessor bar-m
        :initarg :m
        :initform 3)
     (n :allocation :class
        :accessor bar-n
        :initarg :n
        :initform (log 4))))
#<STANDARD-CLASS BAR>
? (class-direct-instance-slots (find-class 'bar))
((M (3) (:M)))
? (class-direct-class-slots (find-class 'bar))
((N #<Anonymous Function #xDF2EA6> (:N)))
? (class-instance-slots (find-class 'bar))
((M (3) (:M)) (X (1) (:X)))
? (class-class-slots (find-class 'bar))
((N (1.3862943611198906) (:N)) (Y (2) (:Y)))
```

---

specializer-direct-methods

[*Generic function*]

Syntax

`specializer-direct-methods` (*specializer* `class`)
`specializer-direct-methods` (*specializer* `list`)

Description

The `specializer-direct-methods` generic function returns a list of all methods that specialize on the given specializer. An `eql` specializer is represented as a list of length 2 whose `car` is the symbol `eql` and whose `cadr` is an MCL object.

In the default world, the `specializer-direct-methods` lists are not cached. The first time you call `specializer-direct-methods` or `specializer-direct-generic-functions`, it maps over all the generic functions, computing the direct methods lists for all specializers. It also enables caching of this information for subsequent calls to `add-method` and `remove-method`. This caching is preserved across calls to `save-application`. The function `clear-class-direct-methods-caches` clears all the cached information and stops `add-method` from keeping track of it until the next call to `specializer-direct-methods` or `specializer-direct-generic-functions`.

Argument *specializer* A class or a list of the form `(eql` *object* `)`.

`specializer-direct-generic-functions` [*Generic function*]

Syntax

`specializer-direct-generic-functions` (*specializer* `class`)
`specializer-direct-generic-functions` (*specializer* `list`)

Description The `specializer-direct-generic-functions` generic function returns a list of all generic functions that specialize on the given specializer. An `eql` specializer is represented as a list of length 2 whose `car` is the symbol `eql` and whose `cadr` is an MCL object.

In the default world, the `specializer-direct-generic-functions` lists are not cached. The first time you call `specializer-direct-methods` or `specializer-direct-generic-functions`, it maps over all the generic functions, computing the direct methods lists for all specializers. It also enables caching of this information for subsequent calls to `add-method` and `remove-method`. This caching is preserved across calls to `save-application`. The function `clear-class-direct-methods-caches` clears all the cached information and stops `add-method` from keeping track of it until the next call to `specializer-direct-methods` or `specializer-direct-generic-functions`.

Argument *specializer* A class or a list of the form `(eql` *object* `)`.

generic-function-methods [*Generic function*]

Syntax `generic-function-methods` (*generic-function* `standard-generic-function`)

Description The `generic-function-methods` generic function returns a list of the methods for *generic-function* .

Argument *generic-function* A generic function.

Example

```
? (defmethod foo ((x integer)) x)
#<STANDARD-METHOD FOO (INTEGER)>
? (defmethod foo ((x fixnum)) (+ x (call-next-method)))
#<STANDARD-METHOD FOO (FIXNUM)>
? (generic-function-methods #'foo)
(#<STANDARD-METHOD FOO (FIXNUM)> #<STANDARD-METHOD FOO (INTEGER)>)
```

method-function [*Generic function*]

Syntax `method-function` (*method* `standard-method`)

Description The `method-function` generic function returns the function that runs when *method* is invoked. This function takes the same number of arguments as the generic function. If it was generated from code containing a call to `call-next-method` , function-calling it with `funcall` will cause Macintosh Common Lisp to crash. (Otherwise it can be function-called safely.)

Argument *method* A standard method.

method-generic-function [*Generic function*]

Syntax `method-generic-function` (*method* `standard-method`)

Description The `method-generic-function` generic function returns the generic function associated with *method* , or `nil` if there is none.

Argument *method* A standard method.

Example

```
? (setq m (defmethod foo ((x integer)) x))
#<STANDARD-METHOD FOO (INTEGER)>
? (method-generic-function m)
#<STANDARD-GENERIC-FUNCTION FOO #xD61B66>
? (remove-method (method-generic-function m) m)
#<STANDARD-GENERIC-FUNCTION FOO #xD61B66>
? (method-generic-function m)
NIL
```

method-name

[*Generic function*]

Syntax method-name (*method* standard-method)

Description The method-name generic function returns the name of *method*

Argument *method* A standard method.

Example

```
? (defmethod foo ((x integer)) x)
#<STANDARD-METHOD FOO (INTEGER)>
? (method-name *)
FOO
```

method-qualifiers

[*Generic function*]

Syntax method-qualifiers (*method* standard-method)

Description The method-qualifiers generic function returns a list of the qualifiers for *method* . (See *Common Lisp: The Language,* pages 839, 849.)

Argument *method* A standard method.

method-specializers

[*Generic function*]

Syntax `method-specializers` (*method* `standard-method`)

Description The `method-specializers` generic function returns a list of the specializers for *method* .

Argument *method* A standard method.

Example

```
? (defmethod bar ((x integer) (y list)) (cons x y))
#<STANDARD-METHOD BAR (INTEGER LIST)>
? (method-specializers *)
(#<BUILT-IN-CLASS INTEGER> #<BUILT-IN-CLASS LIST>)
```

slot-definition-name

[*Generic function*]

Syntax `slot-definition-name` (*slot-definition* `list`)

Description The `slot-definition-name` generic function returns the name of *slot-definition* . Future versions of Macintosh Common Lisp will fully support the`slot-definition` class.

Argument *slot-definition* A slot definition object.

Example

```
? (defclass foo () (x y))
#<STANDARD-CLASS FOO>
? (mapcar 'slot-definition-name
     (class-direct-instance-slots *))
(X Y)
```

copy-instance

[*Generic function*]

Syntax `copy-instance` (*instance* `standard-object`)

Description The `copy-instance` generic function returns a copy of the given instance. The default method merely copies the vector used to store the instance slots for the instance. Users may add methods to perform additional initialization for the copied method.

Argument	*instance*	An instance of `standard-object` or one of its subclasses.

Example

There are examples of `copy-instance` in the Interface Toolkit source code.

`clear-specializer-direct-methods-caches` [*Function*]

Syntax `clear-specializer-direct-methods-caches`

Description The `clear-specializer-direct-methods-caches` function clears all the cached information returned by `specializer-direct-methods` and `specializer-direct-generic-functions`, preventing subsequent calls to `add-method` from caching this information. The next call to either of these functions recomputes the caches and reenables maintenance of them by `add-method`.

`clear-clos-caches` [*Function*]

Syntax `clear-clos-caches`

Description The `clear-clos-caches` function clears CLOS caches in preparation for doing a `save-application` if the value of the `:clear-clos-caches` keyword argument to `save-application` is true (the default). (See Appendix B, "Snapshots and Application Versions of Macintosh Common Lisp.") This function clears the effective method caches stored inside generic functions and the valid initialization argument caches stored inside classes.

`clear-gf-cache` [*Function*]

Syntax `clear-gf-cache` *generic-function*

Description The `clear-gf-cache` function clears the cached effective methods for *generic-function*. This function saves space but causes subsequent invocations of the generic function to be slower until the cache is filled again.

Argument *generic-function*
A generic function.

clear-all-gf-caches [*Function*]

Syntax `clear-all-gf-caches`

Description The `clear-all-gf-caches` function calls `clear-gf-caches` on all generic functions. This function is called by `clear-clos-caches`.

method-exists-p [*Function*]

Syntax `method-exists-p` *generic-function* `&rest` *args*

Description The `method-exists-p` function returns `nil` if *generic-function* is not a generic function or a symbol naming a generic function, or if (`apply` *generic-function args*) would cause an error because there are no applicable primary methods for the given arguments to the generic function. Otherwise, it returns one of the applicable primary methods for *generic-function* . This function is faster than `compute-applicable-methods` and does not `cons`.

Arguments *generic-function*
A generic function or a symbol naming a generic function.

args One or more arguments to the generic function.

check-call-next-method-with-args [*Variable*]

Description The `*check-call-next-method-with-args*` variable determines whether a run-time check is made during calls to `call-next-method` .

When the value of this variable is true (the default), then the check is made to ensure that new arguments do not change the set of methods that are applicable for the generic function.

When the value of this variable is `nil`, *then* no check is made.

The checking is not completely strict. If the required arguments that are passed to `call-next-method` are `eq` to the original required arguments passed to the generic function, then the test passes.

For effective methods that have already been cached, changes to `*check-call-next-method-with-args*` will not take effect until `clear-all-gf-caches` is invoked.

---

defmethod-congruency-override [*Variable*]

Description The `*defmethod-congruency-override*` variable allows you to override standard MCL behavior when you define global generic functions.

When the value of this variable is `nil` *(the default), then* an error is signaled.

When the value of this variable is true, then Macintosh Common Lisp does not signal an error if the function binding of the generic function's name is not a generic function or if a method's lambda list is not congruent to its generic function's lambda list.

If `*defmethod-congruency-override*` is a function, then it is called with two arguments as described next.

If an attempt is made by `defmethod` or `ensure-generic-function` to redefine a regular function, macro, or special form, `*defmethod-congruency-override*` is called with two arguments, the function name (a symbol) and `nil`. If `nil` is returned, an error is signaled. Otherwise, the redefinition is performed.

If `add-method` is instructed to add a method to a generic function and the lambda lists of the method and the generic function are not congruent, `*defmethod-congruency-override*` is called with two arguments, the generic function and the method. If it returns `nil`, an error is signaled. Otherwise, all methods are removed from the generic function, the generic function's lambda list is redefined to be congruent with the method's lambda list, and the method is added.

If `*defmethod-congruency-override*` is not `nil` and not a function, it behaves as if it were a function that always returns non-`nil`. Hence, redefinitions are performed silently. This is very dangerous and should usually be done only by patch files.

MCL class hierarchy

The file `print-class-tree.lisp` in the MCL Examples folder contains functions to print the class hierarchy of an MCL class in a way that makes the direct superclasses and the class precedence list apparent. It includes, as a comment, a hierarchy diagram for every class in the MCL system, sorted by class name.

Types and tag values

In Macintosh Common Lisp, references to Lisp objects are encoded in 32-bit 680x0 longwords. The 3 least significant bits of the longword are referred to as the object's **tag** and determine the type of the object (see the list of tag values that follows). In the case of **immediate objects** such as fixnums, characters, and short floats, the value of the object is contained in the remaining 29 bits. In other cases, the 32-bit longword constitutes a **tagged pointer** to the associated object.

A consequence of this tagging scheme is that all nonimmediate Lisp objects are allocated on 8-byte boundaries.

- The tag value of 0 is used to represent fixnums; the two's-complement value of the fixnum is stored in the upper 29 bits of the longword. Fixnums can therefore store values in the range `-2^28` through `(2^28)-1`, inclusive. Note that this representation allows the direct use of machine arithmetic instructions where applicable.
- The tag value of 1 is used to represent variable-length objects called **uvectors.** Objects with this tag include all arrays, CLOS instances, structure instances, bignums, ratios, complex numbers, `macptr` pointers, packages, and a few more internal types. A pointer that contains this tag points 1 byte beyond the beginning of the storage occupied by the object it points to.

- The tag value of 2 represents symbols; symbol pointers therefore point 2 bytes into the storage allocated to the symbol.
- The tag value of 3 represents double-precision floating-point values; such pointers point 3 bytes into the 64-bit double-float.
- The tag value of 4 represents `cons` cells and `nil`. Since the `car` of a `cons` cell occupies the first of two longwords allocated to that cell, **cons-tagged** objects point at the `cdr` of the `cons` cell.
- The tag value of 5 is used to represent instances of the **short-float** data type; the upper 29 bits of the longword encode a sign bit, a 5-bit exponent, and a 23-bit significand.
- The tag value of 6 is used to denote functions; all valid Lisp objects with this tag point to executable machine code.
- The tag value of 7 is used to represent small immediate objects, including characters. The least significant byte of a character contains the value `#xF`.
  - If bits 8 to 15 of the character contain `#xF`, then the character is a **base character** (a Lisp object of type `base-character`). The character code of the character is contained in the most significant word of the object; if the character is a base character, then this value must be in the range 0 through 255 inclusive.

    The Lisp character type `extended-character` is not implemented in this release of Macintosh Common Lisp. For now at least, all Lisp objects of type `character` are of type `base-character`.
  - Other immediate objects whose tag is 7 and whose low byte is *not* `#xF` are used to represent various constants used by the Memory Manager and the run-time system.

The mapping between tags and Common Lisp types is an implementation detail that is likely to change in future version of Macintosh Common Lisp.

The following functions provide low-level access to objects that are tagged as uvectors.

`uvectorp` [*Function*]

Syntax `uvectorp` *object*

Description The `uvectorp` function returns true if *object* is tagged as a uvector.

Argument *object* A variable-length `uvector` object.

uvsize [*Function*]

Syntax `uvsize` *object*

Description The `uvsize` function returns the size of *object* as a fixnum.

Argument *object* A variable-length `uvector` object.

uvref [*Function*]

Syntax `uvref` *object index*

Description The `uvref` function returns the element of *object* at *index*. The function signals an error unless `(<= 1` *index* `(uvsize` *object* `))`.

The function `setf` may be used with `uvref` to modify an element of a uvector. If *object* is a simple array, `uvref` is the same as `aref`.

Arguments *object* A variable-length `uvector` object.

index An index into *object* .

Reader macros undefined in Common Lisp

In addition to supporting the standard Common Lisp reader macro characters, Macintosh Common Lisp defines the following dispatching reader macros, which are undefined in Common Lisp:

#@	Transforms the subsequent list of two fixnums into a point.
#$	Should be followed by a symbol *symbol* . Interns $*symbol* in the `traps` package and, if `*autoload-traps*` is true, attempts to load an interface constant definition. See Chapter 17, “Higher-Level Operating System Interface.”
#_	Should be followed by a symbol *symbol* . Interns _*symbol* in the `traps` package and, if `*autoload-traps*` is true, attempts to load a trap definition. See Chapter 17, “Higher-Level Operating System Interface.”

Numeric argument undefined in Common Lisp

Macintosh Common Lisp uses the CLOS #P syntax for pathnames, but it also has a numeric argument that specifies one of four possible unusual conditions in the pathname. These numeric arguments are an error in Common Lisp and should not be used in portable code:

`#1P`	Means that the type of a pathname is `:unspecific`.
`#2P`	Means that the name of a pathname is `""`.
`#3P`	Means that the type of a pathname is `:unspecific` and its name is `""`.
`#4P`	Means that the namestring represents a logical pathname.

Numbers

Fixnums are stored as immediate data using a two's-complement representation. They are 29 bits long (see discussion of the tagging scheme, in the section "Types and Tag Values" earlier in this appendix). Note that `eql` fixnums are `eq` (although portable code should not rely on this fact).

Two internal floating-point data formats are supported. The format used to represent instances of the Common Lisp data types `single-float`, `double-float`, and `long-float` corresponds to IEEE double or 64-bit floating-point format. Each such floating-point number is 64 bits (8 bytes) long and consists of a sign bit, an 11-bit binary exponent (allowing exponents in the range –1022 through 1023), and a 52-bit significand. The significand precision is 53 bits.

The format used to represent instances of the Common Lisp data type `short-float` consists of 3 tag bits, a sign bit, a 5-bit binary exponent, and a 23-bit significand (for tags, see discussion of the tagging scheme, in the section "Types and Tag Values" earlier in this appendix). This format is similar to the IEEE single format, but the smaller exponent restricts the range of representable numbers (for example, the values of `least-positive-short-float` and `least-negative-short-float`) and does not allow the representation of denormalized numbers.

On Macintosh computers that do not have floating-point hardware, Macintosh Common Lisp emulates that portion of the floating-point instruction set that it uses.

The following functions extend Common Lisp numeric manipulation.

bignump

[*Function*]

Syntax `bignump` *number*

Description The `bignump` function returns a Boolean indicating whether *number* is a bignum.

Argument *number* A number.

fixnump

[*Function*]

Syntax `fixnump` *number*

Description The `fixnump` function returns a Boolean value `t` if *number* is a fixnum, `nil` if it is not.

Argument *number* A number.

lsh

[*Function*]

Syntax `lsh` *fixnum count*

Description The `lsh` function logically shifts *fixnum* by *count* and returns the result of the operation, which must also be a fixnum. This is the same as the Common Lisp `ash` function, except that any bits shifted out of the (currently) 29 bits of a fixnum are lost.

Arguments *fixnum* A fixnum.

count An integer.

Characters

Characters are represented as immediate data. The character code is the 8-bit extended ASCII value. In this release there is no support for the `extended-character` type: all characters are of type `base-character` and all strings are of type `base-string`.

The reader macros `#\^@` through `#\^_` can be used to read the characters with ASCII values 0 through 31. More generally, `#\^`*c*, for any character *c*, is equivalent to `(code-char (logxor 64 (char-code #\`*c*`)))`. In addition, `#\`*nnn*, where *nnn* is two or three octal digits, is equivalent to `(code-char #o`*nnn*`)`.

The variable `ccl::*name-char-alist*` contains names of special characters. It is used by the `char-name` and `name-char` functions.

Arrays

Table A-2 lists the distinct types of array element that are supported.

Table A-2 Types of array element

Type	Length (bits per element)
`bit`	1
`character`	8
`double-float`	64
`(unsigned-byte 8)`	8
`(signed-byte 8)`	8
`(unsigned-byte 16)`	16
`(signed-byte 16)`	16
`(unsigned-byte 32)`	32
`(signed-byte 32)`	32
`t`	One node (32 bits per element)

Only simple vectors are supported directly. All arrays of rank other than 1 are implemented as displaced arrays. In addition to the memory needed to store its elements, a simple vector requires 8 bytes of overhead; a bit vector requires 9 bytes. A complex (displaced) array has about `2+ (4* rank )` bytes of overhead. The rank of an array must be less than `#x2000` (8K).

No array may have more elements than the number equal to `most-positive-fixnum` (that is, `2*28-1`); therefore, only fixnums are valid array indices.

Table A-3 gives the theoretical limits on the sizes of arrays.

Table A-3 Theoretical limits on array length

Type	Length limit
`bit`	`most-positive-fixnum`
`character`	`#xFFFFF8`
`double-float`	`#x1FFFFF`
`(unsigned-byte 8)`	`#xFFFFF8`
`(signed-byte 8)`	`#xFFFFF8`
`(unsigned-byte 16)`	`#x7FFFFC`
`(signed-byte 16)`	`#x7FFFFC`
`(unsigned-byte 32)`	`#x3FFFFE`
`(signed-byte 32)`	`#x3FFFFE`
`t`	`#x3FFFFE`

All these limits represent arrays requiring approximately 16 MB of contiguous memory.

There is no limit on the size of individual dimensions of an array except the limits imposed by the total array size.

Multidimensional arrays and arrays that were created with non-`nil` values for the `:displaced-to` and/or the `:fill-pointer` arguments to `make-array` are stored as two vectors, a header and a storage vector.

displaced-array-p [*Function*]

Syntax `displaced-array-p` *array*

Description The `displaced-array-p` function returns `nil` if *array* is not a displaced array. If it is a multidimensional array or an array created with non-`nil` values for the `:displaced-to` and/or the `:fill-pointer` arguments to `make-array`, the function returns two values, the storage vector and the offset from the beginning of the storage vector to the beginning of the storage for the array.

Argument *array* An array.

Example

```
? (setq a (make-array 10))
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
? (setq b (make-array 5
                      :displaced-to a
                      :displaced-index-offset 3))
#(NIL NIL NIL NIL NIL)
? (displaced-array-p b)
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
3
? (eq * a)
T
```

Packages

Macintosh Common Lisp, following the forthcoming ANSI Common Lisp standard, uses the package name `common-lisp` instead of `lisp`. The only external symbols of the `COMMON-LISP` package are the approximately 900 symbols of Common Lisp.

The `CCL` package uses the `COMMON-LISP` package. Its exported symbols consist of extensions to Common Lisp provided by Macintosh Common Lisp. The `CCL` package shadows none of the Common Lisp symbols.

The `COMMON-LISP-USER` package uses both the `CCL` and the `COMMON-LISP` packages.

The default value of the `:use` argument to `make-package` and to `defpackage` is the value of the variable `*make-package-use-defaults*`. The initial value of this variable is `("COMMON-LISP" "CCL")`. (See *Common Lisp: The Language,* page 263.)

Macintosh Common Lisp includes a `lisp` package that behaves similarly to the one described in the first edition of *Common Lisp: The Language.* However, full compatibility is not guaranteed.

The variable `*autoload-lisp-package*` determines whether the `LISP` package is loaded when it is first referenced. The value of `*autoload-lisp-package*` is `nil`. If you are running your own code that depends on the `LISP` package, or using code such as PCL or Richard Waters's pretty printer (see *Common Lisp: The Language,* Chapter 27), you may need to do one or more of the following:

- Set the value of `*autoload-lisp-package*` to `t`. You can use the Environment dialog on the Tools menu. When the value of this variable is true, the `:lisp` package is automatically loaded when it is required.
- Load the file `lisp-package.lisp` or `lisp-package.fasl` from the Library folder. This source file defines the `lisp` package.

If you are running your own code, convert it if possible.

When you run code that depends on the `lisp` package and it is not loaded, a restart provides the opportunity to load it.

Additional printing variables

In addition to the standard Common Lisp printer variables (see *Common Lisp: The Language),* Macintosh Common Lisp uses the variables in Table A-4 to control printing.

Table A-4 Additional printing variables

Variable	Purpose
`*print-simple-vector*`	Determines how simple vectors are printed. *Default is* `nil`; prints simple vectors according to the value of `*print-array*`. If true, prints simple vectors readably. If an integer, prints simple vectors with a length less than the integer readably.
`*print-simple-bit-vector*`	Determines how simple bit vectors are printed. *Default is* `nil`; prints simple bit vectors according to the value of `*print-array*`. If true, prints simple bit vectors readably. If an integer, prints simple bit vectors with a length less than the integer readably.
`*print-string-length*`	Determines how strings are printed. *Default is* `nil`; all strings are printed in full. If an integer, strings with a length greater than the integer are printed using an abbreviated format.
`*print-structure*`	Determines whether structures are printed readably using `#S` syntax. *Default is* `t`; structures are printed in an abbreviated format. If true, structures are printed readably.

Memory management

Macintosh Common Lisp divides the application heap into two areas: a Lisp heap and a Macintosh heap. Most Lisp data structures (such as `cons` cells, symbols, arrays, and functions) are stored in the Lisp heap; most Macintosh data structures (such as Window records, bitmaps, and CODE resources) are stored in the Macintosh heap.

Function swapping

Most of the compiled functions in the MCL kernel participate in the **load-on-call** memory management system; that is, these functions are not loaded into memory until they are first called. When garbage collection occurs, all functions that are not active (that is, not awaiting return values on the stack) are purged. If called again, they have to be reloaded. Depending on disk speed, there may be a slight pause when functions are loaded into memory.

The load-on-call system sometimes sacrifices speed for memory conservation. If system memory is sufficient, users who wish to optimize speed may use the functions `purge-functions` and `preload-all-functions` to affect the swapping mechanism.

Executing `(progn (preload-all-functions) (purge-functions nil))` leaves about 500 KB free in a 3 MB partition. If you find you are running out of memory, you may need to use load-on-call memory management or create a larger partition.

Functions written by the user do not participate in the load-on-call system.

The following functions govern the behavior of the load-on-call memory management system.

`purge-functions` [*Function*]

Syntax `purge-functions &optional` *boolean*

Description The `purge-functions` function globally enables or disables purging of swappable functions.

The `purge-functions` function returns `t` if purging is enabled; otherwise, it returns `nil`. It allows the user to disable function purging before executing programs in which purging and reloading are undesirable.

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.

Argument *boolean* If supplied, this parameter turns function swapping on or off. Any non-`nil` value enables function swapping, whereas `nil` disables function swapping.

`preload-all-functions` [*Function*]

Syntax `preload-all-functions`

Description The `preload-all-functions` function loads all swappable functions into memory. Functions may still be purged during garbage collection, depending on the setting of `purge-functions` .

The `preload-all-functions` function is primarily intended for use in conjunction with `purge-functions` .

Examples

The following code may be placed in an `init` file to remove swapping completely:

```
(progn
  (preload-all-functions)
  (purge-functions NIL))
```

This is recommended for configurations with 3 MB or more of RAM.

Here is an example of checking for a safety margin.

```
(when (> (ccl::%freebytes) 2000000)
 (preload-all-functions)
 (purge-functions nil))
```

Garbage collection

Macintosh Common Lisp performs two kinds of garbage collection, ephemeral and full. They are described in this section.

Ephemeral garbage collection

Version 2 of Macintosh Common Lisp contains a new **ephemeral garbage collector,** which attempts to exploit the fact that, in general, most heap-allocated objects become inaccessible soon after they are created. By concentrating its efforts on reclaiming space in that area of the Lisp heap where new objects are created, the ephemeral garbage collector (EGC) can help reduce both the frequency and the duration of full garbage collections.

Generations

This garbage collector is "lifetime based." In cooperation with the storage allocation primitives, the ephemeral garbage collector partitions the population of all dynamically allocated objects into sets, called generations, whose members are created at roughly the same time relative to the members of some other generation.

- Newly created objects are elements of the youngest generation (generation 0).
- When the total size of all objects in generation 0 exceeds a specified threshold, ephemeral garbage collection is invoked. Those objects that were members of the youngest generation and survive the ephemeral garbage collection are "promoted" into the next older generation, generation 1.
- When generation 1 fills up, its partition is garbage collected and surviving objects are promoted into the oldest generation, generation 2.

  The current implementation maintains up to three ephemeral generations and one nonephemeral generation.
- If and when enough objects have been promoted into the oldest generation to cause that generation to fill up, a full garbage collection is performed.

The function `gc-thermometer`, defined in `thermometer.lisp` in your Examples folder, provides a graphic display of the EGC's behavior.

Memory management unit support

Although it can be enabled in any environment in which Macintosh Common Lisp runs, the ephemeral garbage collector is most effective if it can use underlying memory management hardware to detect cases where older objects may have been destructively modified so as to refer to members of younger generations. When memory management unit (MMU) hardware can be used, the ephemeral garbage collector can ignore those memory pages that have not been modified recently.

If this support is unavailable, the ephemeral garbage collector scans all older generations to find the occasional case where such an assignment has taken place. This overhead can be very significant in virtual memory environments; whether or not it is acceptable in real memory environments depends on the speed of the processor, the size limits associated with the ephemeral generations, and the behavior and needs of the application.

Tip Even without MMU support, ephemeral garbage collection is generally much faster than full garbage collection, although collection occurs more often. Ephemeral garbage collection with an MMU is much faster than without one.

The MCL kernel can provide MMU support to the ephemeral garbage collector in the following circumstances:

- If a memory management unit is present. Currently, this means either a 68030 or 68040 processor, or a 68020 processor with a 68851 coprocessor.
- When the operating system either provides system calls to test and reset the modified status of a memory page, or does not interfere with attempts to obtain such information directly from the MMU hardware. Currently, this means that MMU support is not available under A/UX.
- When the MMU is configured to use a reasonably small page size (8K or 4K). Currently, this means either a 68040 processor, the use of the System 7 virtual memory, or the use of the `ptable` system extension provided as part of the MCL distribution.

The ephemeral garbage collector is said to be enabled when Macintosh Common Lisp has been asked to use it; it is said to be active when Macintosh Common Lisp is in fact using it. (It may be enabled but inactive when, for instance, free space in the heap is less than the size limit of the youngest generation.)

The following functions can be used to control and configure the ephemeral garbage collector.

---

egc

[*Function*]

Syntax `egc` *enable-p*

Description The `egc` function attempts to enable and activate the ephemeral garbage collector if the value of *enable-p* is true. (See the discussion of enabling and activation preceding this definition.) If the value of *enable-p* is `nil`, `egc` disables the ephemeral garbage collector. The function returns `t` if the ephemeral garbage collector is enabled after performing the operation, `nil` otherwise.

Argument *enable-p* A Boolean value.

egc-enabled-p [*Function*]

Syntax `egc-enabled-p`

Description The `egc-enabled-p` function returns `t` if the ephemeral garbage collector is enabled and returns `nil` otherwise.

egc-active-p [*Function*]

Syntax `egc-active-p`

Description The `egc-active-p` function returns `t` if the ephemeral garbage collector is active and returns `nil` otherwise.

configure-egc [*Function*]

Syntax `configure-egc` *generation-0-size generation-1-size generation-2-size*

Description If the ephemeral garbage collector is not currently enabled, the `configure-egc` function sets the size limits of the ephemeral generations as indicated and `t` is returned. If the ephemeral garbage collector is enabled, the current values of the size limits are not affected and `nil` is returned.

The arguments should be nonnegative integers that specify the size limits in kilobytes that the ephemeral garbage collector should use for the three ephemeral generations.

Arguments

generation-0-size A positive integer.

generation-1-size A nonnegative integer.

generation-2-size A nonnegative integer.

egc-configuration [*Function*]

Syntax `egc-configuration`

Description The `egc-configuration` function returns three integer values that express the size limits in kilobytes associated with ephemeral generations 0, 1, and 2.

gctime

[*Function*]

Syntax `gctime`

Description The `gctime` function returns five integer values:

- the total number of milliseconds spent in all full and ephemeral garbage collections in the current session
- the total number of milliseconds spent in all full garbage collections in the current session
- if the ephemeral garbage collector is enabled, the total number of milliseconds spent in all ephemeral collections of generation 2 in the current session. If the EGC is not enabled, this value is 0.
- if the ephemeral garbage collector is enabled, the total number of milliseconds spent in all ephemeral collections of generation 1 in the current session. If the EGC is not enabled, this value is 0.
- if the ephemeral garbage collector is enabled, the total number of milliseconds spent in all ephemeral collections of generation 0 in the current session. If the EGC is not enabled, this value is 0.

gccounts

[*Function*]

Syntax `gccounts`

Description The `gccounts` function returns five integer values:

- the total number of full and ephemeral garbage collector invocations in the current session
- the total number of full garbage collector invocations in the current session
- if the ephemeral garbage collector is enabled, the total number of times the ephemeral garbage collector has been invoked on generation 2 in the current session. If the EGC is not enabled, this value is 0.

- if the ephemeral garbage collector is enabled, the total number of times the ephemeral garbage collector has been invoked on generation 1 in the current session. If the EGC is not enabled, this value is 0.
- if the ephemeral garbage collector is enabled, the total number of times the ephemeral garbage collector has been invoked on generation 0 in the current session. If the EGC is not enabled, this value is 0.

Full garbage collection

Macintosh Common Lisp uses a mark/compact/forward garbage collector. Garbage collection occurs automatically as memory is needed. This can happen in response to a Macintosh Operating System call or to a memory request by Macintosh Common Lisp. You can invoke garbage collection manually through the function `gc`.

When running under MultiFinder in System 6, the garbage collector optionally performs a limited amount of event processing, sufficient to partially handle `suspend` and `resume` events and to allow background tasks to run. The garbage collector's event handling does not handle window update events. It simply draws a gray pattern into regions it is expected to update and notifies Lisp's low-level event dispatcher that windows need to be updated. It also does not handle the conversion of the Clipboard on MultiFinder context switches.

gc-event-check-enabled-p [*Function*]

Syntax `gc-event-check-enabled-p`

Description The `gc-event-check-enabled-p` function returns a Boolean value, indicating whether Macintosh Common Lisp performs event processing during garbage collection. A value of `t`, the default, means that event processing is turned on during garbage collection.

set-gc-event-check-enabled-p [*Function*]

Syntax `set-gc-event-check-enabled-p` *boolean*

Description The `set-gc-event-check-enabled-p` function turns garbage collector event processing on or off according to the value of *boolean*.

Argument *boolean* A flag. If the value of *boolean* is true, Macintosh Common Lisp performs event processing during garbage collection.

Evaluation

Macintosh Common Lisp offers two evaluator options: a standard evaluator and a compiling evaluator.

- The standard evaluator conforms to Common Lisp standards as described in the second edition of *Common Lisp: The Language*, Chapter 20. However, `evalhook` and `applyhook` were removed from the Common Lisp standard by vote of the X3J13 committee in November 1989 (after *Common Lisp: The Language* went to press). Macintosh Common Lisp still supports them, but they are deprecated.
- The compiling evaluator compiles nontrivial expressions and then runs them. For looping or self-recursive constructs, the compiling evaluator is much faster (up to several hundred times). The compiling evaluator is used when the variable `*compile-definitions*` is non-`nil`.

In the default environment, the system uses the compiling evaluator; that is, the value of `*compile-definitions*` is `t`.

The following variable governs the behavior of the evaluator.

`*compile-definitions*` [*Variable*]

Description The `*compile-definitions*` variable determines whether MCL expressions are compiled. (See the introductory remarks in this section.)

If the value of this variable is true (the default), then all function definitions and most top-level forms are compiled.

If the value of this variable is `nil`, *then* no compilation is performed.

The value of this variable can be toggled in the Environment dialog box on the Tools menu.

Compilation

This section describes some of the behavior of the MCL compiler and describes some means of influencing that behavior.

Tail recursion elimination

The MCL compiler attempts to minimize the stack usage of compiled functions by being properly **tail recursive.** A function is tail recursive if it returns the value(s) of the last function it calls as its own. In that case, the stack space allocated for the function's returned value(s) can be deallocated before it begins execution.

One side effect of the elimination of tail recursion is that, in general, the Stack Backtrace tools display only a portion of the execution history, since those function calls in which tail recursion was eliminated are no longer awaiting return values.

The compiler can be advised that tail recursion should never be eliminated from calls to certain single-valued global functions. Do this by adding the names of those functions to the list that is the value of the variable `ccl::*nx-never-tail-call*`. You can also use customized compiler policy objects to control when the compiler eliminates tail recursion. (See the section "Compiler Policy Objects" later in this appendix.)

Self-referential calls

Within a named function, the compiler may assume that a call to a function of the same name refers to the same function (unless that function has been declared not inline). Although this approach allows such calls to be compiled slightly more efficiently, debugging tools such as `trace` and `advise` violate this assumption.

This aspect of the compiler's behavior can also be controlled through appropriate use of compiler policy objects.

Compiler policy objects

A `compiler-policy` object is a data structure whose components advise the compiler of the desirability of performing (or avoiding) certain optimizations. Usually, compiler policy objects specify how `optimize` declarations are to be interpreted. (For `optimize` declarations, see *Common Lisp: The Language.*)

Separate compiler policy objects are used for file compilation and for interactive compilation, although the default values of these objects specify identical behavior.

The function `new-compiler-policy` is used to create a compiler policy object and to override the implementation's default behavior. The functions `set-compiler-policy` and `current-compiler-policy` set and return the compiler policy used for interactive compilation (including the use of the `compile` function). The functions `set-current-file-compiler-policy` and `current-file-compiler-policy` set and return the policy object used to compile functions that will be saved in `fasl` files.

compiler-policy

[*Class name*]

Description The `compiler-policy` class is the class of compiler policy objects.

new-compiler-policy

[*Function*]

Syntax

```
new-compiler-policy &key :allow-tail-recursion -
elimination :inhibit-register-allocation :trust -
declarations :open-code-inline :inhibit-safety -
checking :inhibit-event-polling :inline-self-calls
:allow-transforms :force-boundp-checks :allow -
constant-substitution
```

Description The `new-compiler-policy` function creates and returns a new compiler policy in which the default specifications of behavior are overridden by the values associated with the indicated keyword arguments.

Each of these keywords may take one of the following values:

`nil`, which specifies that the associated behavior is suppressed

`t`, which specifies that the associated behavior is performed

a function that takes arguments as described here and returns a Boolean value

Unless otherwise noted, the functions are called with a (possibly null) lexical environment as their lone argument. To determine what value to return, they may reasonably use functions such as `declaration-information` to extract information about the `optimize` declarations (and other declarations) in effect in that environment.

Setting a new compiler policy completely shadows any existing policy.

Arguments

`:allow-tail-recursion-elimination`
When this value is `nil` or a function that inspects the environment and returns `nil`, the compiler does not eliminate tail recursion. The default value is a function that returns true unless the value of the `debug` optimize quantity in the environment is 3.

`:inhibit-register-allocation`
When this value is true or a function that returns true, the compiler does not allocate frequently used values in registers. The default value is a function that returns true when the value of the `debug` optimize quantity in the environment is 3.

`:trust-declarations`
When this value is true or a function that returns true and the value of the `safety` optimize quantity in the environment is not 3, the compiler attempts to exploit type declarations to produce faster and/or smaller code. If those declarations are incorrect, the resulting code may show unpredictable behavior. The default value is a function that returns true if, within the environment, the value of the `speed` optimize quantity is not less than the value of the `safety` optimize quantity.

`:open-code-inline`

When this value is true or a function that returns true and the compiler sees a call to a function that has been declared inline or a call to a primitive operation implemented in the MCL kernel, the compiler may replace that call with a larger (but possibly faster) sequence of instructions. The default value is a function that returns `t` if, within the environment, the value of the `speed` optimize quantity is two or more units greater than the value of the `space` optimize quantity.

`:inhibit-safety-checking`

When this value is true or a function that returns true and the value of the `safety` optimize quantity in the environment is not 3, the compiler is licensed to omit safety checks. (When the compiler performs safety checks, incorrect programs cause errors to be signaled.) The default value is a function that returns `t` if, within the environment, the value of the `speed` optimize quantity is 3 and the value of the `safety` optimize quantity is 0.

`:inhibit-event-polling`

When this value is true or a function that returns true, the compiler may omit instruction sequences that poll for events from loops that are otherwise uninterruptible. The default value is a function that returns `t` if, within the environment, the value of the `speed` optimize quantity is 3 and the value of the `safety` optimize quantity is 0.

`:inline-self-calls`

When this value is true or a function that returns true, the compiler may assume that within a globally named function, calls to a global function of the same name may be compiled without reference to the function cell of the symbol that names that function. The default value is a function that returns `t` unless the value of the `debug` optimize quantity in the environment is 3.

`:allow-transforms`

When this value is true or a function that returns true, the compiler expands compiler macros and may perform other source-to-source transforms. The default value is a function that returns `t` unless the value of the `compilation-speed` optimize quantity in the environment is 3 or the value of the `debug` optimize quantity in the environment is 3.

`:force-boundp-checks`

When this value is true or a function that returns true or when the value of the `safety` optimize quantity is 3, the compiler ensures that variables are bound before referencing them. If a function is provided, it should take two arguments, a symbol that names a variable and a lexical environment. Ordinarily, the compiler omits checking the binding of the variable with `boundp` if the variable reference appears within the scope of a special binding of that variable, or if the reference appears in a file that is being compiled with `compile-file` and appears after a `defvar` or `defparameter` form that defined that variable.

`:allow-constant-substitution`

When this value is true or a function that returns true, the compiler is allowed to substitute the value of a named constant for a reference to the constant. The default value is a function of three arguments: a symbol that names a constant, the value of that constant, and the current lexical environment. The function ignores those arguments and returns `t`.

`current-compiler-policy` [*Function*]

Syntax `current-compiler-policy`

Description The `current-compiler-policy` function returns the current `compiler-policy` used by interactive compilation.

set-current-compiler-policy [*Function*]

Syntax `set-current-compiler-policy &optional` *policy*

Description The `set-current-compiler-policy` function sets the default `compiler-policy` used by interactive compilation to *policy*. If *policy* is `nil` or unsupplied, a copy of the default compiler policy is used.

Argument *policy* A compiler policy.

current-file-compiler-policy [*Function*]

Syntax `current-file-compiler-policy`

Description The `current-file-compiler-policy` function returns the current `compiler-policy` used by file compilation.

set-current-file-compiler-policy [*Function*]

Syntax `set-current-file-compiler-policy &optional` *policy*

Description The `set-current-file-compiler-policy` function sets the default `compiler-policy` used by file compilation to *policy*. If *policy* is `nil` or unsupplied, a copy of the default compiler policy is used.

Argument *policy* A compiler policy.

ignore-if-unused [*Declaration*]

Syntax `ignore-if-unused`

Description The `ignore-if-unused` declaration behaves the same way as `ignore`, but does not signal a warning if the variable is used. This declaration is usually used in macroexpansions.

Miscellaneous MCL expressions

The following MCL expressions provide miscellaneous useful functionality not in Common Lisp.

always-eval-user-defvars [*Variable*]

Description The `*always-eval-user-defvars*` variable determines how Macintosh Common Lisp treats the evaluation of `defvar`.

If an entire buffer or a selection in a buffer is evaluated, `defvar` is never equivalent to `defparameter`.

If the value of this variable is true, then `defvar` is equivalent to `defparameter` when evaluated as a single expression from a Fred buffer or when typed to the Listener.

If the value of this variable is `nil` *(the default)*, *then* `defvar` acts in the normal Common Lisp way (see *Common Lisp: The Language*, pages 86–87).

require-type [*Function*]

Syntax `require-type` *argument type*

Description The `require-type` function is like the Common Lisp `check-type` macro, except that it returns a value rather than using `setf`, and so can be done entirely not inline. If *argument* is of the same type as *type*, `require-type` returns *argument*. If not, it signals an error.

Arguments

argument	Any argument.
type	A type.

Example

```
? (require-type (front-window) 'window)
#<LISTENER "Listener" #x42DDB1>
? (require-type (target) 'listener)
> Error: value #<WINDOW "Inspector Central" #x4618A1> is not of the
expected type LISTENER.
```

structure-typep [*Function*]

Syntax `structure-typep` *form type*

Description The `structure-typep` function returns `t` if *form* is of the given structure type *type* or if it includes *type*. Otherwise it returns `nil`. This function is used by `defstruct` predicates.

Arguments

form	Any form.
type	A type.

structurep [*Function*]

Syntax `structurep` *form*

Description The `structurep` function returns `t` if the given object is a named structure; otherwise it returns *nil* .

Argument

form	An MCL form.

loading-file-source-file [*Variable*]

Description The `*loading-file-source-file*` variable is bound to the namestring of the file containing the source code while `load` is loading a source code file or a `fasl` file. The value of this variable is either the name of the file itself if it is a source code file, or the name of the file that was compiled to create it if it is a `fasl` file. Its default value is `nil`.

Appendix B Snapshots and Application Versions of Macintosh Common Lisp

Contents

The snapshot utility / 648
Creating images / 648
Removing Macintosh pointers / 651
The top-level function / 653
Setting the top-level function / 653
Catching errors and aborts / 654
Using `*idle*` and `event-dispatch` / 655
Using `eval-enqueue` / 655

This chapter describes the “snapshot utility,” which you can use to save images (snapshots) of MCL environments and to create customized applications.

The snapshot utility

This chapter describes a utility that you can use to save complex MCL states and create customized MCL applications.

To create a customized version of Macintosh Common Lisp, arrange your system the way you like (by loading files and so on) and then call `save-application`. The `save-application` function produces an MCL application containing your heap image, then restarts Macintosh Common Lisp. The heap image will contain all the states from files you have loaded and work you have done up to the moment you take the snapshot.

The snapshot utility is useful in two ways:

- If you have created a very complex system state, during development or debugging, you can save the state as an image.
- If you are creating an application with Macintosh Common Lisp, you can prototype the application, exactly as you want it, then save the application as an image.

If what you want is a set of predefined system startups, you can use an MCL `init` file. See the section "The `init` File" in Chapter 1, "The Macintosh Common Lisp Environment."

Creating images

The following functions and variables are used in the creation of images.

`save-application` [*Function*]

Syntax `save-application` *pathname* `&key :toplevel-function :creator :excise-compiler :size :resources :init-file :clear-clos-caches`

Description The `save-application` function creates a stand-alone image containing the functionality of the current Lisp environment. When `save-application` is finished, Macintosh Common Lisp exits to the Finder.

Before dumping a heap image, `save-application` closes all windows, takes down and remembers the current menu bar, and disposes of other pointers to the Macintosh heap. It then executes all the functions on the list `*save-exit-functions*`. Finally, Macintosh Common Lisp performs a garbage collection and dumps the heap image.

You can add functions to the list `*save-exit-functions*`. You may wish to do this if you want to save and restore a certain state in a particular way.

When a heap image is restarted, Macintosh Common Lisp restores Macintosh pointers used by the system, resets the logical hosts `"ccl:"` and `"home:"`, and reinitializes some system configuration variables. Then it runs all the functions specified by `def-load-pointers` in the order they were specified.

Arguments

pathname A pathname for the image to be created. If a file with that name already exists, Macintosh Common Lisp deletes it before `save-application` is performed.

`:toplevel-function`
The top-level function to be set up when the heap image is restarted. Top-level functions are described in the section "The Top-Level Function" later in this appendix. This argument defaults to the current top-level function. If supplied, the top-level function must be a compiled function object (it cannot be a symbol naming a function).

If you provide a top-level function, the resulting application does not load an `init` file or `finder-parameters` and does not print a greeting in the Listener.

`:creator` The `mac-file-creator` os-type for the saved application. The default is `:CCL2`. Set it to something else if you do not want the Finder to consider your saved application the creator of all your MCL files.

`:excise-compiler`
An argument specifying whether to disable the compiler in the resulting application. If the value of this keyword is true, the compiler is disabled. Its default value is `nil`.

`:size` A size specification, which is either a nonnegative integer or a list of two nonnegative integers: (*preferred-size minimum-size*). If present, this argument sets the preferred and minimum partition sizes in the application's `SIZE (-1)` resource. Any `SIZE(0)` resources that the Finder may have added to Macintosh Common Lisp are not copied to the resulting application.

`:resources` A list of resource specifications, where each resource specification is a list of the form (*data resource-type resource-id* `&optional` *resource-name*).

If `:resources` is specified, any resources matching *resource-type* and *resource-id* are not copied from Macintosh Common Lisp to the resulting application. If *data* is non-`nil`, this specification causes *data* to be added as a resource of the specified type and ID.

A resource specification can also be a symbol or function, in which case `funcall` will be run on it with a single argument, the name of the file being saved. When the function is called, the current resource file will be the resource file of the application being saved.

`:init-file` An argument specifying the pathname of an `init` file to load when the MCL image is started, or `nil` (the default), in which case no `init` file is loaded. The `init` file need not be in the same folder as Macintosh Common Lisp; you can specify any pathname you wish.

`:clear-clos-caches`
An argument specifying whether caches are cleared when the application is saved. The default value is true.

Example

Here is an example of saving an application using the `:resources` keyword.

```
(save-application "My App" :resources '(copy-my-apps-resources))

(eval-when (:compile-toplevel :execute :load-toplevel)
  (require :resources))

(defun copy-my-apps-resources (resource-file)
  (declare (ignore resource-file))
  (let ((refnum (#_CurResFile)))
    (with-open-resource-file (my-refnum "My App.r")
      ...)))
```

Removing Macintosh pointers

An important restriction on saved images is that no data on the Macintosh heap is preserved across saves and restarts. When you save an application, any pointers or handles to the Macintosh heap become invalid. For this reason, you should dispose of all Macintosh handles and pointers before doing `save-application`.

If your program maintains pointers to the Macintosh heap, you should deallocate these with a function included on the list `*save-exit-functions*`. You can then reinitialize the pointers and handles with functions specified by `def-load-pointers`.

- *Note:* Leftover Macintosh pointers in a heap image can cause system crashes and other erratic behavior.

The `def-load-pointers` macro can be used to allocate memory on the heap during startup.

`*lisp-cleanup-functions*` [*Variable*]

Description The `*lisp-cleanup-functions*` variable contains a list of functions of no arguments on which `funcall` is run just before Macintosh Common Lisp exits (via `quit` or `save-application`). These functions are called just after the windows are closed.

When saving an application, the functions in `*lisp-cleanup-functions*` are run, then the functions in `save-exit-functions*` are run.

`*save-exit-functions*` [*Variable*]

Description The `*save-exit-functions*` variable contains a list of functions to be called when an image is saved. These functions should perform any preparation necessary for the image saving. The functions are called in the order in which they appear in the list.

When saving an application, the functions in `*lisp-cleanup-functions*` are run, then the functions in `*save-exit-functions*` are run.

def-load-pointers

[*Macro*]

Syntax `def-load-pointers` *name arglist* `&body` *body*

Description The `def-load-pointers` macro is usually used to allocate memory on the Macintosh heap. It associates *name* with `#'(lambda` *arglist* . *body* `)` in a list. If *name* is already on the list, the macro replaces it. If it is not, `def-load-pointers` adds *name* and its function to the list and runs `funcall` on it.

When Macintosh Common Lisp starts up, it calls the functions specified by `def-load-pointers` in the order in which they were specified on the list. This occurs before the `init` file is loaded.

Arguments

name	The name to associate with a function.
arglist	The argument list of the function. The function is called with no arguments, hence this argument should always be `nil`.
body	The body of the function.

lisp-startup-functions

[*Variable*]

Description The `*lisp-startup-functions*` variable contains a list of functions of no arguments on which `funcall` is run after Macintosh Common Lisp starts, just before it enters the top-level function (usually the Listener's read loop). The functions contained in `*lisp-startup-functions*` are run after the functions specified by `def-load-pointers` and before the `init` file is loaded. The functions are called in reverse order from the order in which they appear in the list.

The top-level function

When Macintosh Common Lisp is running, a Lisp function is always running. For example, during normal programming the function `toplevel-loop` runs. This function takes input from the Listener (and other buffers), evaluates it, prints out the result, and then gets another input.

Whenever the top-level function returns, the Lisp kernel arranges for it to be called again. In this way Macintosh Common Lisp can run indefinitely. To quit Lisp, you set the top-level function to `nil` and return from the current top-level function.

There are two ways to set the top-level function. The first is to make an explicit call to the function `%set-toplevel` . The second is to specify a top-level function when you call `save-application` .

Setting the top-level function

Macintosh Common Lisp actually has two notions of the top-level function. One is the function that is currently running. The other is the function (stored in a system variable) that is called when the currently running top-level function returns. During most programming sessions these are both the function `toplevel-loop` .

When you call `%set-toplevel` , you set the pending top-level function. Setting the pending top-level function does not, however, cause an automatic exit from the top level function that is currently running. The new pending top-level function is not called until the top-level function that is currently running exits, either by returning or through a system throw.

`toplevel` [*Function*]

Syntax `toplevel`

Description The `toplevel` function throws past all pending catches to the Lisp kernel, which then calls the pending top-level function. If the pending top-level function is `nil`, Lisp exits to the Finder.

`%set-toplevel` [*Function*]

Syntax `%set-toplevel &optional` *new-toplevel-function*

Description The `%set-toplevel` function sets *new-toplevel-function*, if supplied, as the pending top-level function and returns the previous pending top-level function. If *new-toplevel-function* is not supplied `%set-toplevel` simply returns the pending top-level function.

If *new-toplevel-function* is supplied and `nil`, then Macintosh Common Lisp exits to the Finder when the current top-level function exits.

Argument *new-toplevel-function*
A compiled function object or `nil`. This argument cannot be a symbol.

`toplevel-loop` [*Function*]

Syntax `toplevel-loop`

Description The `toplevel-loop` function implements the read loop. During program prototyping, it may be useful to switch between your own top-level function and `toplevel-loop`.

Catching errors and aborts

You may want your top-level function to catch errors and aborts. If you don't, errors cause the Listener containing the error message to appear. Catching aborts is especially important. If your top-level loop doesn't catch them, pressing Command-period causes the error `Can't throw to tag :abort` to be displayed. You remain in your top-level function.

The following example sets up a miniature read-eval-print loop to replace the standard top-level loop. It also shows what happens if you don't catch an abort.

```
? (defun new-top (&aux form)
    (setq form (read))
    (if (eq form 'done)
        (%set-toplevel #'toplevel-loop)
        (print (eval form))))
NEW-TOP
? (%set-toplevel #'new-top)
#<COMPILED FUNCTION TOPLEVEL-LOOP>
? (toplevel)
(+ 10 10)
20              ;typed command-period here, to abort
> Error: Can't throw to tag :abort.
```

```
> While executing: "Unknown"
(* 20 20)
400              ;done
```

Using *idle* and event-dispatch

The variable *idle* lets your top-level function (or other parts of your program) tell Macintosh Common Lisp when it is idling. If the value of *idle* is true, Macintosh Common Lisp uses *idle-sleep-ticks* instead of *running-sleep-ticks* as the sleep time for calls to #_WaitNextEvent. This approach gives the maximum amount of time to background processes. During normal operation, *idle* is bound to t by the top-level loop as it awaits input.

If your main program consists of a small loop waiting for events, you should place a call to event-dispatch in the loop. This improves the response time to events and gives background applications more time when your program is idling.

These are described in Chapter 11, “Events.”

Using eval-enqueue

An event (such as choosing a command or clicking a dialog box) that begins a long process should not simply execute the process. If it does, the process runs with interrupts disabled and future events are ignored until the process returns. This is fine for quick actions but can be a problem for time-consuming actions. The process is described in more detail in Chapter 11, “Events.”

The solution to this problem is for event actions to queue up forms. The forms are received and processed in order by the top-level loop, which keeps interrupts enabled.

There are many ways to queue up forms. The simplest is to push them onto a list and have the top-level loop pop things from the list.

In many cases you may wish to have a single mechanism that works both with your top-level function and with the standard top-level function toplevel-loop. To do this, you use eval-enqueue to queue up forms. When toplevel-loop is running, it automatically gets these forms and evaluates them. Your top-level loop can do the same thing by calling the function get-next-queued-form.

eval-enqueue [*Function*]

Syntax `eval-enqueue` *form*

Description The `eval-enqueue` function queues up *form* for evaluation in the read-eval-print loop. The `eval-enqueue` function returns immediately. This means that *form* is not executed at the uninterruptible event-handling level but instead is executed as if it had been entered into the Listener. (It is executed only when other forms entered into the Listener or queued up have returned.)

This function is useful for initiating programs from within event handlers. The form is executed as part of the normal read-eval-print loop rather than as part of an event handler. This means that other events can be processed during the execution of *form*.

Note that `eval-enqueue` is a function, and so its argument is evaluated. The result of this evaluation is put into the read-eval-print loop.

Argument *form* An MCL form.

get-next-queued-form [*Function*]

Syntax `get-next-queued-form`

Description The `get-next-queued-form` function returns the next form from the pending queue or returns `nil` if there are no forms pending. A second value returned is `t` if there was a pending form and `nil` if there was no pending form.

During programming sessions, queued-up forms include text entered in the Listener and evaluated from buffers as well as forms passed to `eval-enqueue`.

Appendix C The Common Lisp Object System

Contents

Macintosh Common Lisp and the Common Lisp Object System standard / 659
How CLOS differs from Object Lisp / 659
Definitions / 660
Classes and their superclasses / 660
Slots / 661
Instances / 661
Generic functions and methods / 661
Classes and instances / 662
Creating a class with the macro `defclass` / 662
Creating an instance and giving its slots values / 664
Redefining a class / 665
Allocating the value of a slot in a class / 666
Classes as prototypes of other classes / 667
Methods / 667
Defining a method and creating a generic function / 668
Congruent lambda lists / 669
Documentation of methods / 669
Defining methods on instances / 670
Creating and using accessor methods / 670
Customizing initialization with `initialize-instance` / 672
Creating subclasses and specializing their methods / 673
Method combination / 674
The primary method / 674
The primary method and the class precedence list / 674
Examples of classes with multiple superclasses / 675
When there is a conflict: Choosing between methods / 676
Choosing between methods associated with direct and with more distant superclasses / 677
Creating auxiliary methods and using method qualifiers / 678
Mixin classes and auxiliary methods / 680
Extended example / 680

This appendix is directed toward users who are familiar with Object Lisp, as implemented by Macintosh Common Lisp version 1.3 and previous versions of Macintosh Common Lisp. It provides a summary of the differences between Object Lisp and CLOS, with examples, and includes an extended example of CLOS syntax.

This chapter is not a full introduction to CLOS concepts. For more extended discussions of CLOS, consult the section “If You Are Learning CLOS” in Appendix F, “For More Information.”

If you are already familiar with the syntax of CLOS, you should read Appendix D, “Converting Your Files to CLOS.” In that appendix you will find general rules for conversion, some specific changes to watch out for, and an extended comparison of CLOS and Object Lisp code.

Macintosh Common Lisp and the Common Lisp Object System standard

Macintosh Common Lisp uses the **Common Lisp Object System (CLOS),** the proposed ANSI standard for the Common Lisp community. CLOS is based on five years of experience with Common Lisp, augmented by substantive proposals and changes suggested by its developers and users, and coordinated by X3J13, a subcommittee of ANSI committee X3.

Common Lisp: The Language, second edition, written by Guy L. Steele and others, provides detailed summaries of the X3J13 committee's thinking. Though not an official standards document, the second edition of Steele extensively documents almost all functions that will be proposed in the draft standard. Some changes to the standard have been approved since the time of publication of *Common Lisp: The Language*. Where they affect Macintosh Common Lisp, these changes are documented in the appropriate sections of this manual.

Before receiving official approval as an American National Standard, CLOS will undergo a period of review, required by ANSI. While it is unlikely that this review will produce radical or incompatible changes, it is probable that details of the language will be rethought during review. Those changes will be reflected in future releases of Macintosh Common Lisp.

Wherever possible, the developers of Macintosh Common Lisp have included routines so that your older MCL code will continue to run. However, your Object Lisp code will require conversion; see Appendix D, "Converting Your Files to CLOS."

As you write new code, you should consult the proposed standard as described in Steele or in the draft ANSI standard.

How CLOS differs from Object Lisp

Macintosh Common Lisp version 1.3 used the object protocol Object Lisp, which supported multiple inheritance but only simple method combination. In Object Lisp you could make an object `fourth-grader` with an instance `amanda`, then create `bob`, which inherited from `amanda`, and `george`, which inherited from `bob`. CLOS uses a class-instance protocol. In CLOS you cannot make a subclass from `amanda`, because `amanda` is not a class. Instead you must create another class, possibly a subclass of `fourth-grader`.

Binding and scoping have changed substantially. Instead of having object variables, CLOS classes and instances have slots. Rather than asking an object to run its version of an object function, CLOS applies methods of generic functions to instances.

For an extended example showing differences in coding, consult Appendix D, "Converting Your Files to CLOS."

Definitions

The basic concepts of CLOS include classes, slots, instances, generic functions, and methods.

Classes and their superclasses

A **class** is an object that determines the inherited behavior of other objects, called its **instances.**

- Classes are organized in a hierarchy.
- A class can inherit structure from other classes, which are called its **superclasses.** Its immediate superclasses are called its **direct superclasses.** A class is a **subclass** of each of the classes from which it inherits.
- There are two classes at the top of the class hierarchy. The class named `t` has no superclasses and is a superclass of every class except itself. The class named `standard-object` is a superclass of all the classes programmers create. (However, it is not a superclass of some built-in classes.)
- A class has a name.
- A class has a **class precedence list,** which is the total ordering of the class and all its superclasses. When a class is defined, the order in which its direct superclasses are mentioned determines the order of the class precedence list.

Most Common Lisp types now correspond to a class with the same name. That is, for a type `macptr`, there will also be a class `macptr`.

Slots

A class is associated with a set of **slots**.

Slots are used for storing information associated with a class and its instances. A slot is described by a **slot specifier.** Each slot specifier includes the name of the slot and zero or more slot options. A class is also associated with a set of **class options,** which are slot options with a common value for the whole class. Slot options and class options control the following:

- determining default initial values for a given slot
- reading and writing the values of slots
- controlling whether a value for a given slot is shared by all instances of a class or whether each instance has its own value for the slot
- supplying a set of initialization arguments and default values to be used when instances are created

Instances

Unlike Object Lisp, Macintosh Common Lisp version 2 maintains a clear distinction between classes and instances.

- An **instance** is a Common Lisp object, one of a group of zero or more instances of a class. An instance may have a default state derived from the class definition, or may have its own particular state.
- An instance may use a method defined on its class or on one of the class's superclasses, or may have its own method. See the next section for the definition of methods.
- Unlike Object Lisp objects, CLOS instances cannot generally be used as classes.
- However, every Common Lisp object, even a class object, is an instance of some class (for example, a class object may be an instance of the class `class`).

Generic functions and methods

Behavior is expressed through **generic functions,** Lisp functions whose behavior depends on the classes or identities of the arguments supplied.

- A generic function has **methods** for a number of classes. That is, the generic function has a number of ways to perform a procedure, each way designed to be used for a specific class. For example, the generic function `view-draw-contents` has one method for simple views, one for views, one for windows, and so on. That is, it knows how to draw the contents of a simple view, the contents of a view, and so on.

These methods are related but usually not identical (for instance, the `view-draw-contents` method for `simple-view` doesn't need to handle subviews, but the one for `view` does).

- When you want to perform an operation on some objects, you apply a generic function to an instance. By determining which methods of the generic function are associated with the instance's class, the generic function knows which of its methods to apply to the instance.
- Methods are applied to instances through method combination. A generic function determines one **primary method** suitable for an instance—one basic procedure that is the most appropriate to the instance. It may also call other methods through `call-next-method`. One or more **auxiliary methods** may be run before or after the primary method. The entire group of methods applicable to the instance is called the **effective method.**
- Methods have arguments, just as functions do.
- Methods of the same generic function must always have the same number of required and optional arguments; that is, they must have **congruent lambda lists.**
- The required arguments of a method are **specialized;** that is, each argument is associated with a class or an instance. The specializers of a method determine whether the method is appropriate for a given set of arguments.

Classes and instances

Classes can be created and redefined. You can create instances of classes and give values to slots in a class or an instance.

Creating a class with the macro `defclass`

The macro `defclass` creates a new class. Its first argument is a list of the superclasses of the new class. If the argument is `nil`, the new class is based on `standard-object`.

Its second argument is a list of slot specifiers. Each slot is a list, the first element of which is a symbol that names the slot. The second and third elements are the slot's **initialization argument** and default **initial value form,** if it has them. These appear in definitions as the `initarg` and `initform` keywords.

Initialization arguments are keyword arguments used to supply the values for slots when new instances are created. Initial value forms provide a mechanism for a user to give a default initial value form for a slot.

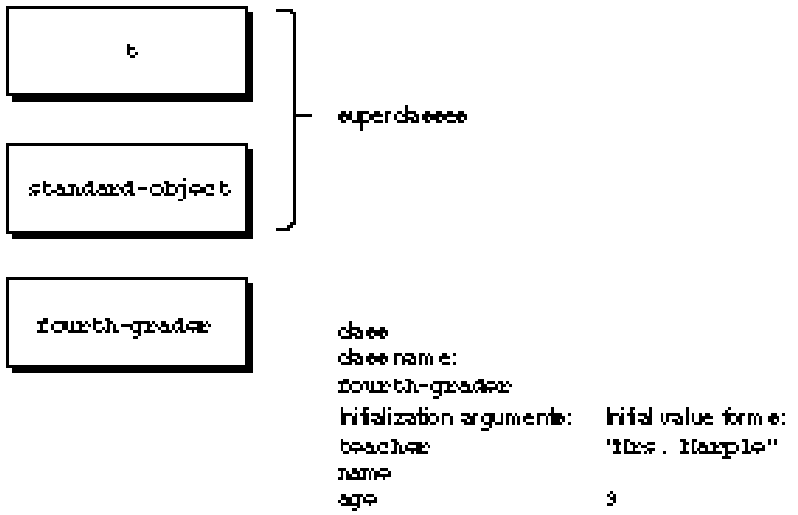
The macro `defclass` returns the new class object.

Here is an example of the use of `defclass`.

```
? (defclass fourth-grader ()
    ((teacher :initarg :teacher
              :initform "Mrs. Marple")
     (name :initarg :name)
     (age :initarg :age :initform 9)))
#<STANDARD-CLASS FOURTH-GRADER>
```

Figure C-1 shows a graphic representation of how the class `fourth-grader` is built. This class is associated with three slots—`name`, `age`, which has the default initial value 9, and `teacher`, which has the default initial value `"Mrs. Marple"`. The name of the new class is the symbol `fourth-grader`.

Figure C-1 The class `fourth-grader`



Creating an instance and giving its slots values

To create an instance of the class, use the generic function `make-instance`. This function creates and returns a new object based on its argument, which should be a class.

```
? (setq john (make-instance 'fourth-grader))
#<FOURTH-GRADER #x436B51>
```

An instance has slots as determined by its class and that class's superclasses.

You can set the values of an instance's slots when you create it. The following example creates an instance of `fourth-grader` and uses the initialization argument `:teacher` to override the default value for that slot:

```
? (setf john (make-instance 'fourth-grader
                            :teacher "Ms. Hsu"))
#<FOURTH-GRADER #x477181>
```

The function `slot-value` retrieves the value of a slot. This function takes two arguments, the class or instance and the name of the slot.

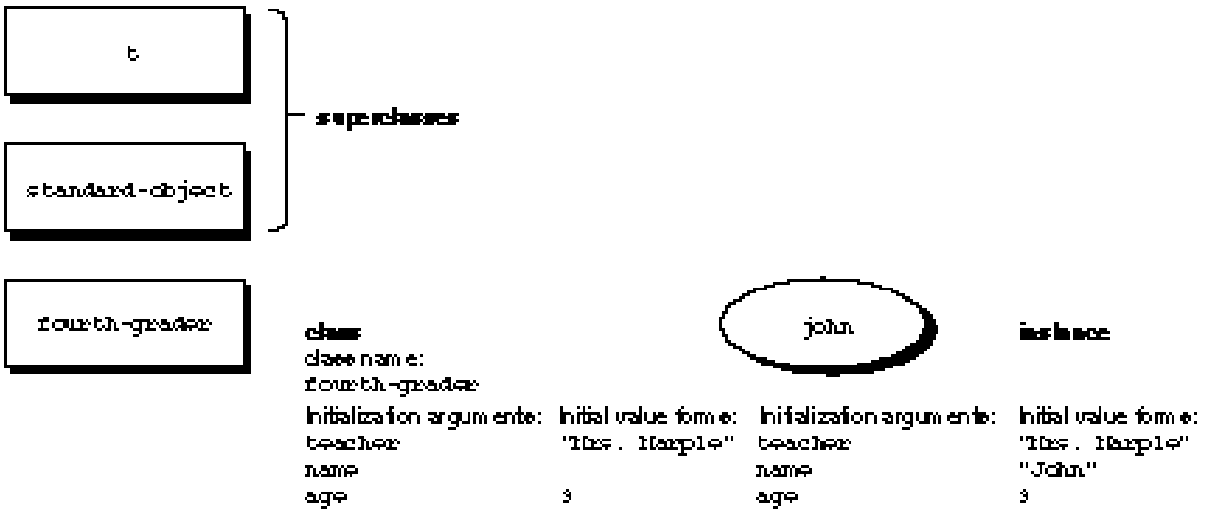
```
? (slot-value john 'teacher)
"Ms. Hsu"
```

You can set the value of most slots of an already created instance by using `setf` with the name of the slot, for example, to give the instance `john` its own name (Figure 2) or to change the value of the `:teacher` slot.

```
? (setf (slot-value john 'name) "John")
"John"
?(slot-value john 'name)
"John"
? (setf (slot-value john 'teacher) "Ms. Miller")
"Ms. Miller"
?(slot-value john 'teacher)
"Ms. Miller"
```

u *Note:* Accessor methods provide a simpler syntax than `slot-value` does. For more details, see "Creating and Using Accessor Methods" later in this appendix.

n **Figure C-2** An instance of `fourth-grader` with a value in the `name` slot



To find the slot value associated with an instance, CLOS first looks up the value associated with the slot at the instance level. If the slot is unbound at the instance level, CLOS looks up the value associated with the instance's class. If the slot is still unbound, CLOS looks for the value in the slot associated with the first class that is a member of the class's class precedence list, then looks in the slot of the second class, and so on until it finds one value, which it returns.

Redefining a class

To redefine a class, simply edit the previous definition of the class and evaluate it again. Here is a revised definition of `fourth-grader` , adding a `school` slot :

```
? (defclass fourth-grader ()
    ((teacher :initarg :teacher
              :initform "Mrs. Marple")
     (school :initarg :school
             :initform "Lawrence School")
     (name :initarg :name)
     (age :initarg :age :initform 9)))
#<STANDARD-CLASS FOURTH-GRADER>
```

After the redefinition, all instances of `fourth-grader`, including ones already created, will have a `school` slot.

```
? (slot-value john 'school)
"Lawrence School"
```

When you create a new instance, you can specify a value for the `school` slot.

Allocating the value of a slot in a class

You can specifically instruct a slot to be associated not with an instance, but with the instance's class, by using the `:allocation` option when you define the slot.

The two `:allocation` options are: `:instance` and `:class`. The `:instance` option is the default; this option means that each instance of a class has local storage for the slot that can be associated with its own value. The `:class` option specifies that the class stores the value of the slot. A slot of this kind is called a **shared slot** or a class slot. The following code defines `picky-fourth-grader`, associating it with a class slot for `subjects-studied`:

```
? (defclass picky-fourth-grader
    (fourth-grader)
    ((subjects-studied
      :initarg subjects-studied
      :initform '(English math woodworking)
      :allocation :class)))
#<STANDARD-CLASS PICKY-FOURTH-GRADER>
```

Since all instances of `picky-fourth-grader` access the same `subjects-studied` slot, you can `setf` the class slot value through any instance:

```
? (setf zane (make-instance 'picky-fourth-grader))
#<PICKY-FOURTH-GRADER #x40CDC9>
? (setf justus (make-instance 'picky-fourth-grader))
#<PICKY-FOURTH-GRADER #x40EFC8>
? (setf (slot-value zane 'subjects-studied)
   '(English math woodworking phenomenology))
(ENGLISH MATH WOODWORKING PHENOMENOLOGY)
? (slot-value justus 'subjects-studied)
(ENGLISH MATH WOODWORKING PHENOMENOLOGY)
```

Classes as prototypes of other classes

As you can see from the example "Creating an Instance and Giving Its Slots Values" earlier in this appendix you can build classes from other classes. Subclasses may add slots or change the default initial value of a slot that already exists. Subclasses may also have their own methods (see the next section, "Methods").

This is the syntax for defining a new class. It defines `modern-fourth-grader`, built on `fourth-grader`, with an additional slot, `computer`, and a new default value for `teacher`.

```
? (defclass modern-fourth-grader
    (fourth-grader)
    ((computer :initarg :computer
               :initform "Macintosh")
     (teacher  :initform "We use HyperCard stacks.")))
#<STANDARD-CLASS MODERN-FOURTH-GRADER>
? (setf mariah (make-instance 'modern-fourth-grader))
#<MODERN-FOURTH-GRADER #x51A209>
? (slot-value mariah 'teacher)
"We use HyperCard stacks."
? (slot-value mariah 'computer)
"Macintosh"
? (setf (slot-value mariah 'teacher) "We program our own in
Macintosh Common Lisp.")
"We program our own in Macintosh Common Lisp."
```

The instance `mariah` inherits slots and values it does not redefine.

```
? (slot-value mariah 'school)
"Lawrence School"
```

Methods

CLOS is built on the idea that many functions operate in different ways when called on different classes of objects. Any function that operates in more than one way is called a **generic function**, a Lisp function whose behavior depends on the parameters supplied to it. Like any other Lisp function, a generic function can be passed as an argument and returned as a value, and it does all the other things a function does.

But while an ordinary function has a single body of code that is always executed, a generic function has a set of bodies of code, called **methods.** When a generic function is called on an instance, the function looks at the class or classes of the instance. In the simplest case, the generic function has a method for the instance's class. (For example, the generic function `#'set-part-color` has a method for `table-dialog-item`. Calling `#'set-part-color` on an instance of `table-dialog-item` calls that method. Calling `#'set-part-color` on an instance of `radio-button-dialog-item` calls a different method.)

In cases where the instance's class has a complex parentage, methods may be combined. What the function does—which bodies of code are called and how they are combined—is calculated from the **class precedence list,** the total ordering of the set of classes from which a class inherits. It also depends on the method combination type.

You can define a generic function by using the macro `defgeneric`. More often, however, you use `defmethod`, which defines a method on a generic function for a particular class. If the generic function does not already exist, `defmethod` creates one.

Defining a method and creating a generic function

By using `defmethod` to define a method, you automatically generate a generic function with the same name as the method. For example, you can create a generic function `#'say` by using `defmethod` to define a method `say`. Here is an example of the use of `defmethod`.

The following function prints a sentence of an instance of `fourth-grader` surrounded by enthusiastic punctuation, then returns `nil`:

```
? (defmethod say ((child fourth-grader) sentence)
    (princ "***")
    (princ sentence)
    (princ "!***")
    (terpri))
SAY
? (setq billy-crystal (make-instance 'fourth-grader
                        :name "Billy Crystal"))
#<FOURTH-GRADER #x60B5F1>
```

When `#'say` is called on `billy-crystal`, it checks the type of `billy-crystal` and finds that it is a `fourth-grader`. The generic function checks to see whether it has a method for `fourth-grader`; since it does, it runs that method.

```
? (say billy-crystal "Marvelous")
***Marvelous!***
NIL
```

Congruent lambda lists

All methods on a generic function have congruent lambda lists. That is, they have the same number of required and optional arguments. (For full details on congruent lambda lists, see *Common Lisp: The Language*, pages 791–792.) For example, the generic function `#'say` always requires an instance to which it is applied and a sentence.

Documentation of methods

In this book, methods on generic functions are documented as follows. The function name is in bold in the header. The syntax of methods is given below the header. The name of the class for which the generic function has a method is in Courier (this class is called the **specializer**). The arguments are in italics, and the initialization argument keywords (if any) are in Courier. The action of the function is described, and the arguments are described at the end of the body of the definition.

say

[*Generic function*]

Syntax `say` (*child* `fourth-grader`) *sentence*

Description The `say` generic function prints *sentence* preceded and followed by three asterisks. It returns `nil`.

Arguments

child	An instance of the class `fourth-grader`.
sentence	A message to print, usually a string.

Defining methods on instances

You can specialize methods on individual instances as well as on entire classes. If you need a method for `#'say` that is called only on `alisa`, an instance of `fourth-grader`, then specify `child` to be `eql` to `alisa`, as shown in the following example. The expression `(declare (ignore sentence))` avoids a compiler warning. The function `(call-next-method)` calls the next most specific method, which is the method on `fourth-grader`:

```
? (defmethod say ((child (eql alisa)) sentence)
    (declare (ignore sentence))
    (call-next-method)
    (princ "I have an HP calculator")
    (terpri))
#<Method SAY ((EQL #<FOURTH-GRADER #x436B51> T)>

?(say alisa "Hello")
***Hello!***
I have an HP calculator
NIL
```

Applied to other fourth-graders, such as `sharon`, `say` still calls its usual `fourth-grader` method.

```
?(say sharon "Hello")
***Hello!***
NIL
```

- *Note:* The function `(call-next-method)` is perfectly acceptable here, but CLOS more commonly would use an `:after` method. Method combination is discussed later in this appendix in the section "Creating Auxiliary Methods and Using Method Qualifiers."

Creating and using accessor methods

The macro `defclass` provides syntax for automatically generating methods to read and write slots. Because the accessor is the most common of these methods, the whole group is called **accessor methods**. You can request three kinds of methods.

- If you request a *reader*, Macintosh Common Lisp generates a method for reading the value of a slot but none for storing a value into it. Readers are used when the slot value won't change.
- If you request a *writer*, Macintosh Common Lisp generates a method for storing a value into a slot, but no method for reading its value.
- If you request an *accessor*, Macintosh Common Lisp generates two methods, a named method for reading the value and a `setf` method on the named method for storing a new value.

Using accessors to read and write the values of slots is preferable because accessor methods hide implementation detail. Your clients can call the accessor method without knowing whether it accesses a slot or computes a value in some other way. The implementation can later change without affecting the client's code.

Here is the `fourth-grader` class redefined using accessor methods. The accessor doesn't need to have the same name as the slot; here the accessor of the slot `teacher` is named `fourth-grade-teacher`:

```
? (defclass fourth-grader ()
    ((teacher :initarg :teacher
              :initform "Mrs. Marple"
              :accessor fourth-grade-teacher)
     (name :initarg :name
           :reader name)
     (school :initarg :school
             :initform "Lawrence School"
             :accessor school)
     (age :initarg :age
          :initform 9
          :accessor age)))
```

The `:reader` and `:accessor` methods simply provide a more abstracted alternative to `slot-value` and do not prevent your using `slot-value` as well. Creating the accessor method `fourth-grade-teacher` is equivalent to

```
? (defmethod fourth-grade-teacher ((student fourth-grader))
    (slot-value student 'teacher))

? (defmethod (setf fourth-grade-teacher)
             (new-teacher (student fourth-grader))
    (setf (slot-value student 'teacher ) new-teacher))
```

Making an instance of `fourth-grader` works the same way as previously.

```
? (setq delia (make-instance 'fourth-grader
                             :teacher "Mr. Smith"
                             :name "Delia"))
```

But you can now use the accessor method `fourth-grade-teacher` to get the `teacher` of `delia` and the accessor method `age` to get the age of `delia`:

```
? (fourth-grade-teacher delia)
"Mr. Smith"
? (age delia)
9
```

Accessor methods create regular generic functions and can be used just like any other Lisp functions. For instance, you can combine `fourth-grader` methods `say` and `school` to have an instance of `fourth-grader` report on the school it goes to. Because all look-ups occur at run time, this works even though `say` was defined before `fourth-grader` was redefined.

```
? (say mariah (school mariah))
***Lawrence School!***
NIL
? (say alisa (fourth-grade-teacher alisa))
***Mrs. Marple!***
I have an HP calculator
NIL
? (setf (age sharon) 10)
10
```

Customizing initialization with `initialize-instance`

Creating instances is not only a matter of associating values with slots; other initialization must often be performed. You specify what needs to be done by specializing the generic function `initialize-instance`. A method on `initialize-instance` for `fourth-grader` might look like the following:

```
(defmethod initialize-instance ((child fourth-grader) &rest initargs)
   (add-to-class-list child)
   (check-vaccinations child)
   (check-special-programs child))
```

When an instance of `fourth-grader` is created, `make-instance` calls `initialize-instance`, which calls the functions `add-to-class-list` and so on. These functions run their methods for `fourth-grader`. The functions that `make-instance` calls do not need to be generic functions and, if they are, do not need to have methods specifically for `fourth-grader`; they might have methods for one of the class's superclasses.

Creating subclasses and specializing their methods

When you create a subclass, you can write methods specific to that subclass.

```
? (defclass shy-kid (fourth-grader)())
#<STANDARD-CLASS SHY-KID>

? (defmethod say ((child shy-kid) sentence)
    (princ "...")
    (princ sentence)
    (princ "...")
    (terpri))
#<Method SAY (SHY-KID T)>

? (setq max (make-instance 'shy-kid))
#<SHY-KID #x609A69>
? (say max "Hi")
...Hi...
NIL
```

An instance of a subclass still inherits slot values and methods from its class's superclasses.

```
? (school max)
"Lawrence School"
```

Method combination

u *Note:* This and the following sections provide a very simplified summary of method combination. For full details see *Common Lisp: The Language*.

How does a function decide which method to use for a particular set of arguments?

There are two categories of methods. The first, the **primary method,** defines the main action of the effective method that the generic function applies to the instance.

The second category, **auxiliary methods,** may modify that action in one of three ways. An auxiliary method has the method qualifier `:before`, `:after`, or `:around`. They run before the primary method, after the primary method, or around the primary method and all of its `:before` and `:after` methods.

The primary method

The primary method applied to the instance is always the most specific method the generic function has for the arguments it has been given. When a generic function has a method for an instance, the instance's method is used. If not, CLOS looks up applicable methods for the instance's direct superclass, then for that superclass's superclass, and so on. As soon as one applicable method is found, the search stops and that method is applied to the instance. (If there is no applicable primary method, the function signals an error.)

The primary method and the class precedence list

It is important to know in what order to look up methods. When a class has many superclasses, behavior is determined by the ordered list of its superclasses—its class precedence list. The class precedence list is basic to method combination.

The order in which the superclasses are listed determines the order in which their primary methods are consulted. If the instance has a method, that method is used. If the class has a method, that is used. Otherwise the look-up consults the first (leftmost) superclass in the class precedence list, looking up the class's methods and those of the class's superclasses, leftmost superclass first. If there is no appropriate method anywhere in the first superclass or its superclasses, the look-up proceeds to the second superclass, and so on. The first applicable primary method that is found is used.

However, when elements of the class precedence list have superclasses in common, CLOS examines, in this order:

1. all of the elements of the first superclass except the class in common
2. all of the elements of the second superclass except the class in common
3. the class in common (followed by its superclasses)

Examples of classes with multiple superclasses

We have seen simple class precedence at work in earlier examples. Let us look at some examples of multiple superclasses.

For example, suppose two classes, `bored-kid` and `happy-kid`, both subclasses of `fourth-grader`, with instances `harvey` and `laura`. The bored kid has homework and the happy kid has a pet. Each subclass has a method for a generic function, `#'go-crazy`:

```
? (defclass bored-kid (fourth-grader)
    ((homework :initarg :homework
               :initform "Easy Steps to Calculus"
               :accessor homework)))
#<STANDARD-CLASS BORED-KID>

? (defmethod go-crazy ((child bored-kid) exclamation)
    (princ "I'm so bored!  ")
    (princ exclamation)
    (princ "!  What a dump!")
    (terpri))
#<Method GO-CRAZY (BORED-KID T)>

?  (setq harvey (make-instance 'bored-kid))
#<BORED-KID #x608879>
```

```
? (go-crazy harvey "Yukh")
I'm so bored!  Yukh!  What a dump!
NIL

? (defclass happy-kid (fourth-grader)
    ((pet :initarg :pet
          :initform "gorilla"
          :accessor pet)))
#<STANDARD-CLASS HAPPY-KID>

? (defmethod go-crazy ((child happy-kid) exclamation)
    (princ "How happy I am!  ")
    (princ exclamation)
    (princ "!  I'm going to shout and throw paper clips!")
    (terpri))
#<Method GO-CRAZY (HAPPY-KID T)>

? (setq laura (make-instance 'happy-kid))
#<HAPPY-KID #x608C51>
? (go-crazy laura "Wow")
How happy I am!  Wow!  I'm going to shout and throw paper clips!
NIL
```

When there is a conflict—Choosing between methods

When two classes inherit from the same superclasses, CLOS uses the class precedence list to determine which primary method to use. For example, we can define two new classes,`happy-kid-with-homework` and`bored-kid-with-pet` , that inherit from the same superclasses, but in opposite order.

```
? (defclass happy-kid-with-homework
    (happy-kid bored-kid)())
#<STANDARD-CLASS HAPPY-KID-WITH-HOMEWORK>
? (setq smiley (make-instance 'happy-kid-with-homework))
#<HAPPY-KID-WITH-HOMEWORK #x609C48>

?  (defclass bored-kid-with-pet (bored-kid happy-kid)())
#<STANDARD-CLASS BORED-KID-WITH-PET>
? (setq bad-max (make-instance 'bored-kid-with-pet))
#<BORED-KID-WITH-PET #x609C53>
```

To find the applicable method for `go-crazy`, CLOS looks for a method associated with the instance, then with the class, then with the first direct superclass in the list of the class's parents. The function `go-crazy` has an applicable method for the first parent, `happy-kid`. Therefore `happy-kid-with-homework` calls the `go-crazy` method associated with `happy-kid` as a primary method. It never calls the method for `bored-kid` because it doesn't get that far.

```
? (go-crazy smiley "Wow")
How happy I am!  Wow!  I'm going to shout and throw paper clips!
NIL
```

But `bored-kid-with-pet` calls the `bored-kid` method.

```
? (go-crazy bad-max "Ugh")
I'm so bored! Ugh! What a dump!
NIL
```

Choosing between methods associated with direct and with more distant superclasses

Suppose that `happy-kid` has a method for `#'say` but `bored-kid` doesn't:

```
? (defmethod say ((child happy-kid) sentence)
    (declare (ignore sentence))
    (princ "This primary method comes from happy-kid.")
    (terpri))
#<Method SAY (HAPPY-KID T)>
```

When CLOS looks for a method, it examines a class before its direct superclasses and a direct superclass before all other direct superclasses specified to its right in the list of the class's parents. Remember, if two of the direct superclasses of a class have a superclass in common, then the order is

1. all of the elements of the first superclass except the class in common
2. all of the elements of the second superclass except the class in common
3. the class in common (followed by its superclasses)

Within those constraints, the local ordering of the class precedence list is determined by taking all the elements of the class and doing a topological sort on them.

Here, `bored-kid` and `happy-kid` have a superclass in common, `fourth-grader`, so `bored-kid` and `happy-kid` both affect the behavior of `bored-kid-with-pet` before `fourth-grader` does:

```
? (say bad-max "I hate homework.")
This primary method comes from happy-kid.
NIL
```

Creating auxiliary methods and using method qualifiers

Only one primary method can be applied to any set of arguments, but its behavior can be modified using auxiliary methods.

All `:before` methods are run before the primary method. CLOS looks for `:before` methods in the same order as it looks for primary methods: instance first, then class, then direct superclasses. This order is called *most specific first* . After all applicable `:before` methods have run, Macintosh Common Lisp calls the primary method, then the `:after` methods in reverse order, from the `:after` methods associated with `t` all the way down to the class and instance `:after` methods if there are any. (This order is called *least specific first* .)

All `:around` methods (which are seldom used) specify code that is to be called instead of other applicable methods, but can pass control to other methods, including the primary, `:before` , and `:after` methods. The effect is that the `:around` method runs "around" the other methods. An `:around` method uses `call-next-method` to pass control to the other methods.

Here is an example of `:before` and `:after` methods.

```
? (defmethod say :before ((child bored-kid) sentence)
    (declare (ignore sentence))
    (princ "First there is a :before method from bored-kid.")
    (terpri))
#<Method SAY :BEFORE (BORED-KID T)>

? (defmethod say :after ((child bored-kid) sentence)
    (declare (ignore sentence))
    (princ "Then there is an :after method from bored-kid.")
    (terpri))
#<Method SAY :AFTER (BORED-KID T)>

? (defmethod say :around ((child bored-kid) sentence)
    (declare (ignore sentence))
    (princ "This illustrates method combination.")
    (terpri)
    (call-next-method))
#<Method SAY :AFTER (BORED-KID T)>

? (say bad-max "I hate homework.")
This illustrates method combination.
First there is a :before method from bored-kid.
```

```
This primary method comes from happy-kid.
Then there is an :after method from bored-kid.
NIL
```

When `say` is called on the instance `bad-max`, CLOS looks first for `:around` methods. Then it searches for `:before` methods, starting with the most specific, and finds and runs a `:before` method associated with `bored-kid`. It runs the applicable primary method, which is associated with `happy-kid`. Then it looks for `:after` methods, starting with the least specific, and finds the one associated with `bored-kid`, which it runs.

There can be multiple `:before` and `:after` methods, as in the following example:

```
? (defmethod say :after ((child fourth-grader) sentence)
    (declare (ignore sentence))
    (format t "This is an :after method for fourth-grader."))
#<Method SAY :AFTER (FOURTH-GRADER T)>

? (defmethod say :after ((child happy-kid) sentence)
    (declare (ignore sentence))
    (princ "Everybody likes my ~A. \(This is an :after method
for happy-kid\)" (pet child))
    (terpri))
#<Method SAY :AFTER (HAPPY-KID T)>

? (setq yoichi (make-instance 'bored-kid-with-pet
                              :pet "dog Lizzie"))
#<BORED-KID-WITH-PET #x554FA9>

? (say yoichi "Nobody ever listens to me.")
This illustrates method combination.
First there is a :before method from bored-kid.
This primary method comes from happy-kid.
This is an :after method for fourth-grader.
Everybody likes my dog Lizzie. (This is an :after method for happy-kid)
Then there is an :after method from bored-kid.
NIL
```

Combining all methods applicable to this instance produces the **effective method,** the complete list of primary and secondary methods.

Mixin classes and auxiliary methods

Mixins (not an official CLOS term) are ordinary CLOS classes with useful values or methods that "mix in" some special behavior. They are generally not used alone, but add their specializations to another class. In Macintosh Common Lisp, they usually appear first in the class precedence list. The class that adds Fred behavior to a window or dialog item, `fred-mixin`, is a mixin.

Extended example

This section shows a commented example of more realistic CLOS code than has been used in this appendix so far. This code creates a new subclass of window, `shape-window`, and a specialized scroller, `shape-scroller`. A shape window has a draw area and a shape-scroller area, which contains two buttons, titled Circles and Squares. Clicking inside the draw area causes the shape window to draw a shape, either a circle or a square. Clicking inside the shape-scroller area does nothing. Changing which radio button is pushed causes all the already drawn shapes to redraw themselves in the new selected shape.

The code also contains a new menu bar, which includes a menu that changes the color of the shapes.

u *Note:* If you are not running Macintosh Common Lisp in a color window, the shapes display as black.

This code is compared with its Object Lisp equivalent in Appendix D, "Converting Your Files to CLOS." It is available in the file `shapes_code.lisp` in your Examples folder.

Here is the extended example.

```
;;Load packages (no changes here unless the package changes):
(require 'scrollers)
(require 'quickdraw)

;;Specify global variables (no change here):
(defvar *shape-color* *black-color*)

;;Create new classes SHAPE-WINDOW and SHAPE-SCROLLER.
;;Class definitions have initialization arguments
;;and initial value forms:
```

```
(defclass shape-window (window)
  ((shape-list    :initarg :shape-list
                  :initform nil
                  :accessor shape-list)
   (current-shape :initarg :current-shape
                  :initform :circle
                  :accessor current-shape)
   (my-scroller   :initarg :my-scroller
                  :initform nil
                  :accessor my-scroller))
   (:default-initargs
                  :window-type :document-with-grow
                  :window-title "Shapes"))
(defclass shape-scroller (scroller)())

;;INITIALIZE-INSTANCE is a generic function that defines what happens
;;when instances are created.
;;It is specialized for each class.
;;Write a method for INITIALIZE-INSTANCE specifying what happens
;;during instance initialization of SHAPE-WINDOW.
;;Among other things, it creates an instance of SHAPE-SCROLLER.
;;SHAPE-SCROLLER doesn't do anything unusual during initialization
;;and simply uses its parent's method:

(defmethod initialize-instance ((window shape-window) &rest initargs)
  (apply #'call-next-method window initargs)
  (setf (my-scroller window)
        (make-instance 'shape-scroller
          :view-position #@(0 30)
          :view-size (subtract-points
                      (view-size window) #@(15 45))))
  (add-subviews window
                (make-instance 'radio-button-dialog-item
                  :view-position #@(100 10)
                  :dialog-item-text "Squares"
                  :dialog-item-action
                  #'(lambda (item)
                      item
                      (setf (current-shape
                             (view-container item)) :square)
                      (clean-slate (view-container item))))
                (make-instance 'radio-button-dialog-item
                  :view-position #@(10 10)
```

```
                  :dialog-item-text "Circles"
                  :dialog-item-action
                  #'(lambda (item)
                      item
                      (setf (current-shape
                             (view-container item)) :circle)
                      (clean-slate (view-container item)))
                  :radio-button-pushed-p t)
                (my-scroller window)))

;;Write other methods for SHAPE-WINDOW and SHAPE-SCROLLER
;;on the generic functions CLEAN-SLATE, SET-VIEW-SIZE,
;;VIEW-CLICK-EVENT-HANDLER and VIEW-DRAW-CONTENTS:

(defmethod clean-slate ((window shape-window))
  (with-focused-view window
    (erase-rect window (view-scroll-position window)
                (add-points (view-scroll-position window)
                            (view-size window)))
    (view-draw-contents window)))

(defmethod set-view-size ((window shape-window) h &optional v)
  (call-next-method)
  (set-view-size (my-scroller window)
                 (subtract-points (make-point h v) #@(15 45))))

(defmethod view-click-event-handler
           ((scroller shape-scroller) position)
  (let ((my-window (view-container scroller)))
    (setf (slot-value  my-window 'shape-list)
          (cons position (shape-list my-window)))
    (view-draw-contents scroller)))

(defmethod view-draw-contents ((scroller shape-scroller))
  (let ((my-window (view-container scroller)))
    (with-focused-view scroller
      (with-fore-color *shape-color*
        (dolist (s (shape-list my-window))
          (if (eq (current-shape my-window) :circle)
            (frame-oval my-window s (add-points s #@(30 30)))
            (frame-rect my-window s (add-points s #@(30 30)))))))))

;;Create a menu bar and make it the active menu bar:

  (set-menubar (list (car (menubar))
                  (make-instance 'menu
                     :menu-title "File"
                     :menu-items
```

```
                    (list (make-instance 'menu-item
                            :menu-item-title "New Shape Window"
                            :menu-item-action
                              (nfunction
                               menu-item-action
                               (lambda nil
                                 (make-instance 'shape-window)))
                            :command-key #\N)
                          (make-instance 'menu-item
                            :menu-item-title "Open"
                            :menu-item-action 'choose-file-dialog
                            :command-key #\O)
                          (make-instance 'menu-item
                            :menu-item-title "Quit"
                            :menu-item-action 'quit)))
                (make-instance 'menu
                   :menu-title "Edit"
                   :menu-items
                     (list (make-instance 'menu-item
                             :menu-item-title "Undo"
                             :disabled t
                             :command-key #\Z
                             :menu-item-action 'undo)
                           (make-instance 'menu-item
                              :menu-item-title "-"
                              :disabled t)
                           (make-instance 'window-menu-item
                              :menu-item-title "Cut"
                              :command-key #\X
                              :menu-item-action 'cut)
                           (make-instance 'window-menu-item
                              :menu-item-title "Copy"
                              :command-key #\C
                              :menu-item-action 'copy)
                           (make-instance 'window-menu-item
                              :menu-item-title "Paste"
                              :command-key #\V
                              :menu-item-action 'paste)
                           (make-instance 'window-menu-item
                              :menu-item-title "Clear"
                              :menu-item-action 'clear)))
;;This new menu changes the colors of the shapes
;;drawn in SHAPE-WINDOW:
```

```
                (make-instance 'menu
                   :menu-title "Colors"
                   :menu-items
                     (list (make-instance 'menu-item
                              :menu-item-title "Black"
                              :menu-item-action
                                (nfunction
                                 menu-item-action (lambda nil
                                    (setq *shape-color*
                                     *black-color*))))
                           (make-instance 'menu-item
                              :menu-item-title "Blue"
                              :menu-item-action
                                (nfunction
                                 menu-item-action (lambda nil
                                    (setq *shape-color*
                                     *blue-color*))))
                           (make-instance 'menu-item
                              :menu-item-title "Red"
                              :menu-item-action
                                (nfunction
                                 menu-item-action (lambda nil
                                    (setq *shape-color*
                                     *red-color*))))
                           (make-instance 'menu-item
                              :menu-item-title "Green"
                              :menu-item-action
                                (nfunction
                                 menu-item-action (lambda nil
                                    (setq *shape-color*
                                     *green-color*)))))))

;;Set the menu bar:
(set-menubar littlebar)

;;Select New Shape Window from the File menu to open a new shape
;;window.
```

Appendix D **Converting Your Files to CLOS**

Contents

General procedure for converting files to CLOS / 686
Turning `defobject` and `exist` pairs into `defclass` and `initialize-instance` pairs / 686
Turning occurrences of `defobfun` into `defmethod` / 687
Replacing free variable references / 688
Removing calls to `ask` / 689
Replacing `oneof` with `make-instance` and `kindof` with `defclass` / 689
Changing function names / 690
An extended example / 691
Calling packages and setting variables / 692
Turning calls to `defobject` and `exist` into `defclass` and `initialize-instance` / 694
Turning other calls to `defobfun` into `defmethod` / 698
Removing `ask` and replacing `oneof` by `make-instance` / 700
Making a `window` instance / 704

This appendix describes how to convert your Macintosh Common Lisp version 1.3 Object Lisp code to the Common Lisp Object System under Macintosh Common Lisp version 2.

General procedure for converting files to CLOS

To transform Object Lisp code to CLOS code, go through five steps:

- Turn your `defobject` and `exist` pairs into `defclass` and `initialize-instance` pairs.
- Turn all occurrences of `defobfun` into corresponding occurrences of `defmethod`.
- Transform all references to free variables into calls to `slot-value` or to accessors.
- Get rid of calls to `ask`.
- Replace `oneof` with `make-instance` and transform `kindof` to `defclass`.

Turning `defobject` and `exist` pairs into `defclass` and `initialize-instance` pairs

In Macintosh Common Lisp 1.3, `defobject` generally contains the class hierarchy and `exist` contains a group of uses of `have`, with their default values. Most frequently, the corresponding material in CLOS goes into `defclass` with its slot specifiers.

The `initialize-instance` procedure specifies the values of slots that can't be initialized with initialization arguments or initial forms. It also performs any other necessary initialization.

If you use `have` dynamically, you can deal with it in one of two ways. You can either

- transform uses of `have` into static slots and values in `defclass`

or

- put an association list as one of the slots in the `defclass`, then redefine `have` to put values on that association list and search for them

You can find an example of the second way of using `have` in the example file `dynamic-slots.lisp`. However, it is more efficient to call values explicitly, and in most cases that is recommended.

Here's an example of old and new versions of code. Here is the Object Lisp version.

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

```
? (ask fourth-grader (have 'teacher "Mrs. Jones")
                       (have 'school "Lawrence School"))
NIL
```

And here is the new CLOS version.

```
? (defclass fourth-grader ()
      ((teacher :initarg :teacher
            :initform "Mrs. Jones"
            :accessor teacher)
       (school :initarg 'school
            :initform "Lawrence School"
            :accessor school)))
 #<STANDARD-CLASS FOURTH-GRADER>
```

Turning occurrences of `defobfun` into `defmethod`

You must turn all of your object functions into generic functions. That is, every piece of code of the form

(defobfun (*foo* editable-text-dialog-item) *arguments body*)

must become

(defmethod *foo* ((*item* editable-text-dialog-item) *rest-of-args*) *body*)

You can find a piece of code that performs most of this transformation for you in the file `defobfun-to-defmethod.lisp` in your Examples folder.

Here is an example of old and new versions of code. Here is the old Object Lisp code.

```
? (defobfun (say fourth-grader) (sentence)
      (princ "***")
      (princ sentence)
      (princ "***")
      (terpri))
SAY
```

Here is the new CLOS code.

```
? (defmethod say ((child fourth-grader) sentence)
  (princ "***")
  (princ sentence)
  (princ "!***")
  (terpri))
SAY
```

Replacing free variable references

Slot values and variables don't access the same namespace, so you must bring all free variable references into the slot namespace. In your version 1.3 code most of these free variables should already be declared.

Again, you have a choice of ways of dealing with free variables. You can either explicitly set them with `slot-value` or use accessor calls.

Accessors usually are preferable, since they allow you to change the representation of your classes without changing any of the user code.

Here's an example of old and new versions of code. First is the Old Object Lisp code.

```
? (setq school "Bronx Science")
"Bronx Science"
? school
"Bronx Science"
? (ask fourth-grader school)
"Bronx Science"
? (ask fourth-grader (have 'school))
NIL
? (ask fourth-grader (setq school "MIT"))
"MIT"
? school
"Bronx Science"
```

The behavior of CLOS is different.

```
? (setq school "Bronx Science")
"Bronx Science"
? school
"Bronx Science"
? (school john)
"Lawrence School"
? (setq braino (make-instance 'fourth-grader
                 :name "Braino"
                 :school "MIT"))
#<FOURTH-GRADER #x60F7C1>
? (school braino)
 "MIT"
```

Removing calls to `ask`

Instead of calling `ask`, you now use `slot-value` or an accessor for a value, or call a method. Depending on the context, you should choose the appropriate thing to do.

Remember that you run methods on instances of a class. All calls must now be to instances, not to class objects. An Object Lisp `ask` for the object function `say` `()`

```
? (ask billy-crystal (say "Marvelous"))
```

becomes a call to the appropriate method of `say` for an instance of `fourth-grader`, `billy-crystal`:

```
? (say billy-crystal "Marvelous")
***Marvelous!***
NIL
```

Replacing `oneof` with `make-instance` and `kindof` with `defclass`

Rather than creating all new objects with `oneof`, you create instances with `make-instance`.

Similarly, you replace `kindof` with `defclass`. But note that in CLOS, you cannot make instances directly into classes. Instead you make new classes based on the class you want to specialize and then create instances of those classes.

Here is an example of `oneof` and `kindof` in Object Lisp.

```
? (setq fastbob (oneof fourth-grader))
#<Object #118, a #117>
? (ask fastbob
    (have 'vpp-computer "Quadra with leash and wheels"))
"Quadra with leash and wheels"
? (setq harvey (kindof fastbob))
#<Object #119, a #118>
```

Here is the corresponding code in CLOS. To create an instance, use `make-instance`.

```
? (setq fastbob (make-instance 'fourth-grader))
#<FOURTH-GRADER #x4951B9>
```

However, to give slots to a class, you must define a class.

```
? (defclass virtual-fastbob (fourth-grader)
    ((very-powerful-portable-computer
      :initarg :vpp-computer
      :initform "Quadra with leash and wheels")))
#<STANDARD-CLASS VIRTUAL-FASTBOB>
? (setq harvey (make-instance 'virtual-fastbob))
#<VIRTUAL-FASTBOB #x495ED9>
```

You can also put the new slot into a mixin.

```
? (defclass portable-power-user-mixin ()
    ((very-powerful-portable-computer
      :initarg :vpp-computer
      :initform "Quadra with leash and wheels")))
#<STANDARD-CLASS PORTABLE-POWER-USER-MIXIN>
? (defclass virtual-fastbob
    (portable-power-user-mixin fourth-grader)())
#<STANDARD-CLASS VIRTUAL-FASTBOB>
```

Note that if `vpp-computer` existed as a free variable in your version 1.3 code, instances in CLOS will no longer find it. See the section “Replacing Free Variable References” earlier in this appendix.

Changing function names

Numerous function names and other symbols have changed in both Common Lisp and Macintosh Common Lisp. To investigate name changes, see this manual or use Apropos on the Tools menu. Functions involving dialog items or windows are frequently affected. Name changes are also documented in the file `old-dialog-hooks.lisp` in the MCL Examples folder.

An extended example

The following code provides an extended example of conversion from Object Lisp to CLOS. This code creates a new type of window, `shape-window`. A shape window has a draw area and a `shape-scroller` area containing two buttons, marked Circles and Squares. Clicking inside the draw area causes the shape window to draw a shape, either a circle or a square. Changing which radio button is pushed causes all the already drawn shapes to redraw themselves in the new selected shape.

The code also contains a new menu bar, which includes a menu that changes the color of the shapes.

u *Note:* If you are not running Macintosh Common Lisp in a color window, the shapes display as black.

In the following pages, Object Lisp code appears on the left pages and CLOS code on the right. In these examples only the code is given; Macintosh Common Lisp's response does not appear.

This code conversion includes several sections:

- calling packages and setting variables
- turning `defobject` and `exist` pairs into `defclass` and `initialize-instance` pairs
- turning other calls to `defobfun` into `defmethod`
- removing calls to `ask`
- turning calls to `oneof` into `make-instance`
- making an instance of the window

This code also appears in the file `shapes_code.lisp` in your Examples folder.

Calling packages and setting variables

The package syntax is the same as in Macintosh Common Lisp version 1.3. However, you should check that the functionality you want is still in the same package. For instance, older Common Lisp code uses the package name `lisp`, but Macintosh Common Lisp version 2 uses `common-lisp`, as required by the proposed new standard.

The variable syntax is unchanged from version 1.3.

Here is the Object Lisp version of the code.

```
()
(in-package :cl-user)
(require 'scrollers)
(require 'quickdraw)
(defvar *shape-color* *black-color*)
(defvar *user-menubar* (menubar))
```

...in CLOS

Here is the corresponding CLOS code.

```
()
(in-package :cl-user)
(require 'scrollers)
(require 'quickdraw)
(defvar *shape-color* *black-color*)
(defvar *user-menubar* (menubar))
```

Turning calls to `defobject` and `exist` into `defclass` and `initialize-instance`

When you defined objects in Object Lisp, you specified their properties using `have` within `defobfun`. In CLOS you specify slot values within `defclass`. You can specify these values only if they can be initialized with initialization arguments or initial value forms.

Procedures to be performed during initialization are specified within the class method for `initialize-instance`. Here is the old Object Lisp code for `defobject` and `exist`.

```
(defobject *shape-window* *dialog*)
(defobject *shape-scroller* ccl::*scroller*)
```

The `defobfun` procedure initializes all values.

```
(defobfun (exist *shape-window*) (init-list)
  (let ((me (self)))
    (have 'shape-list '())
    (have 'current-shape :circle)
    (usual-exist (init-list-default init-list
     :window-type :document-with-grow
     :window-title "shapes"
     :dialog-items
         (list (make-dialog-item *radio-button-dialog-item*
                  #@(97 5)
                  #@(72 16)
                  "Squares"
                  (nfunction dialog-item-action
                    (lambda nil
                      (ask (objvar my-dialog)
                       (setf (objvar current-shape) :square)
                       (clean-slate))
                       (usual-dialog-item-action))))
```

(continued next left-hand page)

Here is the corresponding CLOS code, using `defclass` and `initialize-instance`. Note that the class `shape-scroller` doesn't need its own `initialize-instance`; it inherits its parent's method.

The class definitions specify only slot values.

```
(defclass shape-window (window)
  ((shape-list    :initarg :shape-list
                  :initform nil
                  :accessor shape-list)
   (current-shape :initarg :current-shape
                  :initform :circle
                  :accessor current-shape)
   (my-scroller   :initarg :my-scroller
                  :initform nil
                  :accessor my-scroller))
   (:default-initargs
                  :window-type :document-with-grow
                  :window-title "Shapes"))

(defclass shape-scroller (ccl::scroller)())
```

The `initialize-instance` method for `shape-window` specifies all procedures that are carried out during initialization.

```
(defmethod initialize-instance ((window shape-window) &key)
  (call-next-method)
  (setf (my-scroller window)
    (make-instance 'shape-scroller
        :view-position #@(0 30)
        :view-size (subtract-points
                     (view-size window) #@(15 45))))
```

(continued next right-hand page)

```
            (make-dialog-item *radio-button-dialog-item*
              #@(11 5)
              #@(72 16)
              "Circles"
              (nfunction dialog-item-action
               (lambda nil
                (ask (objvar my-dialog)
                 (setf (objvar current-shape) :circle)
                 (clean-slate))
                 (usual-dialog-item-action)))
              :radio-button-pushed-p t))))
(have 'my-scroller (oneof *shape-scroller*
                     :view-container me
                     :view-position #@(0 30)
                     :view-size
                       (subtract-points (ask me (window-size))
                                        #@(15 45))))))
```

```
(add-subviews window
   (make-instance 'radio-button-dialog-item
      :view-position #@(100 10)
      :dialog-item-text "Squares"
      :dialog-item-action
        #'(lambda (item)
         item
         (setf (current-shape (view-container item)) :square)
         (clean-slate (view-container item))))
   (make-instance 'radio-button-dialog-item
      :view-position #@(10 10)
      :dialog-item-text "Circles"
      :dialog-item-action
        #'(lambda (item)
          item
          (setf (current-shape (view-container item)) :circle)
          (clean-slate (view-container item)))
       :radio-button-pushed-p t)
   (my-scroller window)))
```

Turning other calls to `defobfun` into `defmethod`

Here are some examples of turning Object Lisp `defobfun` calls into `defmethod` calls. These functions define the behavior of the shape window and shape scroller.

```
(defobfun (clean-slate *shape-window*) ()
  (ask (objvar my-scroller)
    (with-focused-view (self)
      (erase-rect (view-scroll-position)
                  (add-points (view-scroll-position) (view-size)))
      (view-draw-contents))))

(defobfun (set-window-size *shape-window*) (h &optional v)
  (usual-set-window-size h v)
  (ask (objvar my-scroller)
    (set-view-size (subtract-points (make-point h v)
                                    #@(15 45)))))

(defobfun (view-click-event-handler *shape-scroller*) (position)
  (ask (view-container)
    (setf (objvar shape-list)
          (cons position (objvar shape-list))))
  (view-draw-contents))

(defobfun (view-draw-contents *shape-scroller*) ()
  (usual-view-draw-contents)
  (with-focused-view (self)
  (with-fore-color *shape-color*
    (dolist (shape (ask (view-container) (objvar shape-list)))
      (if (ask (view-container)
            (eq (objvar current-shape) :circle))
          (frame-oval shape (add-points shape #@(30 30)))
          (frame-rect shape (add-points shape #@(30 30))))))))
```

Here is the corresponding CLOS code. All methods of generic functions act on a class, which is specified as the first argument. Note that some function names have changed in Macintosh Common Lisp because of the integration of views, windows, and dialogs; for instance, `set-window-size` is now `set-view-size` with a method for windows.

```
(defmethod clean-slate ((window shape-window))
  (with-focused-view window
    (let ((scroll-position (view-scroll-position window)))
      (rlet ((rect :rect
                   :topLeft scroll-position
                   :botRight
                     (add-points scroll-position (view-size window))))
        (#_EraseRect rect)))
  (view-draw-contents window)))

(defmethod set-view-size ((window shape-window) h &optional v)
  (call-next-method)
  (set-view-size (my-scroller window)
      (subtract-points (make-point h v) #@(15 45))))

(defmethod view-click-event-handler
        ((scroller shape-scroller) position)
  (let ((my-window (view-container scroller)))
    (setf (slot-value  my-window 'shape-list)
              (cons position (shape-list my-window)))
  (view-draw-contents scroller)))

(defmethod view-draw-contents ((scroller shape-scroller))
  (let ((my-window (view-container scroller)))
    (with-focused-view scroller
      (with-fore-color *shape-color*
        (dolist (s (shape-list my-window))
          (rlet ((rect :rect :topLeft s
                             :botRight (add-points s #@(30 30))))
            (if (eq (current-shape my-window) :circle)
              (#_FrameOval rect)
              (#_FrameRect rect)))))))))
```

Removing `ask` and replacing `oneof` by `make-instance`

The following code defines a new menu bar and sets it to be the active menu bar. This conversion is almost automatic. You only need to remove the call to `ask` and replace `oneof` with `make-instance`.

Here is the Object Lisp code.

```
(progn
  (ask *apple-menu*
    (apply #'remove-menu-items (menu-items))
    (apply #'add-menu-items (list)))
  (set-menubar (list
                 (oneof *menu*
                    :menu-title "File"
                    :menu-items
                      (list (oneof *menu-item*
                              :menu-item-title "New Shape Window"
                              :menu-item-action
                                (nfunction
                                 menu-item-action
                                 (lambda nil (oneof *shape-window*)))
                              :command-key #\N)))
                 (oneof *menu*
                    :menu-title "Edit"
                    :menu-items
                      (list (oneof *window-menu-item*
                              :menu-item-title "Undo"
                              :disabled t
                              :command-key #\Z
                              :menu-item-action 'undo)
                            (oneof *menu-item*
                               :menu-item-title "-"
                               :disabled t)
                            (oneof *window-menu-item*
                               :menu-item-title "Cut"
                               :command-key #\X
                               :menu-item-action 'cut)
```

(continued next left-hand page)

Here is the corresponding CLOS code.

```
(defparameter *littlebar*
  (list (car *default-menubar*)
        (make-instance 'menu
                       :menu-title "File"
                       :menu-items
                         (list (make-instance 'menu-item
                                :menu-item-title
                                  "New Shape Window"
                                :menu-item-action
                                  #'(lambda nil
                                      (make-instance 'shape-window))
                                         :command-key #\N)
                               (make-instance 'menu-item
                                :menu-item-title "Open"
                                :menu-item-action
                                  'choose-file-dialog
                                :command-key #\O)
                               (make-instance 'menu-item
                                :menu-item-title "Quit"
                                :menu-item-action 'quit)))
        (make-instance 'menu
                       :menu-title "Edit"
                       :menu-items
                         (list (make-instance 'menu-item
                                :menu-item-title "Undo"
                                :disabled t
                                :command-key #\Z
                                :menu-item-action 'undo)
                               (make-instance 'menu-item
                                 :menu-item-title "-"
                                 :disabled t)
                               (make-instance 'window-menu-item
                                 :menu-item-title "Cut"
                                 :command-key #\X
                                 :menu-item-action 'cut)
```

(continued next right-hand page)

```
            (oneof *window-menu-item*
               :menu-item-title "Copy"
               :command-key #\C
               :menu-item-action 'copy)
            (oneof *window-menu-item*
               :menu-item-title "Paste"
               :command-key #\V
               :menu-item-action 'paste)
            (oneof *window-menu-item*
               :menu-item-title "Clear"
               :menu-item-action 'clear)))
(oneof *menu*
   :menu-title "Colors"
   :menu-items
     (list (oneof *menu-item*
               :menu-item-title "Black"
               :menu-item-action
                 (nfunction
                  menu-item-action (lambda nil
                     (setq *shape-color*
                      *black-color*))))
            (oneof *menu-item*
               :menu-item-title "Blue"
               :menu-item-action
                 (nfunction
                  menu-item-action (lambda nil
                     (setq *shape-color*
                      *blue-color*))))
            (oneof *menu-item*
               :menu-item-title "Red"
               :menu-item-action
                 (nfunction
                  menu-item-action (lambda nil
                     (setq *shape-color*
                      *red-color*))))
            (oneof *menu-item*
               :menu-item-title "Green"
               :menu-item-action
                 (nfunction
                  menu-item-action (lambda nil
                     (setq *shape-color*
                      *green-color*)))))))))
```

```
                        (make-instance 'window-menu-item
                            :menu-item-title "Copy"
                            :command-key #\C
                            :menu-item-action 'copy)
                        (make-instance 'window-menu-item
                            :menu-item-title "Paste"
                            :command-key #\V
                            :menu-item-action 'paste)
                        (make-instance 'window-menu-item
                            :menu-item-title "Clear"
                            :menu-item-action 'clear)
                        (make-instance 'menu-item
                            :menu-item-title "-"
                            :disabled t)
                        (make-instance 'menu-item
                            :menu-item-title "Original Menubar"
                            :menu-item-action #'(lambda ()
                              (set-menubar *user-menubar*)))))
  (make-instance 'menu :menu-title "Colors"
                 :menu-items
                 (list (make-instance 'menu-item
                          :menu-item-title "Black"
                          :menu-item-action
                           #'(lambda nil
                               (setq *shape-color* *black-color*)))
                       (make-instance 'menu-item
                          :menu-item-title "Blue"
                          :menu-item-action
                           #'(lambda nil
                               (setq *shape-color* *blue-color*)))
                       (make-instance 'menu-item
                          :menu-item-title "Red"
                          :menu-item-action
                           #'(lambda nil
                               (setq *shape-color* *red-color*)))
                       (make-instance 'menu-item
                          :menu-item-title "Green"
                          :menu-item-action
                           #'(lambda nil
                              (setq *shape-color*
                                    *green-color*)))))))
(set-menubar *littlebar*)
```

Making a `window` instance

Again, the code for creating instances is simply `make-instance` rather than `oneof`.

Here is the Object Lisp version.

```
(oneof *shape-window*)
```

The corresponding CLOS code is the action performed by the New Shape Window menu item.

```
(make-instance 'shape-window)
```

Note that you cannot create a subclass from an instance. You must define a new class with `defclass`. See the section "Replacing `oneof` With `make-instance` and `kindof` With `defclass`" earlier in this appendix.

Appendix E **QuickDraw Graphics**

Contents

QuickDraw in Macintosh Common Lisp / 707
Windows, GrafPorts, and PortRects / 707
Points and rectangles / 708
Window state functions / 710
Pen and line-drawing routines / 712
Drawing text / 720
Calculations with rectangles / 721
Graphics operations on rectangles / 726
Graphics operations on ovals / 730
Graphics operations on rounded rectangles / 733
Graphics operations on arcs / 737
Regions / 741
    Calculations with regions / 744
    Graphics operations on regions / 748
Bitmaps / 749
Pictures / 751
Polygons / 753
Miscellaneous procedures / 757

This chapter documents a set of CLOS methods that create an interface with QuickDraw. The code that implements these functions serves as an extended example of CLOS programming and is included as an example file. You should read it if you plan to use QuickDraw extensively in Macintosh Common Lisp, or if you are planning to create your own high-level methods to interface with traps. However, you may prefer to use the traps functionality documented in Chapter 17, “Higher-Level Operating System Interface.”

This appendix assumes some familiarity with the various discussions of QuickDraw in *Inside Macintosh*. You should also be familiar with the "Points" section of Chapter 3, "Points and Fonts," and with the MCL implementation of records, described in Chapter 17, "Higher-Level Operating System Interface."

QuickDraw in Macintosh Common Lisp

Macintosh Common Lisp version 2 allows you to call QuickDraw traps directly (see Chapter 17, “Higher-Level Operating System Interface”). The interface routines support all of the functionality found in the original (64K ROM) Macintosh packages.

The arguments to the MCL QuickDraw functions generally parallel the arguments to the Pascal QuickDraw functions given in *Inside Macintosh* . In several places Pascal functionality has been extended by taking advantage of the optional arguments provided by Macintosh Common Lisp. In some places the order of arguments has been changed to make the mapping of the optional arguments more effective. Last *var* arguments have sometimes been eliminated, and instead a value is returned.

Some QuickDraw functions may be performed only as methods on views. The view must be a window or must be contained in a window. The functions depend on the existence of a graphics port (GrafPort). All other functions may be performed globally.

Before calling any of the functions described in this chapter, you must load the QuickDraw file, which is in your MCL Library directory.

Windows, GrafPorts, and PortRects

All drawing on the Macintosh computer takes place inside **GrafPorts,** the structures upon which a program builds windows. (See *Inside Macintosh* for a complete description of GrafPorts.)

In low-level Macintosh drawing, several levels of initialization are used to set up windows and GrafPorts for drawings. Once they have been created, you must keep track of the current GrafPort when you do any drawing.

This process is simplified for the graphics routines given in this appendix.

- When you create a window, an initialized GrafPort is automatically created.
- Drawing commands are defined as methods on views, which must be windows or contained in windows; when you call a method to perform one of the commands in a window, GrafPorts are set and restored automatically with `with-focused-view`.

- Drawing inside windows is automatically cropped to fit inside the window and the portions of the window that are visible (that is, not covered by other windows).

Drawing is also affected by the clip region (described later) and the PortRect. The **PortRect** is an arbitrary rectangle designating the outermost bounds in which drawing can occur. (See Figure E-2.) It supplies a frame of reference for the window. The default PortRect is infinitely large; you can set it using low-level calls (although you usually won't need to worry about this at all).

Since all the drawing functions use `with-focused-view`, you can speed up drawing considerably if you wrap `with-focused-view` around all calls to multiple drawing functions.

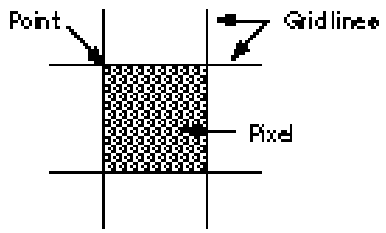
Points and rectangles

In QuickDraw, points are specified by two coordinates, the horizontal coordinate (called *h*) and the vertical coordinate (called *v*). The horizontal coordinate increases as it moves to the right, and the vertical coordinate increases as it moves down. The upper-left corner of a window (called the *origin*) is usually the point (0,0), but the origin may be changed by using the `set-origin` generic function.

Points are stored as encoded integers. Points lie at the intersection of two grid lines on the QuickDraw plane. Note that points and pixels are not equivalent. The point associated with a given pixel is at the upper-left corner of the pixel. (See Figure E-1.)

See the "Points" section of Chapter 3, "Points and Fonts," for a general description of the MCL point data format.

- **Figure E-1** Location of point at upper-left corner of pixel

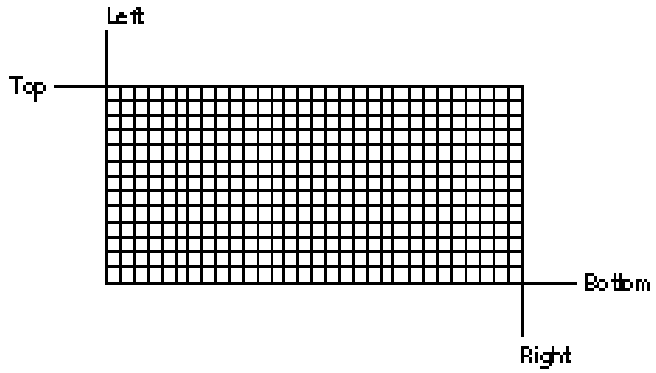


A Macintosh computer stores rectangles as 8-byte records. (Records are blocks of non-Lisp data stored on the Macintosh heap or on the stack; see Chapter 17, "Higher-Level Operating System Interface," for details.)

Rectangle records can be thought of as two points (upper-left and lower-right), or four edges (left, top, right, and bottom). Allocating memory for rectangle records can be inefficient, and so Macintosh Common Lisp provides several forms of memory allocation. The `make-record` function is used to allocate memory for long-lived rectangles, and the `rlet` function is used to allocate records for short-lived rectangles (see Chapter 17, "Higher-Level Operating System Interface," for details).

For many of the MCL QuickDraw functions that use rectangles, you do not need to allocate rectangle records explicitly at all. The rectangles can be specified as four coordinates, or as two points, or as a rectangle record (see Figure E-2). In general, if you use a rectangle only once, it is all right to pass it as two points or four coordinates. However, if you use it several times, it is more efficient to create and pass an actual rectangle record.

Figure E-2 A PortRect



When alternative forms of a point or a rectangle are accepted as arguments, the flexible argument appears last. This order prevents ambiguity about which argument is which and explains why the order of arguments sometimes differs from the order given in *Inside Macintosh* .

Window state functions

The following functions operate on the window containing the view asked to perform a function.

origin [*Generic function*]

Syntax `origin` (*view* `view`)

Description The `origin` generic function returns the coordinates of the upper-left point in the window's content region. This is usually `#@(0 0)` but may be different if it is set by user-written code.

Argument

view	A window or a view contained in a window.

set-origin [*Generic function*]

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

Description The `set-origin` generic function sets the origin to the point specified by *h* and *v*.

The contents of the window are not moved; only future drawing is affected.

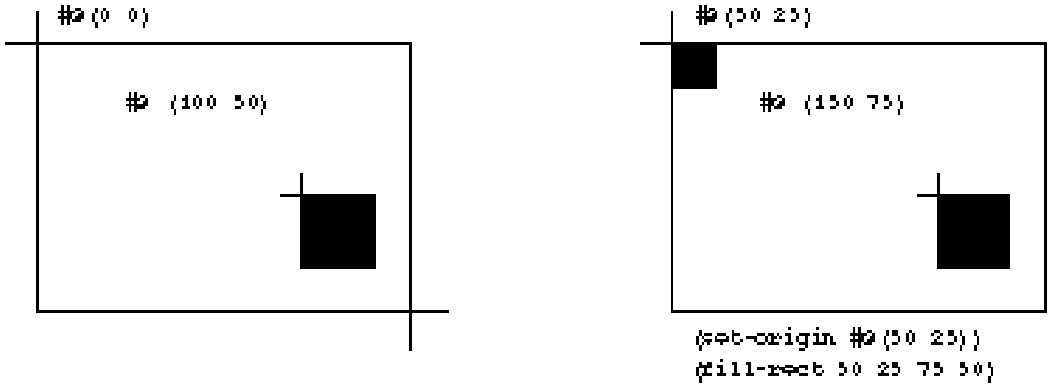
Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If the value of *v* is `nil` (the default), *h* is assumed to represent a point.

Example

Figure E-3 shows an example of using `set-origin` . In that figure you can see that rectangles can be passed as two points, four coordinates, or one rectangle record.

n **Figure E-3** Multiple methods of passing rectangles



A clip region allows drawing in a window to be restricted to an arbitrary region. Drawing occurs only in the clip region. The default clip region is arbitrarily large, so no clipping takes place. Note that regions must be explicitly disposed of; they are not subject to automatic garbage collection.

clip-region

[*Generic function*]

Syntax `clip-region` (*view* `view`) `&optional` *save-region*

Description The `clip-region` generic function returns the window's current clip region.

Arguments

view	A window or a view contained in a window.
save-region	The region in which the window's clip region is returned; otherwise, the clip region is returned in a newly allocated region.

set-clip-region

[*Generic function*]

Syntax `set-clip-region` (*view* `view`) *new-region*

Description The `set-clip-region` generic function sets the window's clip region to *new-region* and returns *new-region* .

Arguments

view	A window or a view contained in a window.
new-region	A region.

See the section "Regions" later in this appendix for functions that allocate and manipulate regions.

clip-rect

[*Generic function*]

Syntax `clip-rect` (*view* `view`) *left* `&optional` *top right bottom*

Description The `clip-rect` generic function makes the window's clip region a rectangular region equivalent to the rectangle determined by *arg*. It returns `nil`.

Arguments

view A window or a view contained in a window.

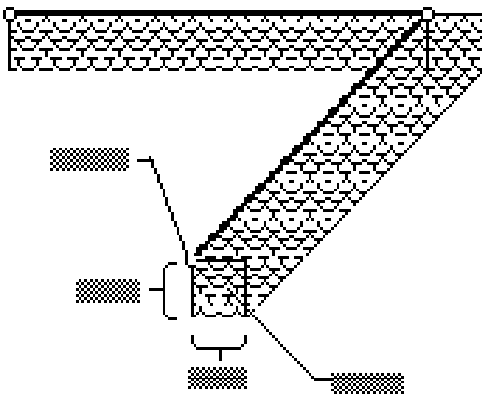
left , *top, right, bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Pen and line-drawing routines

Every window has its own pen. The state of the pen determines how drawing occurs in the window. For example, if the pen is hidden, drawing commands have no effect on the screen. In addition to its state as hidden or shown, a pen has a size (height and width), a position in the window, and a pattern used for drawing. (See Figure E-4.)

Figure E-4 Attributes of a graphics pen



The following functions operate on the window containing the view asked to perform a function.

pen-show

[*Generic function*]

Syntax `pen-show` (*view* `view`)

Description The `pen-show` generic function shows the pen. Drawing occurs only when the pen is shown.

Argument *view* A window or a view contained in a window.

pen-hide

[*Generic function*]

Syntax `pen-hide` (*view* `view`)

Description The `pen-hide` generic function hides the pen. If the pen is hidden, no drawing occurs.

Argument *view* A window or a view contained in a window.

pen-shown-p

[*Generic function*]

Syntax `pen-shown-p` (*view* `view`)

Description The `pen-shown-p` generic function returns `t` if the pen is shown and `nil` if the pen is hidden.

Argument *view* A window or a view contained in a window.

pen-position

[*Generic function*]

Syntax `pen-position` (*view* `view`)

Description The `pen-position` generic function returns a point corresponding to the pen position in local coordinates.

Argument *view* A window or a view contained in a window.

pen-size

[*Generic function*]

Syntax `pen-size` (*view* `view`)

Description The `pen-size` generic function returns the current pen size as a point (expressing a width and height).

Argument *view* A window or a view contained in a window.

set-pen-size

[*Generic function*]

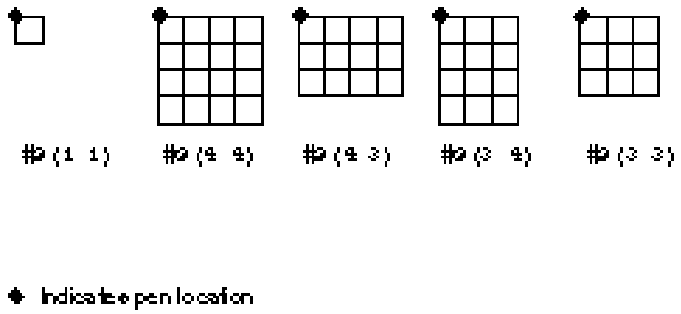
Syntax `set-pen-size` (*view* `view`) *h* `&optional` *v*

Description The `set-pen-size` generic function sets the pen size to the point indicated by *h* and *v*. Figure E-5 shows QuickDraw pen sizes.

Arguments

view	A window or a view contained in a window.
h	The width of the new pen size (or a point representing the width and height, if *v* is not given).
v	The height of the new pen size.

Figure E-5 QuickDraw pen sizes



pen-pattern

[*Generic function*]

Syntax `pen-pattern` (*view* `view`)`&optional` *save-pattern*

Description The `pen-pattern` generic function returns the window's pen pattern.

Arguments

view	A window or a view contained in a window.
save-pattern	A pattern record; the pattern is returned in this record. If *save-pattern* is not given, a new pattern record is allocated to hold the returned pattern.

set-pen-pattern

[*Generic function*]

Syntax `set-pen-pattern` (*view* `view`) *new-pattern*

Description The `set-pen-pattern` generic function sets the window's pen pattern.

Arguments

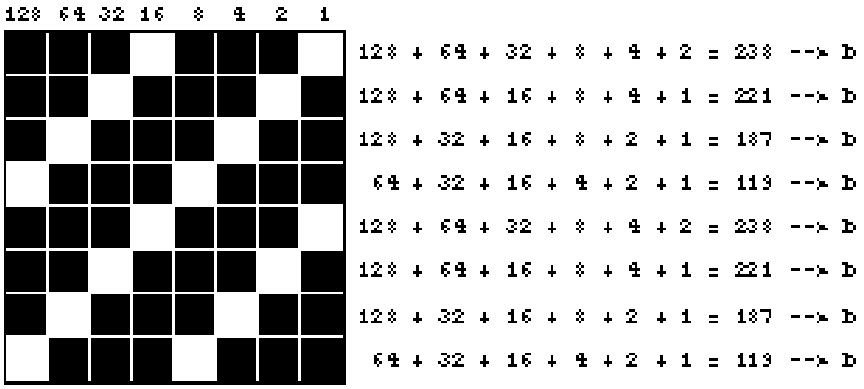
view	A window or a view contained in a window.
new-pattern	A pattern record.

A pattern is stored as a 64-bit block of memory (see Figure E-6). The definition of a pattern record allows patterns to be accessed as 8 bytes or four words.

Pattern records, like all records, continue to take up space on the Macintosh heap until they are explicitly disposed of.

```
(defrecord pattern
  (variant ((b0 byte) (b1 byte) (b2 byte) (b3 byte)
            (b4 byte) (b5 byte) (b6 byte) (b7 byte))
           ((w0 integer) (w1 integer)
            (w2 integer) (w3 integer))))
```

n **Figure E-6** Pen pattern stored as a 64-bit block of memory



Macintosh Common Lisp stores five patterns as constants: `*white-pattern*`, `*black-pattern*`, `*gray-pattern*`, `*light-gray-pattern*`, and `*dark-gray-pattern*`.

Pen modes affect the way drawing occurs in the window. They provide a logical mapping between the current state of pixels in the window and the state of the pixels being drawn.

pen-mode

[*Generic function*]

Syntax `pen-mode` (*view* `view`)

Description The `pen-mode` generic function returns a keyword indicating the window's current pen mode.

Argument *view* A window or a view contained in a window.

set-pen-mode

[*Generic function*]

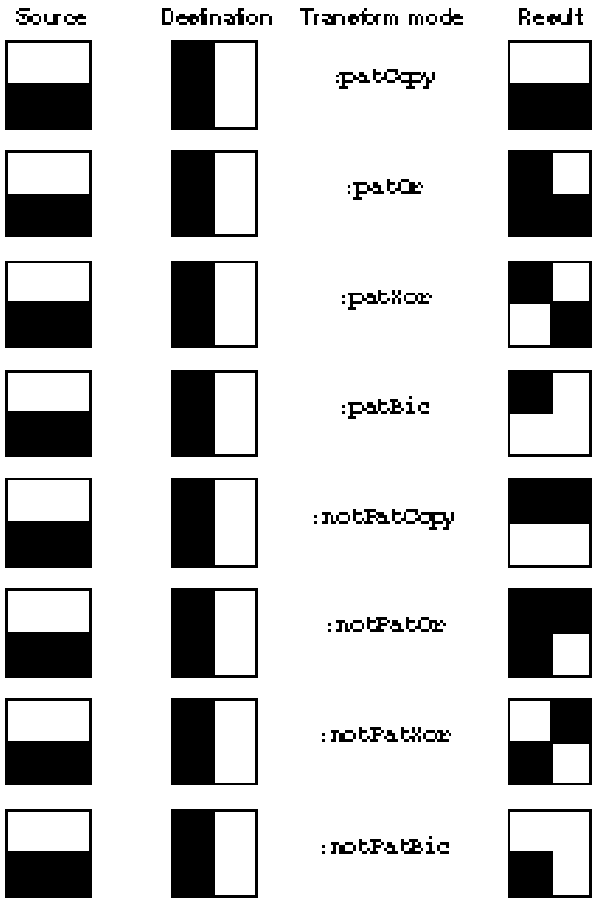
Syntax `set-pen-mode` (*view* `view`) *new-mode*

Description The `set-pen-mode` generic function sets the window's current pen mode.(See Figure E-7.)

Arguments *view* A window or a view contained in a window.

new-mode The new pen mode. This value should be one of the following keywords: `:patCopy`, `:patOr`, `:patXor`, `:patBic`, `:notPatCopy`, `:notPatOr`, `:notPatXor`, `:notPatBic`.

n **Figure E-7** Effect of pen modes on pixels being drawn



Pen-state records represent the pen mode as an integer. This integer is equal to the position of the corresponding pen-mode keyword in the list `*pen-modes*` . To translate an integer into a keyword, make the call (`elt *pen-state*` *mode-integer*). To translate a keyword into an integer, make the call (`position` *mode-keyword* `*pen-state*`).

Here is the definition of a pen-state record.

```
(defrecord PenState
      (pnLoc point)
      (pnSize point)
      (pnMode integer)
      (pnPat pattern))
```

pen-state

[*Generic function*]

Syntax `pen-mode` (*view* `view`) `&optional` *save-pen-state*

Description The `pen-state` generic function returns the current pen state, a record containing the pen's location, size, mode (as an integer), and pattern.

Pen-state records, like all records, continue to take up space on the Macintosh heap until they are explicitly disposed of.

Arguments

view A window or a view contained in a window.

save-pen-state A pointer to a pen-state record; the returned state is stored in this record. If *save-pen-state* is not given, the pen state is returned in a newly allocated record.

set-pen-state

[*Generic function*]

Syntax `set-pen-mode` (*view* `view`) *new-pen-state*

Description The `set-pen-state` generic function sets the window's pen state.

Arguments

view A window or a view contained in a window.

new-pen-state A pen-state record.

pen-normal

[*Generic function*]

Syntax `pen-normal` (*view* `view`)

Description The `pen-normal` generic function sets the pen size to `#@(1 1)`, the pen mode to `patCopy`, and the pen pattern to `*black-pattern*`. The pen location is not changed.

Argument

view A window or a view contained in a window.

move-to

[*Generic function*]

Syntax `move-to` (*view* `view`) *h* `&optional` *v*

Description The `move-to` generic function moves the pen to the point specified by *h* and *v* without doing any drawing. It returns the point to which the pen moved.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

move

[*Generic function*]

Syntax `move` (*view* `view`) *h* `&optional` *v*

Description The `move` generic function moves the pen *h* points to the right and *v* points down without doing any drawing.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

line-to

[*Generic function*]

Syntax `line-to` (*view* `view`) *h* `&optional` *v*

Description The `line-to` generic function draws a line from the pen's current position to the point represented by *h* and *v*.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

line

[*Generic function*]

Syntax `line` (*view* `view`) *h* `&optional` *v*

Description The `line` generic function draws a line to a point *h* points to the right and *v* points down from the current pen position.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

Drawing text

Macintosh Common Lisp draws text in windows by using a window as an output stream. Drawing of text takes place starting at the current pen position using the window's current font, size, style, and mode. The initial pen position determines the placement of the lower-left corner of the first character drawn, and the pen is moved to the right the width of each character after it is drawn. Special characters, such as carriage returns and backspaces, have no effect.

When a window is created, its pen position is `#@(0 0)`. This means that any text drawn in it will be above the visible portion of the window until the pen position is lowered.

stream-tyo

[*Generic function*]

Syntax `stream-tyo` (*view* `view`) *char*

Description The `stream-tyo` generic function draws *char* at the current pen position, in the current font, using the current text transfer mode. It then moves the pen to the right the width of the character. Because windows are streams, all stream output functions (such as `prin1`) can be performed on them. The `stream-tyo` function is not normally called directly but instead by stream output functions.

Arguments

view	A window or a view contained in a window.
char	A character.

Calculations with rectangles

The following functions do not draw; they simply perform calculations. They do not depend on a GrafPort, and so they are defined globally rather than as generic functions.

`offset-rect` [*Function*]

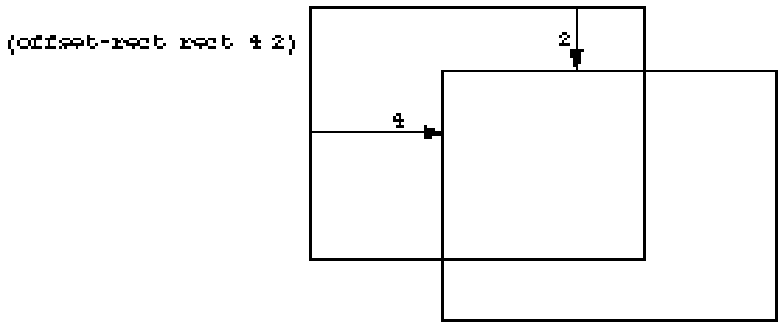
Syntax `offset-rect` *rectangle h* `&optional` *v*

Description The `offset-rect` function moves *rectangle h* to the right and *v* down. (See Figure E-8.) It returns the destructively modified rectangle .

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

n **Figure E-8** Offset rectangle, with *h* equal to 4 and *v* equal to 2



inset-rect

[*Function*]

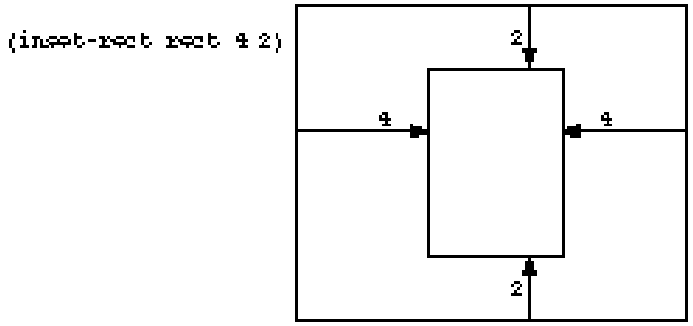
Syntax `inset-rect` *rectangle h* `&optional` *v*

Description The `inset-rect` function shrinks or expands *rectangle* by *h* and *v*. It returns the destructively modified rectangle. If *h* and *v* are positive, the left and right sides and the top and bottom move toward the center. If *h* and *v* are negative, the sides move outward. See Figure E-9.

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

Figure E-9 Inset rectangle, with *h* equal to 4 and *v* equal to 2



intersect-rect

[*Function*]

Syntax `intersect-rect` *rect1 rect2 dest-rect*

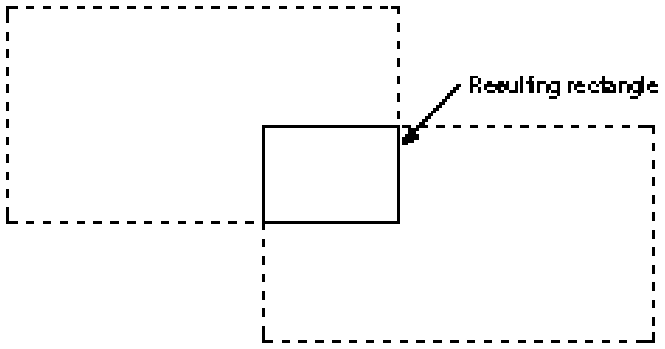
Description The `intersect-rect` function stores in *dest-rect* the rectangle created by the intersection of *rect1* and *rect2* and returns *dest-rect*. (See Figure E-10.)

A single rectangle may be passed as *dest-rect* and as *rect1* or *rect2*, making it unnecessary to allocate one extra rectangle.

Arguments

rect1	A rectangle.
rect2	A rectangle.
dest-rect	A rectangle record used to hold the intersection of *rect1* and *rect2*.

Figure E-10 Rectangle resulting from the intersection of two others



union-rect

[*Function*]

Syntax `union-rect` *rect1 rect2 dest-rect*

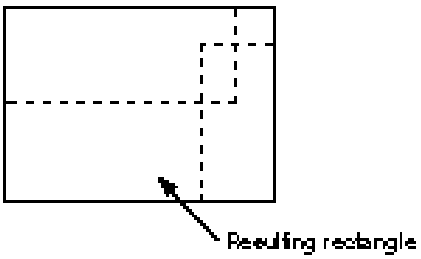
Description The `union-rect` function stores in *dest-rect* the smallest rectangle that encloses both *rect1 and rect2* and returns *dest-rect*. (See Figure E-11.)

A single rectangle may be passed as *dest-rect* and as *rect1* or *rect2*, making it unnecessary to allocate one extra rectangle.

Arguments

rect1	A rectangle.
rect2	A rectangle.
dest-rect	A rectangle record used to hold the rectangle enclosing *rect1* and *rect2*.

n **Figure E-11** Smallest rectangle completely enclosing two others



point-in-rect-p [*Function*]

Syntax `point-in-rect-p` *rectangle h* `&optional` *v*

Description The `point-in-rect-p` function returns `t` if the point specified by *h* and *v* is inside *rectangle;* otherwise, it returns `nil`.

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

points-to-rect [*Function*]

Syntax `points-to-rect` *point1 point2 dest-rect*

Description The `points-to-rect` function stores in *dest-rect* the smallest rectangle that encloses both *point1* and *point2*, and returns *dest -rect.*

The `points-to-rect` function is useful when you have two corner points but don't know which one is the top left and which one is the bottom right.

Arguments

point1	A point that specifies one of the corners of the rectangle.
point2	A point that represents the other corner of the rectangle.
dest-rect	A rectangle record used to hold the result of the calculations.

point-to-angle

[*Function*]

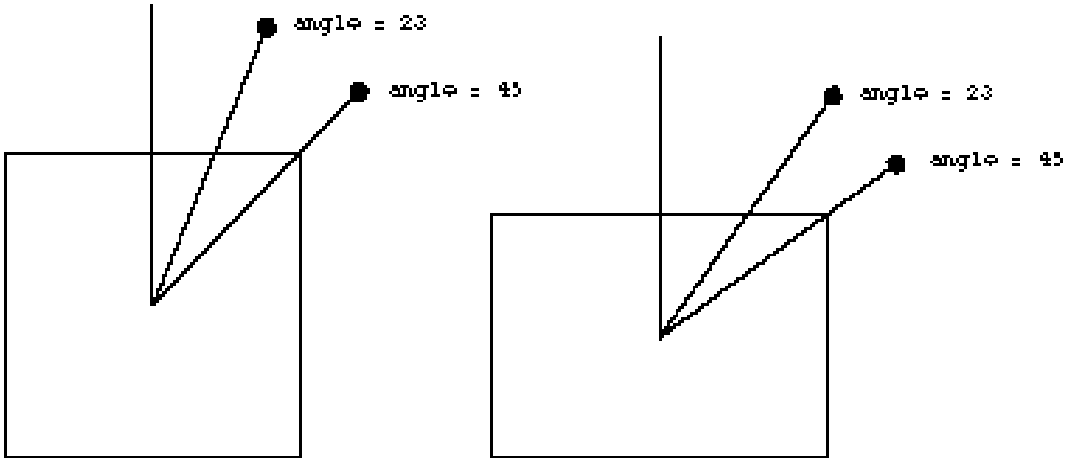
Syntax `point-to-angle` *rectangle h* `&optional` *v*

Description The `point-to-angle` function returns an angle number calculated from *rectangle* and the point specified by *h* and *v* (for details, see *Inside Macintosh*). (See Figure E-12.)

Arguments

rectangle	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to represent a point.

Figure E-12 Point to angle, calculated from two rectangles



equal-rect

[*Function*]

Syntax `equal-rect` *rect1 rect2*

Description The `equal-rect` function returns `t` if *rect1* and *rect2* are equal and `nil` otherwise.

Arguments

rect1	A rectangle.
rect2	A rectangle.

empty-rect-p [*Function*]

Syntax `empty-rect-p` *left* `&optional` *top right bottom*

Description The `empty-rect-p` function returns `t` if the rectangle specified by *arg* is empty (contains no points) and `nil` otherwise.

A rectangle is empty if its bottom coordinate is less than or equal to the top or if the right coordinate is less than or equal to the left.

Arguments *left* , *top, right, bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on rectangles

Five generic functions govern graphics operations on rectangles.

frame-rect [*Generic function*]

Syntax `frame-rect` (*view* `view`) *left* `&optional` *top right bottom*

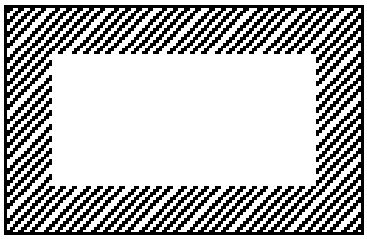
Description The `frame-rect` generic function draws a line just inside the boundaries of the rectangle specified by *arg*, using the current pen. (See Figure E-13.)

Arguments *view* A window or a view contained in a window.

left , top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

n **Figure E-13** Rectangle framed in the current pen



paint-rect

[*Generic function*]

Syntax `paint-rect` (*view* `view`) *left* `&optional` *top right bottom*

Description The `paint-rect` generic function fills the rectangle specified by*arg* with the current pen pattern and mode.

Arguments *view* A window or a view contained in a window.

left , top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

erase-rect

[*Generic function*]

Syntax `erase-rect` (*view* `view`) *left* `&optional` *top right bottom*

Description The `erase-rect` generic function fills the rectangle specified by *arg* with the current background pattern (in `patCopy` mode).

Arguments

view A window or a view contained in a window.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

invert-rect

[*Generic function*]

Syntax `invert-rect` (*view* `view`) *left* `&optional` *top right bottom*

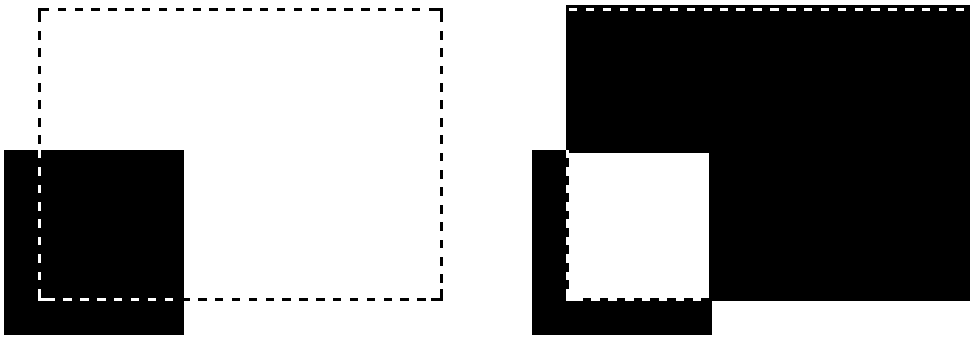
Description The `invert-rect` generic function inverts the pixels inside the rectangle specified by *arg* . (See Figure E-14.)

Arguments

view A window or a view contained in a window.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

n **Figure E-14** Effects of `paint-rect` and `invert-rect`



`fill-rect` [*Generic function*]

Syntax `fill-rect` (*view* `view`) *pattern left* `&optional` *top right bottom*

Description The `fill-rect` generic function fills the rectangle specified by *arg* with *pattern* (in `:patCopy` mode).

Arguments

view A window or a view contained in a window.

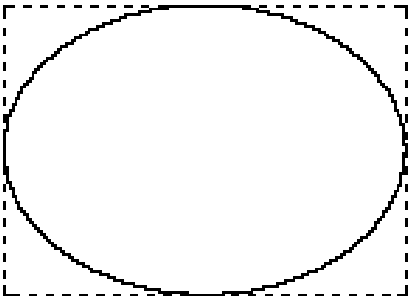
pattern A pattern record; see `pen-pattern` earlier in this appendix for a discussion of pattern records.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on ovals

Ovals are drawn just inside rectangles. The oval is determined by the specified rectangle. (See Figure E-15.)

n **Figure E-15** An oval within a rectangle



`frame-oval` [*Generic function*]

Syntax `frame-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `frame-oval` generic function draws a line just inside the boundaries of the oval specified by the rectangle, using the current pen pattern, mode, and size. The rectangle is specified by *arg*.

Arguments

view A window or a view contained in a window.

left , *top, right, bottom*
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

paint-oval

[*Generic function*]

Syntax `paint-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `paint-oval` generic function fills the oval specified by the rectangle specified by the arguments with the current pen pattern and mode.

Arguments

view A window or a view contained in a window.

left , *top, right, bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

erase-oval

[*Generic function*]

Syntax `erase-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `erase-oval` generic function fills the oval specified by the rectangle with the current background pattern (in: `patCopy` mode). The rectangle is specified by the arguments

Arguments

view A window or a view contained in a window.

left , *top, right, bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

invert-oval

[*Generic function*]

Syntax `invert-oval` (*view* `view`) *left* `&optional` *top right bottom*

Description The `invert-oval` generic function inverts the pixels enclosed by the oval specified by the rectangle. The rectangle is specified by the arguments.

Arguments

view A window or a view contained in a window.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

fill-oval

[*Generic function*]

Syntax `fill-oval` (*view* `view`) *pattern left* `&optional` *top right bottom*

Description The `fill-oval` generic function fills the oval specified by the rectangle with *pattern* (in `:patCopy` mode). The rectangle is specified by the arguments

Arguments

view A window or a view contained in a window.

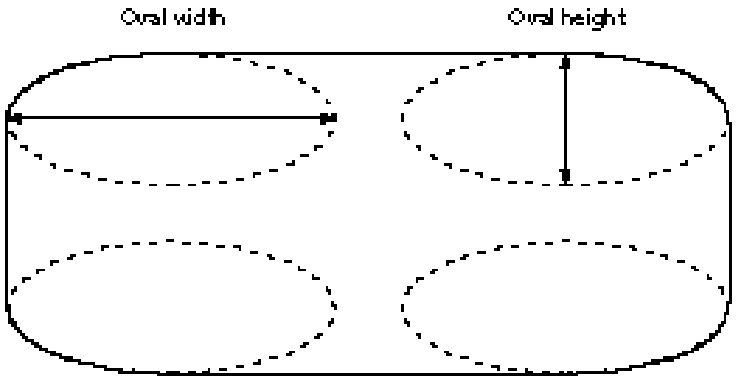
pattern A pattern record; see `pen-pattern` earlier in this appendix for a discussion of pattern records.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on rounded rectangles

A rounded rectangle (see Figure E-16) is a rectangle whose corners are rounded. The shapes of the corners are determined by ovals associated with the rounded rectangles. Thus, a rounded rectangle is determined by (1) the rectangle, (2) the width of the oval, and (3) the height of the oval.

n **Figure E-16** Rounded rectangle



`frame-round-rect` [*Generic function*]

Syntax `frame-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description The `frame-round-rect` generic function draws a line just inside the boundaries of the rounded rectangle, using the current pen pattern, mode, and size. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height* .

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`paint-round-rect` [*Generic function*]

Syntax

`paint-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description

The `paint-round-rect` generic function fills the rounded rectangle with the current pen pattern and mode. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height* .

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

erase-round-rect

[*Generic function*]

Syntax `erase-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description The `erase-round-rect` generic function fills the rounded rectangle *t* with the current background pattern using `patCopy` mode. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height* .

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left , top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

invert-round-rect

[*Generic function*]

Syntax `invert-round-rect` (*view* `view`) *oval-width oval-height left* `&optional` *top right bottom*

Description The `invert-round-rect` generic function inverts the pixels enclosed by the rounded rectangle. The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height* .

Arguments

view A window or a view contained in a window.

oval-width The width of the oval used to shape the rounded corner.

oval-height The height of the oval used to shape the rounded corner.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`fill-round-rect` [*Generic function*]

Syntax

`fill-round-rect` (*view* `view`) *pattern oval-width oval-height left* `&optional` *top right bottom*

Description

The `fill-round-rect` generic function fills the specified rounded rectangle with the given *pattern* (in `:patCopy` mode). The rounded rectangle is specified by the rectangle, *oval-width,* and *oval-height* .

Arguments

view A window or a view contained in a window.

pattern A pattern record; see `open-pattern` earlier in this appendix for a discussion of pattern records.

oval-width The width of the oval used to shape the rounded rectangle corner.

oval-height The height of the oval used to shape the rounded rectangle corner.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Graphics operations on arcs

These functions perform graphics operations on arcs and wedge-shaped sections of ovals.

frame-arc

[*Generic function*]

Syntax `frame-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description The `frame-arc` generic function draws a line just inside the arc specified by the rectangle, *start-angle* , and *arc-angle* using the current pen pattern, mode, and size. The rectangle is specified by the arguments. Figure E-17 shows an arc with a start angle of 45 and an arc angle of 135 inside a rectangle with the coordinates `#@(0 0)` and `#@(150 100)` .

Arguments

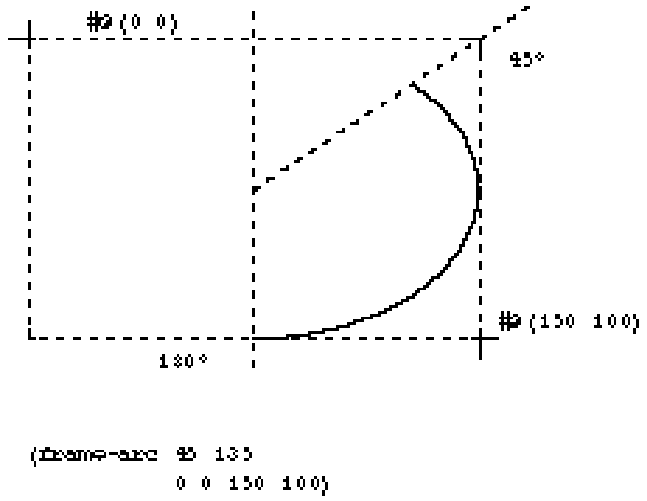
view A window or a view contained in a window.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up. (See the documentation of the `FrameArc` procedure in *Inside Macintosh* .)

arc-angle The angle subtended by the arc.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

n **Figure E-17** Framing an arc



paint-arc

[*Generic function*]

Syntax `paint-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description The `paint-arc` generic function fills the arc specified by the rectangle, *start-angle*, and *arc-angle* with the current pen pattern and mode. The rectangle is specified by the arguments

Arguments

view A window or a view contained in a window.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.

arc-angle The angle subtended by the arc.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

erase-arc

[*Generic function*]

Syntax `erase-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description The `erase-arc` generic function fills the specified arc with the current background pattern. The arc is specified by the rectangle, *start-angle* , and *arc-angle* . The rectangle is specified by *left, top, right, bottom.*

Arguments

view A window or a view contained in a window.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.

arc-angle The angle subtended by the arc.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

invert-arc

[*Generic function*]

Syntax `invert-arc` (*view* `view`) *start-angle arc-angle left* `&optional` *top right bottom*

Description The `invert-arc` generic function inverts the pixels enclosed by the arc specified by the rectangle, *start-angle* , and *arc-angle* . The rectangle is specified by *left, top, right, bottom.*

Arguments

view A window or a view contained in a window.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.

arc-angle The angle subtended by the arc.

left , *top, right, bottom*
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

`fill-arc` [*Generic function*]

Syntax

`fill-arc` (*view* `view`) *pattern start-angle arc-angle left* `&optional` *top right bottom*

Description

The `fill-arc` generic function draws a line just inside the arc specified by the rectangle, *start-angle* , and *arc-angle* using the current pen pattern, mode, and size. The rectangle is specified by *left, top, right, bottom*.

Arguments

view A window or a view contained in a window.

pattern A pattern record; see `pen-pattern` earlier in this appendix for a discussion of pattern records.

start-angle The angle at which the arc originates, represented as an integer. An angle of 0 points straight up.

arc-angle The angle subtended by the arc.

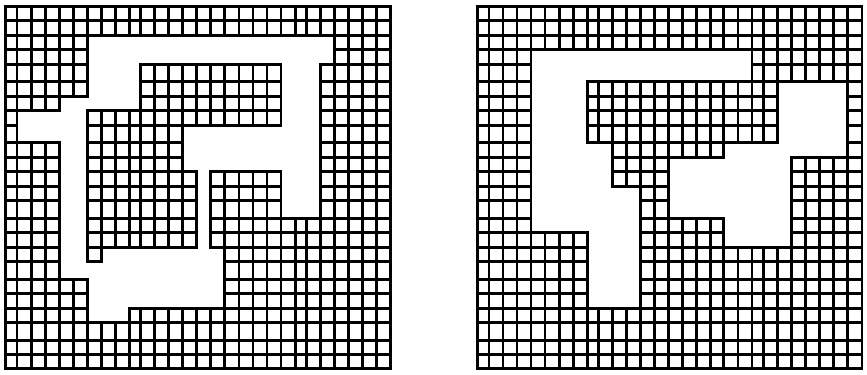
left , *top, right, bottom*
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

Regions

A region divides the graphics plane of points into two sets of points: those inside the region and those outside the region. Regions can have any arbitrary shape. (See Figure E-18.)

The storage for regions is not subject to automatic garbage collection. You must reclaim region storage by calling the function `dispose-region` . With this limitation, the use of regions has been greatly simplified from the specification given in *Inside Macintosh* . Specifically, much of the initialization of regions is performed automatically.

n **Figure E-18** Regions



`new-region` [*Function*]

Syntax `new-region`

Description The `new-region` function allocates a new empty region and returns it.

dispose-region [*Function*]

Syntax `dispose-region` *region*

Description The `dispose-region` function reclaims storage space used by *region* and returns `nil`.

Argument *region* A region.

copy-region [*Function*]

Syntax `copy-region` *region* `&optional` *dest-region*

Description The `copy-region` function either copies *region* into *dest-region*, if it is supplied, or creates a new region equivalent to *region*. It returns the new region or *dest-region* .

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments *region* A region.
dest-region Another region.

set-empty-region [*Function*]

Syntax `set-empty-region` *region*

Description The `set-empty-region` function destructively modifies *region* so that it is empty and returns the empty region.

Argument *region* A region.

set-rect-region [*Function*]

Syntax `set-rect-region` *region left* `&optional` *top right bottom*

Description The `set-rect-region` function sets *region* so that it is equivalent to the rectangle specified by the arguments and returns the rectangular region.

Arguments *region* A region.

left , top, right, bottom

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

open-region

[*Generic function*]

Syntax `open-region (`*view* `view)`

Description The `open-region` generic function hides the pen and begins recording a region. Subsequent drawing commands to the window add to the region. Recording ends when `close-region` is called. The function returns `nil`.

It is an error to call `open-region` a second time without first calling `close-region` .

Argument *view* A window or a view contained in a window.

close-region

[*Generic function*]

Syntax `close-region (`*view* `view) &optional` *dest-region*

Description The `close-region` generic function shows the pen and returns a region that is the accumulation of drawing commands in the window since the last `open-region` for the window. It returns the result in *dest-region,* if supplied, or else in a newly created region.

It is an error to call `close-region` before `open–region` has been called.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments *view* A window or a view contained in a window.

dest-region A region.

Calculations with regions

The following functions do not draw; they simply perform calculations. They do not depend on a GrafPort, and so they are defined globally rather than as generic functions.

offset-region [*Function*]

Syntax `offset-region` *region h* `&optional` *v*

Description The `offset-region` function destructively offsets *region* by *h* to the right and *v* down and returns the offset region. If only *h* is given, it is interpreted as an encoded point, and its coordinates are used.

Arguments

region	A region.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

inset-region [*Function*]

Syntax `inset-region` *region h* `&optional` *v*

Description The `inset-region` function destructively shrinks or expands *region* by *h* horizontally and *v* vertically and returns it. If only *h* is given, it is interpreted as an encoded point, and its coordinates are used.

Arguments

region	A region.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

intersect-region

[*Function*]

Syntax `intersect-region` *region1 region2* `&optional` *dest-region*

Description The `intersect-region` function returns a region that is the intersection of *region1* and *region2*. It returns the result in *dest-region*, if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

union-region

[*Function*]

Syntax `union-region` *region1 region2* `&optional` *dest-region*

Description The `union-region` function returns a region that is the union of *region1* and *region2*. It returns the result in *dest-region* , if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

difference-region

[*Function*]

Syntax `difference-region` *region1 region2* `&optional` *dest-region*

Description The `difference-region` function returns a region that is the difference of *region1* and *region2*. It returns the result in *dest-region* , if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

xor-region

[*Function*]

Syntax `xor-region` *region1 region2* `&optional` *dest-region*

Description The `xor-region` function returns a region that consists of all the points that are in *region1* or *region2*, but not both. It returns the result in *dest-region* , if supplied, or else in a newly created region.

Note that if a new region is created, you must dispose of it explicitly to reclaim its storage space.

Arguments

region1	A region.
region2	A region.
dest-region	A region.

point-in-region-p

[*Function*]

Syntax `point-in-region-p` *region h* `&optional` *v*

Description The `point-in-region-p` function returns `t` if the point specified by *h* and *v* is contained in *region;* otherwise, it returns `nil`. If only *h* is given, it is interpreted as an encoded point.

Arguments

region	A region.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

rect-in-region-p

[*Function*]

Syntax `rect-in-region-p` *region left* `&optional` *top right bottom*

Description The `rect-in-region-p` function returns `t` if the intersection of the rectangle specified by the arguments and *region* contains at least one point; otherwise it returns `nil`.

Arguments

region A region.

left , *top*, *right*, *bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

equal-region-p

[*Function*]

Syntax `equal-region-p` *region1 region2*

Description The `equal-region-p` function returns `t` if *region 1* and *region 2* are identical in size, shape, and position; otherwise it returns `nil`.

Arguments

region1 A region.

region2 A region.

empty-region-p

[*Function*]

Syntax `empty-region-p` *region*

Description The `empty-region-p` function returns `t` if *region* contains no points and `nil` otherwise.

Argument

region A region.

Graphics operations on regions

These functions allow graphics operations on regions.

frame-region [*Generic function*]

Syntax `frame-region` (*view* `view`) *region*

Description The `frame-region` generic function draws a line just inside the boundaries of *region* , using the current pen.

Arguments

view	A window or a view contained in a window.
region	A region.

paint-region [*Generic function*]

Syntax `paint-region` (*view* `view`) *region*

Description The `paint-region` generic function fills *region* with the current pen pattern and mode.

Arguments

view	A window or a view contained in a window.
region	A region.

erase-region [*Generic function*]

Syntax `erase-region` (*view* `view`) *region*

Description The `erase-region` generic function fills *region* with the current background pattern, using `patCopy` mode.

Arguments

view	A window or a view contained in a window.
region	A region.

invert-region

[*Generic function*]

Syntax `invert-region` (*view* `view`) *region*

Description The `invert-region` generic function inverts the pixels enclosed by *region* .

Arguments

view	A window or a view contained in a window.
region	A region.

fill-region

[*Generic function*]

Syntax `fill-region` (*view* `view`) *pattern region*

Description The `fill-region` generic function fills *region* with *pattern* , using `:patCopy` mode.

Arguments

view	A window or a view contained in a window.
pattern	A pattern record; see `pen-pattern` earlier in this appendix for a discussion of pattern records.
region	A region.

Bitmaps

Bitmaps are rectangular arrays of pixels that are either black or white. The following functions are useful in manipulating bitmaps.

make-bitmap

[*Function*]

Syntax `make-bitmap` *left* `&optional` *top right bottom*

Description The `make-bitmap` function returns a new bitmap the size of the rectangle specified by the arguments. This bitmap is not displayed anywhere but can be used for calculations and storage.

Arguments *left , top, right, bottom*

These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

---

copy-bits

[*Function*]

Syntax `copy-bits` *bitmap1 bitmap2 rect1 rect2* `&optional` *pen-mode region*

Description The `copy-bits` function copies and scales the bits inside *rect1* of *bitmap1* to the bits inside *rect2* of *bitmap2* using the transfer mode *pen-mode* .

If *region* is given, `copy-bits` clips the transferred bitmap to *region* and *pen–mode* assumes the default value `:srcCopy` .

Arguments

bitmap1	A bitmap.
bitmap2	A bitmap.
rect1	A rectangle.
rect2	A rectangle.
pen-mode	A pen mode. Should be one of the following keywords: `:patCopy` , `:patOr` , `:patXor` , `:patBic` , `:notPatCopy` , `:notPatOr` , `:notPatXor` , `:notPatBic` .
region	A region.

---

scroll-rect

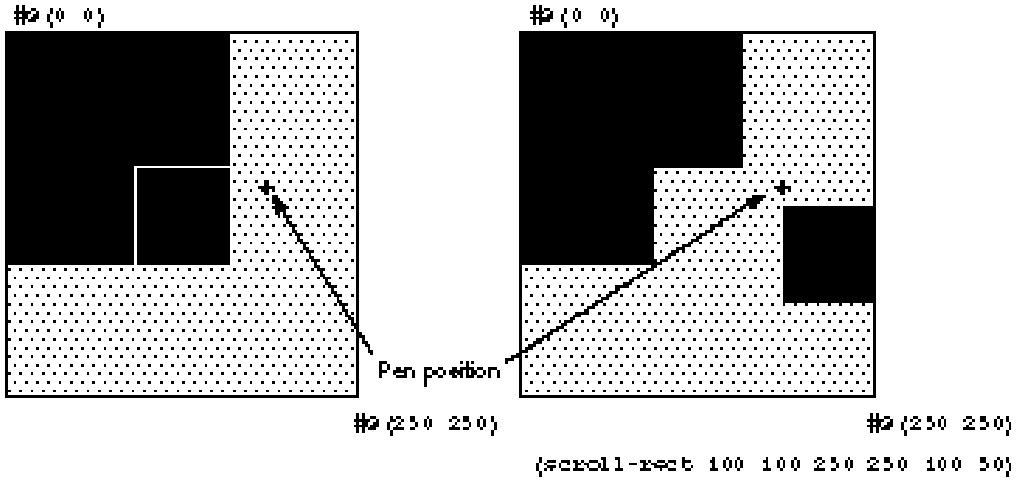
[*Generic function*]

Syntax `scroll-rect` (*view* `view`) *rectangle h* `&optional` *v*

Description The `scroll-rect` generic function shifts the bits *h* pixels to the right and *v* pixels down within *rectangle* , erases the uncovered region, and adds the uncovered region to the window's update region. See Figure E-19 for an example of a scrolled rectangle.

Arguments	*view*	A window or a view contained in a window.
	rectangle	A rectangle.
	h	Horizontal position.
	v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

n **Figure E-19** A rectangle scrolled down and to the right



Pictures

A picture is a recording of a sequence of QuickDraw commands. Pictures may be played back at a later time, into any window. The MCL picture commands are slightly different from the QuickDraw ones, because Macintosh Common Lisp takes care of some of the memory management automatically. There are also some additional capabilities for manipulating pictures not found in QuickDraw.

start-picture

[*Generic function*]

Syntax `start-picture` (*view* `view`) *left* `&optional` *top right bottom*

Description The `start-picture` generic function hides the pen and starts recording QuickDraw commands in a picture whose frame is the rectangle specified by the arguments, if supplied. Otherwise, the window's PortRect is the frame. The function returns `nil`.

It is an error to call `start-picture` a second time before calling `get-picture`.

You must dispose of pictures explicitly by calling `kill-picture` to reclaim their storage space

Arguments

view A window or a view contained in a window.

left , *top, right, bottom*
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

get-picture

[*Generic function*]

Syntax `get-picture` (*view* `view`)

Description The `get-picture` generic function shows the pen and returns a new picture representing the cumulative effect of all the QuickDraw commands given since the last call to `start–picture`.

It is an error to call `get-picture` before `start-picture` has been called in a window.

You must dispose of pictures explicitly by calling `kill-picture` to reclaim their storage space

Argument

view A window or a view contained in a window.

draw-picture

[*Generic function*]

Syntax `draw-picture` (*view* `view`) *picture* `&optional` *left top right bottom*

Description The `draw-picture` generic function draws *picture* in the window and returns *picture* .

Note that if the PortRect was used as a frame when the picture was made, and if the PortRect was arbitrarily large (the default set up by Macintosh Common Lisp), then scaling will produce no drawing (since the drawing frame is so much smaller than the creation frame).

Arguments

view A window or a view contained in a window.

picture A picture.

left , top, right, bottom
These four arguments are used together to specify the rectangle. If only *left* is given, it should be a pointer to a rectangle record. If only two arguments are given, they should be points specifying the upper-left and lower-right coordinates of the rectangle. If all four arguments are given, they should be coordinates representing the left, top, right, and bottom of the rectangle.

kill-picture

[*Function*]

Syntax `kill-picture` *picture*

Description The `kill-picture` function reclaims the storage space used by *picture* and returns `nil`.

Argument *picture* A picture.

Polygons

The MCL polygon commands are different from QuickDraw ones because Macintosh Common Lisp handles some of the memory management automatically.

start-polygon

[*Generic function*]

Syntax `start-polygon` (*view* `view`)

Description The `start-polygon` generic function hides the pen and starts making a polygon. Subsequent `line` and `line-to` commands are added to a new polygon.

Within a single window, it is an error to call `start-polygon` twice before calling `get-polygon` .

You must dispose of polygons explicitly, using `kill-polygon` to reclaim their storage space.

Argument *view* A window or a view contained in a window.

get-polygon

[*Generic function*]

Syntax `get-polygon` (*view* `view`)

Description The `get-polygon` generic function shows the pen and returns a polygon representing the cumulative effect of all the `line` and `line-to` commands since the last call to `start–polygon` .

Within a single window, it is an error to call `get-polygon` before a `start-polygon` has been called.

You must dispose of polygons explicitly, using `kill-polygon` to reclaim their storage space.

Argument *view* A window or a view contained in a window.

kill-polygon

[*Function*]

Syntax `kill-polygon` *polygon*

Description The `kill-polygon` function reclaims storage space used by *polygon* and returns `nil`.

Argument *polygon* A polygon.

offset-polygon

[*Function*]

Syntax `offset-polygon` *polygon h* `&optional` *v*

Description The `offset-polygon` function offsets *polygon* by *h* to the right and *v* down. This function can be performed outside of windows because it does not involve drawing. If only *h* is given, it is interpreted as an encoded point.

Arguments

polygon	A polygon.
h	Horizontal position.
v	Vertical position. If *v* is nil (the default), *h* is assumed to specify both values.

frame-polygon

[*Generic function*]

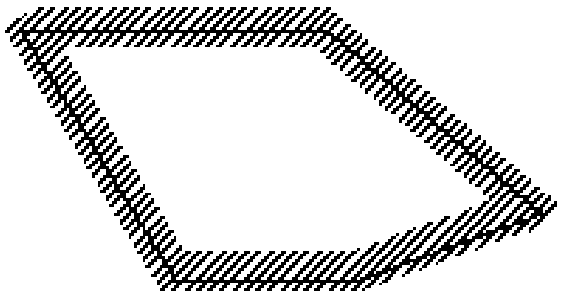
Syntax `frame-polygon` (*view* `view`) *polygon*

Description The `frame-polygon` generic function draws a line just inside the boundaries of *polygon* using the current pen. A framed polygon is shown in Figure E-20.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

n **Figure E-20** A framed polygon



paint-polygon [*Generic function*]

Syntax `paint-polygon` (*view* `view`) *polygon*

Description The `paint-polygon` generic function fills *polygon* with the current pen pattern and mode.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

erase-polygon [*Generic function*]

Syntax `erase-polygon` (*view* `view`) *polygon*

Description The `erase-polygon` generic function fills *polygon* with the current background pattern, using `patCopy` mode.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

invert-polygon [*Generic function*]

Syntax `invert-polygon` (*view* `view`) *polygon*

Description The `invert-polygon` generic function inverts the pixels enclosed by *polygon*.

Arguments

view	A window or a view contained in a window.
polygon	A polygon.

fill-polygon [*Generic function*]

Syntax `fill-polygon` (*view* `view`) *pattern polygon*

Description The `fill-polygon` generic function fills *polygon* with *pattern* using `:patCopy` mode.

Arguments	*view*	A window or a view contained in a window.
	pattern	A pattern record; see `pen-pattern` earlier in this appendix for a discussion of pattern records.
	polygon	A polygon.

Miscellaneous procedures

This section contains functions to perform miscellaneous graphics procedures.

`local-to-global` [*Generic function*]

Syntax `local-to-global` (*view* `view`) *h* `&optional` *v*

Description The `local-to-global` generic function returns a global point that corresponds to the window's local point specified by *h* and *v*. If only *h* is given, it is taken to be an encoded point.

Arguments	*view*	A window or a view contained in a window.
	h	Horizontal position.
	v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

`global-to-local` [*Generic function*]

Syntax `global-to-local` (*view* `view`) *h* `&optional` *v*

Description The `global-to-local` generic function returns a point in the window's coordinate system that corresponds to the global point specified by *h* and *v*. If only *h* is given, it is interpreted as an encoded point.

Arguments	*view*	A window or a view contained in a window.
	h	Horizontal position.
	v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

get-pixel

[*Generic function*]

Syntax `get-pixel` (*view* `view`) *h* `&optional` *v*

Description The `get-pixel` generic function returns `t` if the pixel specified by *h* and *v* is black and within the window's `VisRgn`; otherwise, it returns `nil`. If only *h* is given, it is interpreted as an encoded point.

Arguments

view	A window or a view contained in a window.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

scale-point

[*Function*]

Syntax `scale-point` *rect1 rect2 h* `&optional` *v*

Description The `scale-point` function returns a point whose horizontal and vertical values are the scaled horizontal and vertical values of the point specified by *h* and *v*. If only *h* is given, it is interpreted as an encoded point. The scaling corresponds to the ratios of the width and height of *rect1* to the width and height of *rect2*.

Arguments

rect1	A rectangle.
rect2	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

map-point

[*Function*]

Syntax `map-point` *source-rect dest-rect h* `&optional` *v*

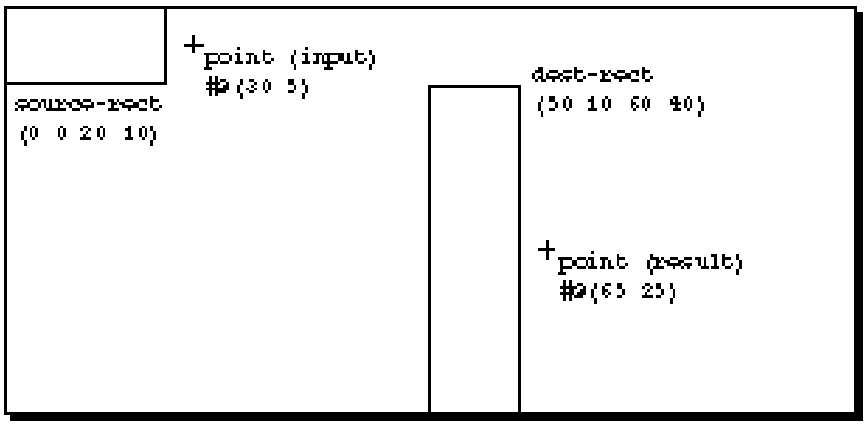
Description The `map-point` function returns a point that corresponds to *dest-rect* as the point specified by *h* and *v* corresponds to *source-rect*. If only *h* is given, it is interpreted as an encoded point.

The effect of `map-point` is shown in Figure E-21, where the point (30, 5) corresponds to *source-rect* as the point (65,25) does to *dest-rect*. The point `#@(30 5)` is half the width of *source-rect* to the right of *source-rect* and is located at the vertical midpoint of *source-rect* . The point `#@(65 25)` bears the same relation to *dest-rect* .

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
h	Horizontal position.
v	Vertical position. If *v* is `nil` (the default), *h* is assumed to specify both values.

Figure E-21 Effect of `map-point`



map-rect

[*Function*]

Syntax `map-rect` *source-rect dest-rect mapped-rect*

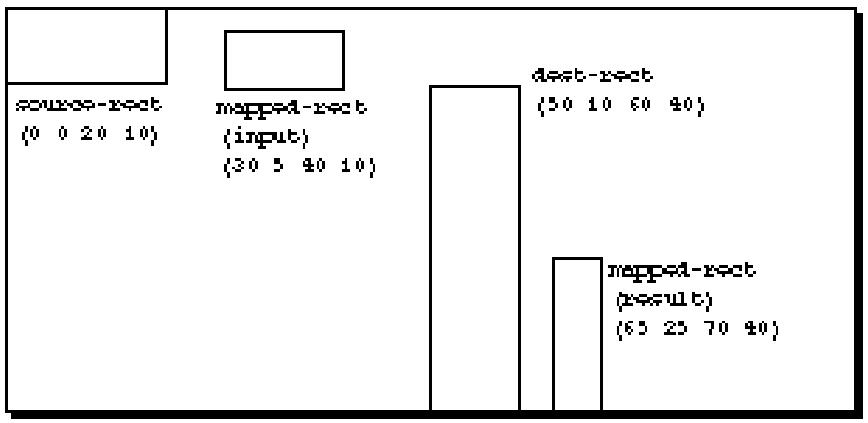
Description The `map-rect` function returns a rectangle that corresponds to *dest-rect* as *mapped-rect* corresponds to *source-rect* . The function destructively modifies *mapped-rect* to hold the returned value. The effect of `map-rect` is shown in Figure E-22, where the returned rectangle corresponds to *dest-rect* as *mapped-rect* (input) corresponds to *source-rect*.

This function is performed by applying `map-point` to the corner points of *mapped-rect* .

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
mapped-rect	A rectangle.

n **Figure E-22** Effect of `map-rect`



map-region

[*Function*]

Syntax `map-region` *source-rect dest-rect region*

Description The `map-region` function returns a region that corresponds to *dest-rect* as *region* corresponds to *source-rect* . The function destructively modifies *region* to hold the return value.

This function is effectively performed by applying `map-point` to all the points in the region.

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
region	A region.

map-polygon

[*Function*]

Syntax `map-polygon` *source-rect dest-rect polygon*

Description The `map-polygon` function returns a polygon that corresponds to *dest-rect* as *polygon* corresponds to *source-rect* . The function destructively modifies *polygon* to hold the returned value.

This function is effectively performed by applying `map-point` to all the points that define the polygon.

Arguments

source-rect	A rectangle.
dest-rect	A rectangle.
polygon	A polygon.

Appendix F **For More Information**

Contents

The single most valuable source / 764
If you are learning Lisp / 764
If you are learning CLOS / 765
If you're learning about Macintosh programming / 766
If you're updating from a previous version of Macintosh Common Lisp / 766
If you'd like examples of MCL programming / 765
If you'd like to communicate with other MCL programmers / 767
Communicating through Internet / 767
Communicating through Usenet / 767
Communicating through AppleLink / 768
Communicating through CompuServe / 768
Communicating through other services / 768
Bugs and bug fixes / 768
Infrequent announcements of general interest / 769
If you're a developer: APDA / 769

This appendix describes useful background reading and other sources of information, including example files, online help, and the MCL mailing list, `info-mcl`, available to all MCL programmers.

The single most valuable source

At least until the release of the draft ANSI specification, *Common Lisp: The Language* is the standard reference manual and language specification for CLOS, as well as the most up-to-date specification for all of Common Lisp. It is not a tutorial, but is a must for every Lisp user.

Steele, Guy L., and others. *Common Lisp: The Language.* Second edition. Maynard, MA: Digital Press, 1990.

If you are learning Lisp

Many good tutorials exist on Common Lisp. Here is a selection of the best.

Brooks, Rodney. *Programming in Common Lisp.* New York: John Wiley, 1985.

Koschmann, Timothy. *The Common Lisp Companion* . New York: John Wiley, 1990.

This well-written book includes material from the second edition of *Common Lisp: The Language* , including information on CLOS. The author is on the ANSI XJ313 standards committee for Common Lisp.

Miller, Molly M., and Eric Benson. *Lisp Style and Design.* Maynard, MA: Digital Press, 1990.

An excellent intermediate-level book for someone learning Common Lisp.

Norvig, Peter. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* San Mateo, CA: Morgan Kauffmann, 1991.

An excellent book for all levels of Common Lisp programming, with many examples. Source code for this book is available.

Tatar, Deborah. *A Programmer's Guide to Common Lisp.* Maynard, MA: Digital Press, 1987.

Touretzky, David. *Common Lisp: A Gentle Introduction to Symbolic Computing*. Reading, MA: Addison-Wesley, 1990.

Touretzky writes well and includes many good examples for novices.

Wilensky, Robert. *Common LispCraft*. New York: Norton, 1986.

Winston, Patrick, and Berthold Claus Paul Horn. *Lisp*. Third edition. New York: Harper & Row, 1989.

A thorough academic text that includes material on CLOS.

Finally, though it uses Scheme instead of Lisp, the following is possibly the best book ever written on computer programming. With a relatively small knowledge of Lisp, you can follow it:

Abelson, Harold, and Gerald Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press, 1985.

If you are learning CLOS

Because CLOS is such a new standard, it is integrated into relatively few tutorials. Sonya Keene's book, below, is generally reckoned to be the most thorough CLOS tutorial on the market. Keene was also a contributor to the second edition of *Common Lisp: The Language*.

Keene, Sonya E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Reading, MA: Addison-Wesley, 1989.

Lawless, Jo A., and Molly M. Miller. *Understanding CLOS: The Common Lisp Object System*. Maynard, MA: Digital Press, 1991.

The books by Winston and Horn, Norvig, and Koschmann, described in the previous section, also include material on CLOS.

The following book describes current thinking about the Metaobject Protocol.

Kiczales, Gregor, and others. *The Art of the Metaobject Protocol*. Cambridge, MA: MIT Press, 1991.

Sources for this book are available for downloading.

If you're learning about Macintosh programming

If you want to know more about programming the Macintosh, Apple Developer University offers excellent self-paced courses for novice and intermediate Macintosh programmers, including courses on Macintosh Programming Fundamentals and Introduction to Object-Oriented Programming. "Live" courses on other subjects are also offered through Developer University.

The standard reference for Macintosh programming is *Inside Macintosh*, published by Addison-Wesley and also available through APDA (see the later section on APDA). *Inside Macintosh* is the complete documentation on the interfaces to the Macintosh internal routines. It is available on paper, online, and on several CD-ROMs.

Depending on the development work you are doing, you may not need to use *Inside Macintosh* as a reference. However, *Inside Macintosh* is indispensable for programming all low-level Macintosh features.

If you're updating from a previous version of Macintosh Common Lisp

If you're not familiar with the Common Lisp Object System, read Appendix C, "The Common Lisp Object System." Appendix D, "Converting Your Files to CLOS," is a guide to converting your code to the new object standard. (The experience of early MCL users is that converting code is not nearly so daunting as thinking about it.)

Remember that Macintosh Common Lisp is documented in the *Macintosh Common Lisp Reference*, but Common Lisp itself is documented in *Common Lisp: The Language*.

If you'd like examples of MCL programming

The Examples and Library folders of Macintosh Common Lisp provides examples of MCL programming. Other examples and continuing discussions of Macintosh Common Lisp and Lisp programming issues take place through the MCL online discussion group, `info-mcl`. Third-party code for other MCL tools and extensions is also stored on `info-mcl` and can be downloaded. See the next section.

If you'd like to communicate with other MCL programmers

Apple supports MCL developers through an Internet discussion group and Usenet group, `info-mcl`. The `info-mcl` mailing list posts specific technical questions and their answers and provides a forum in which MCL developers can communicate with each other. It also provides an archive and a library of extensions, third-party code, and patches, which are available for downloading through anonymous FTP. Many programmers find this an extremely valuable feature of Macintosh Common Lisp.

You may send your own code for posting on `info-mcl`. (Apple reserves the right to distribute this code to other users unless specifically requested not to.)

Access to the `info-mcl` bulletin board is available through online services that provide a mail gateway to Internet, such as CompuServe.

The `info-mcl` archives are available on AppleLink, under "Developer Support: Developer Talk: Macintosh Developer Tool Discussion: Macintosh Common Lisp Discussion."

Communicating through Internet

To sign up for `info-mcl`, send a request to

```
info-mcl-request@cambridge.apple.com
```

Send messages to `info-mcl` at the address

```
info-mcl@cambridge.apple.com
```

Communicating through Usenet

To sign up for Usenet, get into rn in UNIX and give the command

```
g comp.lang.lisp.mcl
```

Send messages through mail to info-mcl, or through rn.

Communicating through AppleLink

You can communicate with Internet through AppleLink. To sign up for info-mcl, send a request to the AppleLink address

```
info-mcl-req@cambridge.apple.com@internet#
```

Send messages to info-mcl at the AppleLink address

```
info-mcl@cambridge.apple.com@internet#
```

Communicating through CompuServe

Sign up for info-mcl by sending mail to

```
>INTERNET: info-mcl-request@cambridge.apple.com
```

Send messages to info-mcl from CompuServe at the following address:

```
>INTERNET: info-mcl@cambridge.apple.com
```

Communicating through other services

Some other communications services have links with Internet, and more are being developed. Please check with the Customer Services group at your preferred communications service.

Bugs and bug fixes

If you find a bug in Macintosh Common Lisp, you can report it through the mailing list bug-mcl, reachable in the same way as info-mcl. This list is one-way. Fixes of general interest are distributed through info-mcl. Apple reserves the right to redistribute any sample code or bug fixes that are sent to bug-mcl.

Infrequent announcements of general interest

The mailing list `announce-mcl` is a low-volume mailing list for major announcements about MCL. It is for people who want to find out only about new releases and technical notes. It is reachable in the same way as `info-mcl`.

If you're a developer—APDA

APDA, Apple's source for developer tools, offers convenient worldwide access to over three hundred development tools, resources, and training products and to information for anyone interested in developing applications on Apple platforms. Customers receive a quarterly *APDA Tools Catalog* featuring the most current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. APDA offers convenient payment and shipping options, including site licensing.

To order products or get additional information, contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014-6299 USA

800-282-2732 (United States)
800-637-0029 (Canada)
408-562-3910 (International)
Fax: 408-562-3971
Telex: 171-576
AppleLink address: APDA

Index

A

Abelson, Harold 765
abort-break function 349
Abort menu item 11, 17
About Macintosh Common Lisp menu item 14
accessor methods 671
add-dialog-items. *See* add-subviews generic function
Add Horizontal Guide menu item 288
add-key-handler generic function 382
add-menu-items generic function 98
add-modeline generic function 54
add-points function 68
add-self-to-dialog (obsolete function) 202. *See* install-view-in-window generic function
add-subviews generic function 136, 189
add-to-killed-strings function 501
Add Vertical Guide menu item 288
advisedp macro 368
advise macro 366
advising 366–368
aedesc record type 416
aedisposedesc trap 416
ae-error macro 415
ae-error-str macro 415
aeinteractwithuser trap 417
alert, creating 187
Alert icons 250
allow-returns-p generic function 215, 476
allow-tabs-p generic function 215, 476
APDA 769
appleevent-error condition 414
appleevent-idle function 417
Apple events 411–423
  application variable 414
  call installed queued reply handler 419
  check required parameters 416
  deinstall handler 418
  errors 414, 415
Apple events *(continued)*
  events with indefinite extent 416
  extract Lisp path from FSSPEC 417
  install handler 417
  install queued reply handler 418
  no queued reply handler 419
  Open Application handler 420
  Open Documents handler 422
  Print Documents handler 422
  Quit Application handler 421
  wait state 417
Apple menu 14, 89, 119
application class 414
applications, creating 648–656
apropos function 340
apropos-list function 341
Apropos menu item 18
arcs
  erase 739
  fill 740
  invert pixels 739
  paint 738
arglist function 342, 356
argument lists
  displaying 36
  printing 43, 338
arguments, initialization (initargs) 663
array elements 626
arrays 626–628
ash (common Lisp function) 625
ask (obsolete ObjectLISP function), equivalent 664
always-eval-user-defvars variable 644
application variable 414
arglist-on-space variable 21, 23, 36
arrow-cursor variable 179, 405
autoload-lisp-package variable 21, 629
autoload-traps variable 21, 567

background-event-ticks variable 397, 398
background-sleep-ticks variable 397
backtrace-on-break variable 22, 355
black-color variable 269
black-pattern variable 83
black-rgb variable 269
blue-color variable 269
break-on-errors variable 22, 354
break-on-warnings variable 22, 354
brown-color variable 269
check-call-next-method-with-args variable 619, 620
clear-mini-buffer variable 36, 502
color-available variable 263
compile-definitions variable 358, 637
comtab variable 470, 507, 512
control-x-comtab variable 512
current-character variable 507, 511
current-event variable 377, 394, 395
current-keystroke variable 379, 511
current-view variable 129
cursorhook variable 129, 402, 404, 405
dark-gray-color variable 269
dark-gray-pattern variable 83
dark-green-color variable 269
default-menubar variable 90
default-pathname-defaults variable 315
emacs-mode variable 22, 36, 39
eventhook variable 394, 396
fasl-save-definitions variable 336
fasl-save-doc-strings variable 336
fasl-save-local-symbols variable 22, 43, 336, 338
features variable (Common Lisp) 370
font-list variable 82
foreground-event-ticks variable 397, 398
foreground-sleep-ticks variable 397
fred-default-font-spec variable 36, 167
fred-history-length variable 471
fred-keystroke-hook variable 510
fred-special-indent-alist variable 483
gray-color variable 269
gray-pattern variable 83
green-color variable 269
help-output variable 43, 346
i-beam-cursor variable 179, 405
idle-sleep-ticks variable 396
idle variable 394
  defined 396
  using 655
inspector-disassembly variable 371
killed-strings variable 33
last-command variable 496, 510, 511
light-blue-color variable 269
light-gray-color variable 269
light-gray-pattern variable 83
lisp-cleanup-functions variable 651
.lisp-pathname variable 308
lisp-startup-functions variable 652
listener-comtab variable 512
listener-default-font-spec variable 5, 167
listener-history-length variable 471
listener-window-position variable 5
listener-window-size variable 6
loading-file-source-file variable 645
load-verbose variable 22
logical-directory-alist variable 330, 331
logical-pathname-alist obsolete variable. *See* *logical-directory-alist*
make-package-use-defaults variable 628
menubar-bottom variable 84
menubar-frozen variable 104
menu-id-object-alist variable 104
menubar variable 89
mini-buffer-font-spec variable 36
mini-buffer-help-output variable 43, 338
modal-dialog-on-top variable 259
module-file-alist variable 309
module-search-path variable 309
modules variable 310
mouse-view variable 129
multi-click-count variable 46
next-screen-context-lines variable 38, 44, 45
  defined 37
open-file-streams variable 324

orange-color variable 269
package variable (Common Lisp) 370
paste-with-styles variable 23, 33, 37
pathname-translations-pathname variable 309
pen-modes variable 82
pink-color variable 269
pixels-per-inch-x variable 84
pixels-per-inch-y variable 84
print-abbreviate-quote variable 25
print-simple-bit-vector variable 629
print-simple-vector variable 629
print-string-length 629
print-structure variable 629
purple-color variable 269
readtable variable (Common Lisp) 370
record-source-file variable 18, 23, 42, 336, 338
record-types variable 593
red-color variable 269
save-definitions variable 21, 23, 36, 336, 358
save-doc-strings variable 23, 43, 337, 339
save-exit-functions variable 651
save-fred-window-positions variable 23, 37
save-local-symbols variable 23, 43, 337, 338
save-position-on-window-close variable 37
scrap-handler-alist variable 408
scrap-state variable 407, 408
screen-height variable 83
screen-width variable 83
show-cursor-p variable 480
step-print-length variable 359
step-print-level variable 359
style-alist variable 83
tan-color variable 269
top-listener variable 5
trace-bar-frequency variable 365
trace-level variable 364
trace-max-indent variable 364
trace-print-length variable 364
trace-print-level variable 364
update-cursor function 404
verbose-eval-selection variable 24
warn-if-redefine-kernel variable 337
warn-if-redefine variable 24, 337
watch-cursor variable 180, 405
white-color variable 269
white-pattern variable 83
white-rgb variable 269
windoid-count variable 174, 184
window-default-position variable 164, 173
window-default-size variable 164, 173
window-default-zoom-position variable 176
window-default-zoom-size variable 178
yellow-color variable 269
@ (at symbol) variable 372
auxiliary methods 678

B

backtrace 19, 346, 355–358
Backtrace menu item 19
Balloon Help xxiii
base character 622
beep 493
bibliography 763–769
BIGNUMP function 625
bitmaps
  copy bits 750
  create 749
  scrolling rectangle 750
break function 353
break loop 349–355
  MCL forms associated with 353
  MCL functions associated with 355
Break menu item 11, 18, 352
Brooks, Rodney 764
buffer-bwd-sexp function 463
buffer-capitalize-region function 459
buffer-char-font. *See* buffer-char-font-spec function
buffer-char-font-spec function 189, 465
buffer-char function 455
buffer-char-pos function 460
buffer-char-replace function 455
buffer-column function 454
buffer-current-font-spec function 466

buffer-current-sexp-bounds function 459
buffer-current-sexp function 457
buffer-current-sexp-start function 458
buffer-delete function 459
buffer-downcase-region function 459
buffer-fwd-sexp function 462
buffer-getprop function 451
buffer-get-style function 467
buffer-insert-file function 463, 465
buffer-insert function 456
buffer-insert-substring function 456
buffer-insert-with-style function 457
buffer-line-end function 454
buffer-line function 453
buffer-line-start function 453
buffer-mark class 446, 447
buffer-mark-p function 189, 447
buffer marks
  definition 443, 446
  MCL expressions relating to 447–469
buffer-modcnt function 450
buffer-next-font-change function 468
buffer-not-char-pos function 460
buffer-plist function 451
buffer-position function 452
buffer-previous-font-change function 468
buffer-putprop function 452
buffer-remove-unused-fonts function 465, 467
buffer-replace-font. *See* buffer-replace-font-spec function
buffer-replace-font-codes function 469
buffer-replace-font-spec function 189, 466
buffers 443–445
  definition 443, 445
  evaluate 17
  select, in Fred 47
buffer-set-font. *See* buffer-set-font-spec function
buffer-set-font-codes function 469
buffer-set-font-spec function 189, 466
buffer-set-style function 468
buffer-size function 450
buffer-skip-fwd-wsp&comments function 189, 463
buffer-string-pos function 461
buffer-substring function 456
buffer-substring-p function 461
buffer-upcase-region function 459
buffer-word-bounds function 462
buffer-write-file function 464
built-in-class class 607
button-dialog-item class 190, 206
button dialog items, creating instances 206
buttons 206–207, 208
  default 207, 208, 209
  radio 218–221

C

caches 618–619
call-next-method function 619, 620, 670
Cancel button 258
cancel function 349
cancel-text keyword 251
caps-lock-key-p function 393
catch-abort. *See* restart-case
catch-cancel macro 249
catch-error. *See* handler-case
catch-error-quietly. *See* ignore-errors function
Caution icon 250
ccl::*name-char-alist* variable 626
ccl::compile-time-class class 607
ccl::std-class class 607
cell-contents generic function 225
cell-deselect generic function 231
cell-font generic function 229
cell-position generic function 233
cell-selected-p generic function 231
cell-select generic function 231
cell-size generic function 229
cell-to-index generic function 245
change-key-handler generic function 383
characters 626
  capitalize 51
  literal 50
  lowercase 50
  outputting from stream 428
  quoted 50
  read from stream 429
  unread from stream 430
  uppercase 50
check-box-checked-p generic function 217
check-box-check generic function 218
check-box-dialog-item class 190, 216

checkbox dialog items 216–218
creating instances 217
check-box-uncheck generic function 218
check-required-params function 416
check-type macro, not inline version of 644
choose-directory-dialog function 329
choose-file-default-directory function 329
choose-file-dialog function 327
choose-new-file-dialog function 328
C language
calling Macintosh Common Lisp 532
calling sequence 560
with Foreign Function Interface 533
records of type AUTOMATIC 542
class allocation 666
class class 607
class-class-slots generic function 612
class-direct-class-slots generic function 612
class-direct-instance-slots generic function 611
class-direct-subclasses generic function 609, 610
classes 660–664
class slots 612
direct class slots 612
direct instance slots 611
direct subclasses 609
direct superclasses 610
hierarchy 621
instance slots 612
precedence list 621
precedence list of methods specialized on class 610
prototype instance 611
class-instance-slots generic function 612
class precedence list 668, 674
class-precedence-list generic function 610
class-prototype generic function 611
clear-all-gf-caches function 619
clear-clos-caches function 618, 619
clear generic function 16, 116, 181, 216, 494
clear-gf-cache function 618
Clear menu item 16
clear-record macro 591
clear-specializer-direct-methods-caches function 618
Clipboard 446
clip-rect generic function 712
clip-region generic function 711
CLOS (common Lisp Object System) 658–684
CLOS compatibility xxii
close-region generic function 743
Close menu item 15
code definition for symbol, examining 42, 43, 338
collapse-selection generic function 487
:A0–A4 keyword 527, 529, 602, 603
:A5 keyword 529
:action-function keyword 253, 254
:after keyword 361
:allocation options 666
:allow-constant-substitution keyword 642
:allow-empty-strings keyword 253
:allow-returns keyword 476
:allow-tabs keyword 476
:allow-tail-recursion-elimination keyword 640
:allow-transforms keyword 642
:auto-update-default keyword 235
:before keyword 360
:body keyword 192, 199, 275, 477
:bold keyword 69
:boolean keyword 599, 602, 603
:border-p keyword 207
:bottom keyword 162
:buffer-chunk-size keyword 477
:button-string keyword 328
:by-address keyword 525
:by-reference keyword 525
:by-value keyword 525
:cancel keyword 258
:cancel-text keyword 252
:case keyword 305
:cell-fonts keyword 225
:cell-size keyword 225
:centered keyword 162
:char keyword 526
:check-args keyword 522, 526
:check-box-checked-p keyword 217
:check-error keyword 597, 600
:chunk-size keyword 449
:class keyword 157, 158, 159, 189
:close-box-p keyword 155, 474
:closed keyword 258
:color keyword 268

:color-p keyword 155, 189, 473
:command-key keyword 107, 117
:comtab keyword 470, 474, 477
:condense keyword 69
:content keyword 170, 275
:copy-styles-p keyword 471, 473, 477
:cstring keyword 524
:D0 keyword 599
:D0–D7 keyword 527, 529, 530, 602, 603
:default-button keyword 207
:default-item keyword 235
:default-menu-background keyword 92, 274
:default-menu-item-title keyword 92, 101, 274
:default-menu-title keyword 92, 274
:defaults keyword 305
:device keyword 303
:dialog item-colors 189. *See also* :part-color-list keyword
:dialog-item-action keyword 192, 194, 236
:dialog-item-enabled-p keyword 192, 477
:dialog-item-handle keyword 192
:dialog-item-text keyword 192, 211, 235, 477
:direction keyword 237, 427
:directories keyword 314
:directory keyword 303, 328, 329
:directory-pathnames keyword 314
:disabled keyword 107, 118
:document keyword 155, 473
:document-with-grow keyword 155, 473
:document-with-zoom keyword 155, 473
:do-it keyword 367
:double-edge-box keyword 155, 473
:double keyword 523, 526
:draw-outline keyword 477
:entry-names keyword 519
:erase-anonymous-invalidations keyword 139, 156
:extended keyword 523
:extend keyword 69
:ffenv-name keyword 520
:filename keyword 472, 489
:files keyword 314
:float keyword 526
:font keyword 449
:force-boundp-checks keyword 642
:fork keyword 322
:frame keyword 170, 192, 199, 275, 477
:full-long keyword 526
:handle keyword 576
:help-spec keyword 94, 107, 118, 127, 129, 192, 236, 238, 251, 474
:hilite keyword 170, 275
:history-length keyword 471, 474
:host keyword 303
:if-does-not-exist keyword 318
:if-exists keyword 319, 321, 322
:include-invisibles keyword 157, 158, 159
:include-windoids keyword 157, 158, 159
:inhibit-event-polling keyword 641
:inhibit-register-allocation keyword 640
:inhibit-safety-checking keyword 641
:initial-string keyword 252
:inline-self-calls keyword 641
:italic keyword 69
:item-display keyword 235
:item-key keyword 114, 275
:item-mark keyword 114, 275
:item-title keyword 114, 275
:language keyword 522
:left keyword 162
:libraries keyword 519
:library-entry-names keyword 519
:lisp-ref keyword 524
:long keyword 523, 526, 529, 530, 598, 599, 602, 603
:mac-file-creator keyword 319
:mac-file-type keyword 319, 328
:max keyword 237
:menu-background keyword 101, 274
:menubar keyword 92, 274
:menu-colors keyword 94, 235
:menu-item-action keyword 107, 118
:menu-item-checked keyword 107, 118
:menu-item-colors keyword 107, 118
:menu-items keyword 94, 235
:menu-item-title keyword 106, 117
:menu-title keyword 94, 101, 274
:method keyword 360, 363, 366, 368
:min keyword 237
:modeless keyword 252, 254, 255
:name keyword 304
:no-check-arg keyword 526
:no-error keyword 315

:no-text keyword 250
:novalue keyword 526, 530, 599, 603
:ok-text keyword 249, 252
:open-code-inline keyword 641
:origin keyword 579
:ostype keyword 599
:outline keyword 69
:owner keyword 106
:page-size keyword 237
:pane-splitter keyword 237
:parent 189. *See also* :color-p keyword
:part-color-list keyword 189, 192, 199
:plain keyword 69
:pointer keyword 576
:position keyword 249, 250, 252, 268
:procid keyword 155, 473
:prompt keyword 268, 328
:pstring keyword 524
:ptr keyword 523, 526, 529, 530, 599, 602, 603
:radio-button-pushed-p keyword 219
:read-only keyword 449
:replace keyword 520
:resolve-aliases keyword 314
:return-block keyword 529, 602
:reverse-args keyword 522
:right keyword 162
:scrollee keyword 237
:scroll-size keyword 237
:selection-type keyword 224, 254
:sequence-order keyword 244
:sequence-wrap-length keyword 243, 244
:setting keyword 237
:shadow keyword 69
:shadow-edge-box keyword 155, 473
:single-edge-box keyword 155, 473
:size keyword 249, 250, 252
:srcbic keyword 69
:srccopy keyword 69
:srcor keyword 69
:srcpatbic keyword 69
:srcpatcopy keyword 69
:srcpator keyword 69
:srcpatxor keyword 69
:srcxor keyword 69
:stack keyword 527
:step keyword 361
:style keyword 107, 118
:table-dimensions keyword 224
:table-hscrollp keyword 225
:table-print-function keyword 224, 225, 226, 254
:table-sequence keyword 243
:table-vscrollp keyword 224
:test keyword 314
:text-edit-sel-p keyword 477
:text keyword 192, 199, 275, 477
:thumb keyword 192, 199, 275
:title-bar keyword 170, 275
:title keyword 170
:tool keyword 155, 473
:top keyword 162
:track-thumb-p keyword 237
:trap-modifier-bits keyword 599, 600
:trust-declarations keyword 640
:type keyword 304
:underline keyword 69
:update-function keyword 94, 107, 118
:version keyword 304
:view-container keyword 127, 129, 191, 472, 475
:view-font keyword 127, 128, 154, 192, 472, 476
:view-nick-name keyword 127, 128, 154, 191, 472, 476
:view-position keyword 127, 128, 154, 191, 472, 476
:view-scroll-position keyword 128, 154, 472
:view-size keyword 127, 128, 154, 184, 191, 472, 476
:view-subviews keyword 129, 154, 472
:visible-dimensions keyword 225
:wild keyword 303
:wild-inferiors keyword 303
:window-do-first-click keyword 155, 184, 474
:window-layer keyword 155, 473
:window-show keyword 154, 473
:window-title keyword 154, 254, 473
:window-type keyword 155
:word keyword 523, 526, 529, 530, 598, 599, 602, 603
:wptr keyword 127, 156, 193, 474
:wrap-p keyword 472
:yes-text keyword 250
color-blue function 265
color-green function 264
Color Picker 262, 268

`color-red` function 264
colors 261, 262–276
  blue 265
  Color Picker 262, 268
  Color QuickDraw 263
  component values 265
  dialog items 275
  display as same color 265
  encoding 262
  green 264
  implementation 261
  menu bars 91, 274
  menu items 275
  menus 274
  red 264
  returning encoded color 263
  RGB records 266, 267, 269
  values 265
  windows 270–272, 275
`color-to-rgb` function 266
`color-values` function 264, 265
Color Window 290
`color-window-mixin`. *See* `:color-p` keyword
`command-key` generic function 110
`command-key-p` function 393
command keystrokes 9
Command-Meta-click 43
command tables 504
  definition 505
Common Lisp Object System (CLOS) 658–684
COMMON LISP: THE LANGUAGE 659
compatibility
  implementation of file system in MCL version 2 297
  MCL version 2 and CLOS xxii
  between MCL version 2 and earlier versions 628, 659
  MCL version 2 and Metaobject Protocol xxii
  updating from previous versions 766
compilation 7, 10, 638–643
  compiler policy objects 638
  eliminating tail recursion 638
  options 335–337
  self-referential calls 638
compiled file (.FASL file) 10
Compile File menu item 17
`compiler-policy` class 639
compiler policy objects 638
`compute-applicable-methods` generic function 619
comtab 507, 512
  definition 505
  shadowing 507
`comtab-find-keys` function 515
`comtab-get-key` function 514
`comtab-key-documentation` function 515
`comtab-set-key` function 513
`comtabp` function 513
`configure-egc` function 634
container 132
  definition of 123
context lines 37
continue 18
`continue` function 354
Continue menu item 18
Control-A keystroke 44
Control-B keystroke 44
Control-D keystroke 52
Control-E keystroke 44
Control–equal sign keystroke 42, 339
Control-F keystroke 44
Control-G keystroke 4, 60
Control key 39
`control-key-p` function 393
Control-K keystroke 52
Control–Left Arrow keystroke 44
Control-Meta-A keystroke 44
Control-Meta-B keystroke 44
Control-Meta–close parenthesis keystroke 45
Control-Meta-Delete keystroke 52
Control-Meta-E keystroke 44
Control-Meta-F keystroke 44
Control-Meta-H keystroke 47
Control-Meta-K keystroke 52
Control-Meta-L keystroke 56
Control-Meta-N keystroke 4, 45
Control-Meta-number keystroke 57
Control-Meta-O keystroke 49
Control-Meta–open parenthesis keystroke 45
Control-Meta-P keystroke 45
Control-Meta-Q keystroke 49
Control-Meta-semicolon keystroke 53, 55
Control-Meta-Shift–Down Arrow keystroke 48
Control-Meta-Shift-M keystroke 54
Control-Meta-Shift-N keystroke 48
Control-Meta-Shift-P keystroke 48
Control-Meta–Shift–Up Arrow keystroke 48

Control-Meta–Space bar keystroke 47
Control-Meta-T keystroke 51
Control-Meta-underscore keystroke 56
Control-M keystroke 53, 339
Control modifier 39
Control-N keystroke 44
Control-number keystroke 57
Control-O keystroke 49
Control-P keystroke 44
Control-Q keystroke 50, 60
Control–question mark keystroke 42, 338
Control-Return keystroke 49
Control-Right Arrow keystroke 44
Control-R keystroke 60
Control-S Control-W keystroke 60
Control-S Control-Y keystroke 60
Control-Shift-A keystroke 47
Control-Shift-E keystroke 47
Control-Shift–Left Arrow keystroke 46
Control-Shift-N keystroke 47
Control-Shift-P keystroke 47
Control-Shift–Right Arrow keystroke 47
Control-Shift-V keystroke 48
Control-S keystroke 60
Control-S Meta-W keystroke 60
Control–Space bar keystroke 51
Control-Tab keystroke 45
Control-T keystroke 51
Control-U keystroke 57
Control-V keystroke 45
Control-W keystroke 49, 53, 60
Control-X Control-A keystroke 43, 338
Control-X Control-C keystroke 54
Control-X Control-D keystroke 43, 339
Control-X Control-E keystroke 54
Control-X Control-F keystroke. *See* Control-X Control-V keystroke
Control-X Control-I keystroke 43, 339, 369
Control-X Control-M keystroke 54, 339
Control-X Control-R keystroke 54, 339
Control-X Control-S keystroke 56
Control-X Control-V keystroke 56
Control-X Control-W keystroke 56
Control-X Control-X keystroke 48, 51
Control-X H keystroke 47
Control-X semicolon keystroke 55
Control-X U keystroke 56
Control-Y keystroke 49, 60
`convert-coordinates` function 151
`copy-bits` function 750
Copy command 33
`copy-comtab` function 513
`copy-file` function 322
`copy` generic function 16, 116, 181, 216, 372, 494
`copy-instance` generic function 617
Copy menu item 16
`copy-record` macro 592
`copy-region` function 742
`create-file` function 318
creating instances
  button dialog items 206
  checkbox dialog items 217
  default button dialog items 208
  dialog items 191
  editable-text dialog items 211
  floating windows 184
  Fred dialog items 475
  Fred windows 472
  menu items 106
  menus 94
  pop-up menus 235
  radio-button dialog items 219
  scroll-bar dialog items 236
  sequence dialog items 243
  simple views 127
  static-text dialog items 210
  table dialog items 224
  views 128
  window menu items 117
  windows 153
`current-compiler-policy` function 642
`current-file-compiler-policy` function 643
current expression, definition 41
`current-key-handler` generic function 211, 381
cursors 401
  arrow 405
  GC cursor 13
  I-beam 405
  setting shape 403, 404
  updating 404
  updating shape 402
  visibility, checking 480
  watch 405
cursors determining shape 402
customizing MCL behavior 20, 25
Cut command 33

cut generic function 16, 116, 181, 216, 372, 494
Cut menu item 16

D

dead keys, in Fred 40
debugging 333–373
  MCL functions related to 340
declaration-information function 640
debugging commands in Fred 337–340
default-button-dialog-item class 208
default-button dialog items 208
default-button generic function 208
default-button-p generic function 209
default buttons 207–209
default values (initforms) 663
defccallable macro 560
deffcfun macro 521
defffun macro 521
deffpfun macro 521
def-fred-comtab macro 504
defgeneric macro 668
definitions, editing 9
definitions, listing 19
def-load-pointers macro 652
def-logical-directory function 331
def-logical-pathname (obsolete function) 331
defpascal macro 560
defmethod macro 668
defrecord macro 576
defstruct macro 645
deftrap macro 569
deinstall-appleevent-handler function 418
delete 496
delete-file function 318
Delete keystroke 52, 60
deletion 52
descent 73, 76
Design Dialogs menu item 288
dialog boxes 187, 289
  Document-with-Grow 289
dialog class 256
dialog item-default-size. *See* view-default-size generic function
dialog-item-action-function generic function 195
dialog-item-action generic function 194
  check-box-dialog-item 217
dialog-item class 191
dialog-item-dialog. *See* view-container generic function
dialog-item-disable generic function 201
dialog-item-enabled-p generic function 201
dialog-item-enable generic function 200
dialog-item-font. *See* view-font generic function
dialog-item-handle generic function 203
dialog-item-nick-name. *See* view-nick-name generic function
dialog-item-position. *See* view-position generic function
dialog items 190–259
  activate event handler 203
  add 290
  add to window 202
  Alert icons 250
  associated function 194, 195
  button 206
  Caution icon 250
  checkboxes 216–218
  color of parts 199
  color of parts, returning 200
  color of parts, setting 200
  colors 275
  creating instances 191
  deactivate event handler 204
  default button 208, 209
  default size 204
  default width correction 205
  definition 186, 190
  disable 201
  editable-text 210–216
  enable 200
  enabled, checking if 201
  event handling 195
  find 259
  find named sibling 135
  find named subview 135
  focused 205
  focus on container and draw contents 196
  font 198
  font codes 77
  font codes set 78
  font setting 198
  font specifiers 198

dialog items *(continued)*
font specifiers setting 199
functions, advanced 201
GrafPort, use specific 205
graphic dialog items 250
handle 203
items in view 193
MCL functions related to dialog items 190–205
Note icon 250
printing 493
radio buttons 218–221
remove from window 202, 203
sample files 201
`sequence-dialog-item` 243
size 196
size setting 196
specialized 206
static-text
Stop icon 250
table dialog items 221–234
text 197
text setting 197
`dialog-items` generic function 193
`dialog-item-size`. *See* `view-size` generic function
`dialog-item-text` generic function 197, 211
`dialog-item-width-correction` generic function 205
dialogs 124–189, 246–259, 286–293
cancel 248
changed dialog functions 188–189
changes in Macintosh Common Lisp version 2 188
creating 288
creating with Designer 287
definition 124, 256
display-message turnkey dialog 249
editing 287
functionality 187
get-string-from-user turnkey dialog 252
modal 246
modeless 246
select-item-from-list turnkey dialog 254
yes-or-no turnkey dialog 250
`difference-region` function 745
directories
default directory 329
default directory set 329
Macintosh default 311
MCL operations related to directories 318–321
structured 313
`directory` function 314
`directory-pathname-p` function 316
`directoryp` function 315
`disassemble` function 356
disk
eject 325
eject, checking 325
eject and unmount 325
`disk-ejected-p` function 325
`displaced-array-p` function 627
`dispose-ffenv` function 521
`dispose-record` macro 416, 584
`dispose-region` function 742
documentation 18
conventions xxv–xxxi
online xxiii
`documentation` generic function 343, 356
documentation types 344
Documentation menu item 18
`$wmggport` constant 595
Document dialog box 289
Document-with-Grow dialog box 289
Document-with-Zoom dialog box 289
`do-subviews` macro 134
`double-click-p` function 392
`double-click-spacing-p` function 392
Double-Edge-Box dialog box 289
double quotation mark 50
`draw-cell-contents` generic function 225, 226
`draw-menubar-if` function 105
`draw-picture` generic function 753
`drive-name` function 326
`drive-number` function 327

E

`ed-arglist` generic function 43, 338
`ed-back-to-indentation` generic function 45
`ed-backward-char` generic function 44
`ed-backward-select-char` generic function 46

ed-backward-select-sexp generic function 46
ed-backward-select-word generic function 46
ed-backward-sexp generic function 44
ed-backward-word generic function 44
ed-beep function 493
ed-beginning-of-buffer generic function 45
ed-beginning-of-line generic function 44
ed-bwd-up-list generic function 45
ed-capitalize-word generic function 51
ed-copy-region-as-kill generic function 53
ed-current-sexp generic function 484
ed-current-symbol generic function 483
ed-delete-char generic function 52
ed-delete-forward-whitespace generic function 53
ed-delete-horizontal-whitespace generic function 53
ed-delete-whitespace generic function 53
ed-delete-with-undo generic function 497, 498
ed-delete-word generic function 52
ed-downcase-word generic function 50
ed-edit-definition generic function 42, 338
ed-end-of-buffer generic function 45
ed-end-of-line generic function 44
ed-end-top-level-sexp generic function 44
ed-eval-current-sexp generic function 54
ed-eval-or-compile-current-sexp generic function 54
ed-eval-or-compile-top-level-sexp generic function 54
ed-exchange-point-and-mark generic function 48, 51
ed-forward-char generic function 44
ed-forward-select-char generic function 46
ed-forward-select-sexp generic function 47
ed-forward-select-word generic function 46
ed-forward-sexp generic function 44
ed-forward-word generic function 44
ed-fwd-up-list generic function 45
ed-get-documentation generic function 43, 339
ed-help function 42, 338
ed-history-undo generic function 56
ed-indent-comment generic function 55
ed-indent-differently generic function 45
ed-indent-for-lisp generic function 49
ed-indent-sexp generic function 49
ed-insert-char generic function 481
ed-insert-double-quotes generic function 50
ed-insert-sharp-comment generic function 50
ed-insert-with-style generic function 482
ed-inspect-current-sexp generic function 43, 339
ed-i-search-forward generic function 60
ed-i-search-reverse generic function 60
editable-text-dialog-item class 190, 210
editable-text dialog items 210–216, 445
  creating instances 211
edit-definition function 356
Edit Definition menu item 18
edit-definition-p function 344
Edit Dialog menu item 288
editing definition of source code 18
Edit menu 16
Edit Menubar menu item 287
editor 7–8
  evaluating from 10
  format to use with other editors 9
edit-select-file generic function 56
ed-kill-backward-sexp generic function 52
ed-kill-comment generic function 53, 55
ed-kill-forward-sexp generic function 52
ed-kill-line generic function 52
ed-kill-region generic function 53
ed-kill-selection generic function 501
ed-last-buffer generic function 56
ed-macroexpand-current-sexp generic function 54
ed-macroexpand-1-current-sexp generic function 54
ed-move-over-close-and-reindent generic function 45

ed-newline-and-indent generic function 49
ed-next-line generic function 44
ed-next-list generic function 45
ed-next-screen generic function 45
ed-numeric-argument generic function 57
ed-open-line generic function 49
ed-previous-line generic function 44
ed-previous-list generic function 45
ed-previous-screen generic function 44
ed-print-history generic function 56
ed-push/pop-mark-ring generic function 51
ed-read-current-sexp generic function 54, 339
ed-rubout-char generic function 52
ed-rubout-word generic function 52
ed-select-beginning-of-line generic function 47
ed-select-current-sexp generic function 47
ed-select-end-of-line generic function 47
ed-select-next-line generic function 47
ed-select-next-list generic function 48
ed-select-next-screen generic function 48
ed-select-previous-line generic function 47
ed-select-previous-list generic function 48
ed-select-previous-screen generic function 48
ed-select-top-level-sexp generic function 47
ed-self-insert generic function 511
ed-set-comment-column generic function 55
ed-skip-fwd-wsp&comments. *See* buffer-skip-fwd-wsp&comments function
ed-split-line generic function 49
ed-start-top-level-sexp generic function 44
ed-transpose-sexps generic function 51
ed-transpose-words generic function 51
ed-universal-argument generic function 57
ed-upcase-word generic function 50
ed-what-cursor-position generic function 42, 339
ed-yank generic function 49
ed-yank-pop generic function 49
effective method 679
egc-active-p function 634
egc-configuration function 634
egc-enabled-p function 634
egc function 12, 633
eject&unmount-disk function 325
eject-disk function 325
ELT (common Lisp function) 245
Emacs 7
empty-rect-p function 726
empty-region-p function 747
enter-key-handler generic function 213
Enter keystroke 54
Environment menu item 19
environment options 24
ephemeral garbage collection 12
  activating 633
  configuring 634
  counting invocations 635
  definition 632
  enabling 633
  generation 632
  timing 635
equal-rect function 725
equal-region-p function 747
erase-arc generic function 739
erase-oval generic function 731
erase-polygon generic function 756
erase-rect generic function 728
erase-region generic function 748
erase-round-rect generic function 735
errors 333–373
  handling 348–355
Escape key as Meta modifier 40
Eval Buffer menu item 17
eval-enqueue function 399, 655, 656
Eval menu 17
Eval Selection menu item 17
evaluate
  buffer 17
  selection 17
evaluating expressions 6
evaluating from editor 10
evaluation 6
  in Fred 6, 8
evaluator
  compiling 637
  standard 637

event ticks 396, 397, 398
event-dispatch function 388, 394, 655
event handler 394
  activate view 380
  click in view 377, 387
  deactivate view 380
  disable event processing during execution of forms 401
  mouse 403
  mouse button up 385
  null event 384
  queue form for evaluation 399, 656
  release key 384
  release mouse button 385
  update window 388
  view activate 380
  view deactivate 380
  window 398
  window key up 384
  window no event 384
  window select 384
  window update 388
  window zoom 386
event-keystroke function 508
events 376–410
  idling 655
  interrupts 655
  queued 655
  response time 655
  time-consuming events 655
event-ticks function 388, 397
examples of code xxiii, 30, 33, 40, 41, 214, 223
  Apple Events 423
  application programming 277
  Boyer-Moore search algorithm 61
  calling traps 570
  CLOS 658–684
  CLOS conversion 685–704
  :pict scrap handler 408
  communicating with HyperCard 423
  defining a scrap handler 406
  defrecord macro 577
  dialog boxes with Alert icons 250
  dialog functions, obsolete 188
  dialog items 187, 201
  dialog items, implementing custom class 246
  ephemeral garbage collector 632
  font menus 7, 120
examples of code *(continued)*
  Foreign Function Interface 533
    and C language 533
    testing 533
  help balloons 127
  icons 190
  input/output stream 426
  interface building 277
  interface programming 705
  logical pathnames, setting 520
  MCL class hierarchy 621
  menu-item-update methods 119
  menus 87
  Meta key rebound to Escape 7
  pop-up menus 234
  QuickDraw 705
  stack allocation 543
  using class-direct-subclasses 609
  using invalidate-view 139
  using set-view-font 168
  using view-click-event-handler 149
  using view-draw-contents 390
  where to find 767
  windoid event handling 183
execution stack 349
exit 15
exit-key-handler generic function 213
expand-logical-namestring (obsolete function) 315
expressions
  current, definition 41
  evaluating from editor 6
  inspecting 43, 339
  macroexpanding 339
  macroexpanding repeatedly 339
  reading 339
extended keyboard keys 41
extended wildcards 317
extensions
  .fasl (compiled file) 10
  .lisp (source file) 10
  .text (non-Lisp text file) 10
externalize-scrap generic function 409, 410

F

fact 361
failing i-search prompt 58
fasl file (compiled) 10

ff-call function 528
ff-load function 519
ff-lookup-entry function 530
field-info function 574
fields, variant 579
file-create-date function 321
file-locked-p function 323
File menu 15
filenames 295–331
  definition 297
files
  compile 17
  create new file 15
  loading 17, 308
  MCL operations related to files 318–324
  open 15
  revert to last saved version 15
  save 15
  save as 15
  save copy as 15
file-write-date function 321
fill-oval generic function 732
fill-polygon generic function 756
fill-rect generic function 729
fill-region generic function 749
fill-round-rect generic function 736
find-clicked-subview generic function 137
find-dialog-item generic function 259
find-mactype function 573
find-menu function 91
find-menu-item generic function 100
find-named-dialog-items. *See* find-named-sibling generic function, view-named generic function
find-named-sibling generic function 135, 189
find-record-descriptor function 572
find-view-containing-point generic function 146
find-window function 160
floating-point data type 624
floating windows 183
  creating instances 184
  definition 124, 183
  instantiation 184
  number visible 184
fixnump function 625
fixnums 624
flush-volume function 326
focused dialog item 205
focused view 130, 131
focusing, definition of 123
focus-view generic function 129, 130
font codes
  for current GrafPort 79
  for current GrafPort, set 79
  merge 80
  set 80
font-codes-info function 76
font-info function 73
fonts 69–83
  alist of style keywords and numbers 83
  ascent 73, 76
  behavior of insertion font 33
  cell, table dialog item 230
  check for existence 71
  default, in Fred windows 36
  default, in Listener windows 5
  descent 73, 76
  font codes 70
    MCL functions related to 75–82
  font specifications 69–70
    definition 69
    from font codes 71
    MCL functions related to 71–74
  Fred 464
  leading 73, 76
  list of all installed fonts 82
  name 69
  set insertion font 488
  setting 33
  size 69
  style 69
  transfer mode 69
  widmax 73, 76
  width of string 72
  window 166
    default 167
font-spec function 71
force-output function 444
foreign functions 517–533
forms, initial value (INITFORMS) 663
Forward Delete keystroke (extended keyboard) 52
frame-arc generic function 740
frame-oval generic function 730
frame-polygon generic function 755
frame-rect generic function 726
frame-region generic function 748

`frame-round-rect` generic function 733
`fred-mixin` class 680
Fred 7
  Apple Extended Keyboard keys 41
  background color 487
  chunk size 478
  Clipboard 33
  context lines 37
  dead keys 40
  debugging commands 337–340
  default font 36
  deleting 52
  editing, implementation of 443
  Emacs mode 22, 36
  Escape key as Meta modifier 40
  evaluating from 10
  evaluation 6, 8
  font, setting 33
  fonts 33, 464
  foreground color 488
  format to use with other editors 9
  Fred Commands menu item 19
  Fred dialog items 445
  Fred windows 443, 444
  horizontal scroll 486
  horizontal scroll, set 486
  insertion font behavior 33
  kill ring 33
  line wrap checking 491
  Lisp operations 54
  Macintosh editing features 7, 38
  margin return 490
  margin set 491
  Meta and Control modifiers 7
  multiple fonts 33
  programming 441
  set package of windows 34–35
  setting a font 33
  spaces per tab return 491
  styles, retaining 37
  using Macintosh Option character set 40
  window information, saving 37
  windows 124
  yank, rotating 49
`fred-blink-position` generic function 480
`fred-buffer` function 189
`fred-buffer` generic function 450, 478
`fred-chunk-size` generic function 449, 478
Fred commands
  add mode line 54
  capitalize word 51
  comment column, set 55
  comment insert or align 55
  copy current region 53
  current editor window 42, 339
  defining 504
  delete 52, 53
    character left 52
    character right 52
    current region 53
    horizontal white space 53
    s-expression left 52
    s-expression right 52
    to end of current line 52
    to next non-white-space character 53
    white space 53
    word left 52
    word right 52
  documentation of function 43, 339
  evaluate current s-expression 54
  evaluate or compile current sexp 54
  evaluate or compile top-level s-expression 54
  examine code definition for symbol 42
  files 56
  function documentation 339
  help 19, 42
  help commands 42–43
  Help window 338
  incremental search 57–61
    commands 60–61
  insert 49–51
    current kill ring string into buffer 49
    double quotation mark 50
    line 49
    next kill ring string into buffer 49
    parentheses 50
    quoted character 50
    sharp-sign comment 50
  inspect current s-expression 43, 339
  kill comment 53, 55
  Lisp operations 54–55
  lowercase word 50
  macroexpand current s-expression 54, 339
  macroexpand current s-expression repeatedly 339

Fred commands *(continued)*
move 43–45
back character 44
backward s-expression 44
backward word 44
beginning of buffer 45
beginning of current top-level s-expression 44
beginning of line 44
end of buffer 45
end of current top-level s-expression 44
end of line 44
forward character 44
forward s-expression 44
forward word 44
line down 44
line up 44
next screen 45
over next close parenthesis 45
previous screen 44
to beginning of current s-expression 45
to end of current s-expression 45
to next list 45
to previous list 45
numeric arguments 57
print argument list of function 43, 338
push mark onto ring 51
read current s-expression 54, 339
reindent
current line 49
current s-expression 49
for readability 45
save window to file 56
save window to new file 56
select 46–48
backward character 46
backward s-expression 46
backward word 46
beginning of line 47
current s-expression 47
current top-level s-expression 47
end of line 47
entire buffer 47
forward character 46
forward s-expression 47
forward word 46
line down 47
line up 47
Fred commands *(continued)*
next screen 48
previous screen 48
text file and open window 56
split line 49
terminate a command 60
transpose
point and mark 48, 51
s-expressions 51
words 51
undo 56
uppercase word 50
windows 56
yank 49
Fred Commands menu item 19
`fred-copy-styles-p` generic function 482
`fred-dialog-item` class 445, 475
Fred dialog items, creating instances 475
`fred-display-start-mark` generic function 189, 479
`fred` function 474
`fred-hpos` generic function 189, 485
`fred-hscroll` generic function 486
`fred-last-command` generic function 510
`fred-line-vpos` generic function 189, 485
`fred-margin` generic function 490
`fred-mixin` class 443, 470
`fred-package` generic function 490
`fred-point-position` generic function 484
`fred-tabcount` generic function 491
`fred-update` generic function 189, 444, 481
`fred-vpos` generic function 189, 485
`fred-window` class 444, 471
Fred windows,
creating instances 472, 474
`fred-wrap-p` generic function 491
`front-window` function 157, 370
`fsspec` type 417
`full-pathname` function 315
`funcallable-standard-class` class 607
functions
documentation 43, 339
printing argument list 43, 338

G

garbage collection 631, 636–637
ephemeral 12, 632
full 12
`mmu` support 633

gc-event-check-enabled-p function 636
gccounts function 635
gc function 13, 636
gctime function 635
generation, definition 632
generic-function class 607
generic-function-methods generic function 615
generic functions 661–662
  associated with a method 615
  defining new 668
  definition 661, 667
  method exists, checking 619
  methods 615
  methods that run when a generic function is invoked 615
  on given specializer 614
get-back-color generic function 487
get-dead-keys (obsolete MCL function) 41
get-fore-color generic function 488
get-internal-scrap generic function 406, 409
get-next-event function 394
get-next-queued-form function 656
get-picture generic function 752
get-pixel generic function 758
get-polygon generic function 754
get-record-field function 592
get-scrap function 406, 409
get-string-from-user function 252
global point
  from local point 757
  to local point 757
global-to-local generic function 757
global variables 19
gorilla 676
grafport-font-codes function 79
GrafPort origin 145
GrafPorts 707
grafport-write-string macro 72
graphics 707

H

handle
  dialog item 192
  locked, checking 594
handle-locked-p function 594
handlep function 556
handler-case 189
have (obsolete ObjectLISP function), equivalent 664
Help
  Fred 42, 338
  Fred commands 19
  Listener commands 19
help features xxiii
Help keystroke 369
help-spec generic function 101
hfs device 326
hfs-volume-p function 326
hget-signed-long function 547
hierarchy 621
highlight-table-cell generic function 227
Horn, Berthold C. P. 765
href macro 584, 587
HyperCard, communicating with 423

I

ignore (Common Lisp declaration) 643
ignore-if-unused declaration 643
ignore-errors function 189
immediate objects 621
implementation 605–645
Include Close Box attribute 290
incremental compilation 7
incremental search 57
  commands 60–61
index-to-cell generic function 245
initargs 663
initforms 663
initialize-instance generic function
  button-dialog-item 206
  check-box-dialog-item 217
  default-button-dialog-item 208
  dialog-item 191
  editable-text-dialog-item 211
  fred-dialog-item 475
  fred-window 472
  input-stream 427
  menu 94
  menu-item 106
  output-stream 427
  pop-up-menu 235
  radio-button-dialog-item 219
  scroll-bar-dialog-item 236
  sequence-dialog-item 243
  simple-view 127

initialize-instance generic function *(continued)*
static-text-dialog-item 210
table-dialog-item 224
view 128
windoid 184
window 153
window-menu-item 117
initial value forms (initforms) 663
initialization arguments (initargs) 663
initialize-instance generic function
vs. make-instance 672
in-package statement 35
input-stream class 426
insertion font 33
inset-rect function 722
inset-region function 744
Inside Macintosh 766
Inspect (command on Tools menu) 369
inspect function 347, 371
inspect generic function 19
inspecting current s-expression 339
Inspect menu item 19
Inspector 369-372
features variable 370
package variable (Common Lisp) 370
readtable variable (common Lisp) 370
disassembly 371
front-window function 370
Help 370
list all Lisp objects that have been inspected 370
list-all-packages function (common Lisp) 370
list devices 370
logical directory names 370
logical hosts 370
MCL functions related to the Inspector 371–372
options 370
Record Types option 370
windows function 370
install-appleevent-handler function 417
installation xxxii
install-queued-reply-handler function 418
install-view-in-window generic function 132, 189, 192, 202, 203, 204, 382
instance allocation 666
instances
copy instance 617
creating 664
updating 663, 666
instantiation
button dialog items 206
checkbox dialog items 217
default button dialog items 208
dialog items 191
editable-text dialog items 211
floating windows 184
Fred dialog items 475
Fred windows 472
menu items 106
menus 94
pop-up menus 235
radio-button dialog items 219
scroll-bar dialog items 236
sequence dialog items 243
simple views 127
static-text dialog items 210
table dialog items 224
views 128
window menu items 117
windows 153
Interface Toolkit 277–293
add dialog items 290
edit dialog items 291
loading 279
internalize-scrap generic function 409, 410
intersect-rect function 722
intersect-region function 745
invalidate-corners generic function 138
invalidate-region generic function 139
invalidate-view generic function 139
invert-arc generic function 739
invert-oval generic function 732
invert-polygon generic function 756
invert-rect generic function 728
invert-region generic function 749
invert-round-rect generic function 735
i-search prompt 58
i-search reverse prompt 58
item-named. *See* view-named generic function

K

Keene, Sonya E. 765
keyboard equivalents
Command-Meta-click 43
Command-Option-click 42
Control-A 44
Control-B 44
Control-E 44
Control-equal sign 42, 339
Control-F 44
Control-G 60
Control-K 52
Control–Left Arrow 44
Control-M 54
Control-Meta-A 44
Control-Meta-B 44
Control-Meta-Delete 52
Control-Meta-E 44
Control-Meta-F 44
Control-Meta-H 47
Control-Meta-K 52
Control-Meta-L 56
Control-Meta-N 45
Control-Meta-number 57
Control-Meta-O 49
Control-Meta-P 45
Control-Meta-Q 49
Control-Meta-semicolon 53, 55
Control-Meta-Shift–Down Arrow 48
Control-Meta-Shift-M 54
Control-Meta-Shift-N 48
Control-Meta-Shift-P 48
Control-Meta-T 51
Control-Meta-underscore 56
Control-Meta–close parenthesis 45
Control-Meta–open parenthesis 45
Control-Meta–Shift–Up Arrow 48
Control-Meta–Space bar 47
Control-N 44
Control-number 57
Control-O 49
Control-P 44
Control-Q 50, 60
Control–question mark 42, 338
Control-R 60
Control-Return 49
Control-Right Arrow 44
Control-S 60
Control-S Control-W 60
keyboard equivalents *(continued)*
Control-S Control-Y 60
Control-Shift-A 47
Control-Shift-E 47
Control-Shift–Left Arrow 46
Control-Shift-N 47
Control-Shift-P 47
Control-Shift–Right Arrow 47
Control-Shift-V 48
Control-S Meta-W 60
Control–Space bar 51
Control-T 51
Control-Tab 45
Control-U 57
Control-underscore 56
Control-V 45
Control-W 49, 53, 60
Control-X Control-A 43, 338
Control-X Control-C 54
Control-X Control-D 43, 339
Control-X Control-E 54
Control-X Control-F. *See* Control-X Control-V keystroke
Control-X Control-I 369
Control-X Control-M 54
Control-X Control-R 54
Control-X Control-S 56
Control-X Control-V 56
Control-X Control-W 56
Control-X Control-X 48, 51
Control-X H 47
Control-X semicolon 55
Control-X U 56
Control-Y 49, 60
defining 504
Delete 52, 60
Enter 54
Help 369
Left Arrow 44
Meta-B 44
Meta-backslash 53
Meta-C 51
Meta–close parenthesis 45
Meta-D 52
Meta-Delete 52
Meta–double quotation mark 50
Meta-F 44
Meta–greater than 45
Meta-L 50
Meta–left angle bracket 45

keyboard equivalents *(continued)*
Meta-Left Arrow 44
Meta–left parenthesis 50
Meta–less than 45
Meta-M 45
Meta-number 50, 57
Meta–open parenthesis 50
Meta-period 42, 338
Meta-point 42, 338
Meta–right angle bracket 45
Meta–Right Arrow 44
Meta-semicolon 55
Meta–sharp sign 50
Meta-Shift–Left Arrow 46
Meta-Shift–Right Arrow 46
Meta-Shift-V 48
Meta–Space bar 53
Meta-T 51
Meta-U 50
Meta-V 44
Meta-W 49, 53
Meta-Y 49
Return, vs. Control-Return 4
Right Arrow 44
Shift–Down Arrow 47
Shift–Left Arrow 46
Shift–Page Down 48
Shift–Page Up 48
Shift–Right Arrow 46
Shift–Up Arrow 47
Tab 49
`key-handler-idle` generic function 383
`key-handler-list` generic function 381, 383
`key-handler-mixin` class 212, 381
`key-handler-p` generic function 212, 383
keys, extended Apple Keyboard, in Fred 41
`keystroke-code` function 508
`keystroke-function` generic function 509
`keystroke-name` function 508
Kiczales, Gregor xxii, 765
`kill-picture` function 752, 753
`kill-polygon` function 754
kill ring 16, 446, 500
data structure 33
definition 33
Koschmann, Timothy 764

L

lambda lists, congruent 669
Lawless, Jo A. 765
leading 73, 76
Left Arrow keystroke 44
`line` generic function 720
`lines-in-buffer` function 455
`line-to` generic function 719
Lisp operations 54
`list-all-packages` function (Common Lisp) 370
List Definitions menu item 19
Listener 3–6
defined 3
editing features 4
evaluating complete expression by pressing Return 6
Listener Commands menu item 19
variables associated with 5–6
Listener commands, Help on 19
Listener Commands menu item 19
literal characters 50
loading files 11
Load menu item 17
load-on-call memory management 630
`local` macro 357
local point
from global point 757
to global point 757
`local-to-global` generic function 757
`lock-file` function 323
logical directory names 330–331
logical hosts 298, 301–302
`long-method-combination` class 606
`lsh` function 625

M

`mac-default-directory` function 311
`mac-directory-namestring` function 313
`mac-file-creator` function 323
`mac-file-namestring` function 313
`mac-file-type` function 323
Macintosh Common Lisp
compatibility between version 2 and earlier versions 659
customizing for applications 648–656
customizing MCL behavior 20, 25
hardware and software requirements xxxi
implementation notes 605–645

Macintosh Common Lisp *(continued)*
information about 763–769
install xxxii
menu bar commands 13–20
programming procedure 9–11
quit xxxii
start xxxii
syntax xxix–xxxi
Macintosh data structures 537
Macintosh editing features 38
Macintosh Operating System 122, 537, 539
MCL functions for strings, pointers, and handles 554–558
Macintosh Programmer's Workshop interfaces 537
Macintosh Programmer's Workshop, communicating with 423
Macintosh Toolbox 537
calling Macintosh Common Lisp 532
`mac-namestring` function 312
macptrs 538, 539
`macptrp` function 555
macro dispatch characters, inspect 370
`macroexpand` function 339
macroexpanding current s-expression 339
macroexpanding current s-expressions repeatedly 339
`macroexpand-1` function 339
mactype, definition (a record field type) 566
`make-bitmap` function 749
`make-buffer` function 449
`make-color` function 263
`make-comtab` function 512
`make-dialog-item` function 193
`make-instance` generic function 664
`menu` 94
`menu-item` 106
`simple-view` 127
`view` 128
vs. `initialize-instance` 672
`window` 153
`window-menu-item` 117
`make-mark` function 447
`make-pathname` function 303
`make-point` function 67
`make-record` function 542
`make-record` macro 583
map
point 759
polygon 761
rectangle 760
region 761
`map-point` function 759
`map-polygon` function 761
`map-rect` function 760
`map-region` function 761
`map-subviews` generic function 134
`map-windows` function 159
`mark-backward-p` function 448
`markp`. *See* `buffer-mark-p` function
`maybe-default-stream-reader` macro 433
`maybe-default-stream-writer` macro 432
MCL menu items
apropos 18
cancel current operation 17
close current window 15
compile file 17
continue current operation 18
copy text 16
cut text 16
delete text 16
documentation 18
edit definition 18
environment options 19
evaluate selection 17
evaluate window contents 17
exit 15
find source code 18
help on Fred 19
help on Listener 19
list definitions 19
load file 17
open file 15
open selected file 15
page setup 15
paste text 16
print 15
print options 15, 19
quit 15
restart current operation 18
revert window contents to last saved version of file 15
save copy of Listener contents as named file 15
save window contents to file 15

MCL menu items *(continued)*
save window contents under another name 15
search files for given string 19
search for definitions 18
search for text 16
select window 20
select window contents 16
stack backtrace 19
suspend current operation 18
undo earlier edits 16
undo last edit 16
memory management 541–561, 630–637
allocation 541
automatic (“garbage collection”) 12–13, 631637
load-on-call 630
Macintosh Memory Manager, bypassing 542
MCL functions for accessing memory 543–553
menu
Apple 14, 119
Apple, can never be removed 89
colors 274
Edit 16
Eval 17
Tools 18
menu bar 88–93
class 88
colors 91, 92, 93, 274
definition 88
finding menu title 91
flickering, prevention of 104
freezing 104
installing new set of menus 89
listing currently installed menus 89
redrawing 105
startup list 90
updating 119
`menubar` class 88
menu bar commands 13–20
`menubar` function 89
`menu` class 93
`menu-deinstall` generic function 96
`menu-disable` generic function 97
`menu-element` class 88
`menu-enabled-p` generic function 97
`menu-enable` generic function 97
menu handle 103
`menu-handle` generic function 103
`menu-id` generic function 104
`menu-installed-p` generic function 96
`menu-install` generic function 88, 96
`menu-item-action-function` generic function 105
`menu-item-action-function` generic function 105, 108
`menu-item-action` generic function 108
`menu-item-check-mark` generic function 111
`menu-item` class 106
`menu-item-disable` generic function 109
`menu-item-enabled-p` generic function 110
`menu-item-enable` generic function 110
menu items 105–115
action 108
actions 105, 108
active window menu items 115
adding to menu 98
Balloon Help 101
check-mark character 111
check-mark character, setting 111
colors 114, 115, 275
command-key equivalent 110
command-key equivalent, setting 110
creating instances 106
disabling 109
enabled, checking if 110
enabling 110
finding in menu 100
font style 112
font style, setting 112
in active window 117
listing 100
Macintosh Common Lisp menu items
About Macintosh Common Lisp 14
performing action 108
removing from menu 99
retitling 109
setting action 108
setting check-mark character 111
setting command-key equivalent 110
title 109
updating 113, 119
window menu items 115
`menu-items` generic function 100
`menu-item-style` generic function 112
`menu-item-title` generic function 109

menu-item-update-function generic function 113
menu-item-update generic function 113, 119
menus 85, 120. *See also* menu items, window menu items
  advanced menu features 103–105
  Balloon Help 101
  colors 88, 101, 102
  creating instances 94
  definition 93
  deinstalling 96
  dimming 97
  disabling 97
  enabled, checking if 97
  enabling 97
  File 15
  font style 98
  freezing menu bar 104
  handle to menu record 103
  hierarchical 87
  identification number association list 104
  installation, checking for 96
  installing 96
  MCL forms relating to menu elements 98–102
  MCL forms relating to menus 93
  removing from menu bar 96
  restoring 97
  retitling 95
  title 95, 437
  turning off 97
  turning on 97
  unique numeric identification 104
  update function 98
  updating 103, 119
  Windows 20
menu-style generic function 98
menu-title generic function 95, 437
menu-update generic function 103, 119
menu-update-function generic function 98
merge-font-codes function 80
message-dialog function 249
Meta-B keystroke 44
Meta-backslash keystroke 53
Meta-C keystroke 51
Meta–close parenthesis keystroke 45
Meta-Delete keystroke 52
Meta-D keystroke 52
Meta–double quotation mark keystroke 50
Meta-G keystroke 4
Meta–greater than keystroke 45
Meta key 39
Meta–left angle bracket keystroke 45
Meta-Left Arrow keystroke 44
Meta–left parenthesis keystroke 50
Meta–less than keystroke 45
Meta-L keystroke 50
Meta-M keystroke 45
Meta modifier 39
Meta-number keystroke 57
Meta–number sign keystroke 50
metaobject class 606
Metaobject Protocol xxii, 606–621
  introspective MOP functions 608
  MCL functions related to 608–621
  metaobject classes 606–608
Meta–open parenthesis keystroke 50
Meta-period keystroke 42, 338
Meta-quotation mark keystroke 50
Meta–right angle bracket keystroke 45
Meta–Right Arrow keystroke 44
Meta-semicolon keystroke 55
Meta–sharp sign keystroke 50
Meta-Shift–Left Arrow keystroke 46
Meta-Shift–Right Arrow keystroke 46
Meta-Shift-V keystroke 48
Meta–Space bar keystroke 53
Meta-T keystroke 51
Meta-U keystroke 50
Meta-V keystroke 44
Meta-W keystroke 49, 53
Meta-Y keystroke 49
method class 606
method-combination class 606
method-exists-p function 619
method-function generic function 615
method-generic-function generic function 615
method-name generic function 616
method qualifiers 678
method-qualifiers generic function 616
methods 661–662
  accessor 671
  auxiliary 678
  congruent lambda lists 669
  defining new methods 668
  definition 662
  for a generic function 615

methods *(continued)*
generic function associated with method 615
generic functions run by method 615
name 616
on given specializer 613
qualifiers 616
return method for generic function 619
specializers 617
`method-specializers` generic function 617
Miller, Molly M. 764, 765
`mini-buffer` class 502
`mini-buffer-string` generic function 504
`mini-buffer-update` generic function 503
minibuffers
clear 36
default font 36
in Fred 502
mixins 680
MMU 633
modal dialog 246
`modal-dialog` generic function 257
modeless dialog 246
MOP. *See* Metaobject Protocol
`most-positive-fixnum` constant 64
mouse
button pressed, checking 391
create mouse-sensitive items 403
double clicks 392
double clicks, checking 392
position 391
`mouse-down-p` function 391
mouse-sensitive items, creating 403
`move` generic function 719
`move-mark` function 448
`move-to` generic function 719
multitasking 11

N

namestrings
definition 297
parsing 305
`new-compiler-policy` function 639
New Dialog menu item 288
New menu item 15
`new-region` function 741
`next-screen-context-lines` function 38
nickname 146, 191
`no-queued-reply-handler` generic function 419
Norvig, Peter 764
Note icon 250
numbers 624–625
number-sign dollar-sign reader macro 568
number-sign underbar reader macro 567
numeric arguments 624

O

`offset-polygon` function 755
`offset-rect` function 721
`offset-region` function 744
online documentation xxiii
OPEN (Common Lisp function) 319
Open menu item 15
`open-region` generic function 743
`open-application-handler` generic function 420
`open-documents-handler` generic function 422
Open Selected File menu item 15
`optimize` declarations 640
option character set 50
Option character set, in Fred 40
dead keys 40
Option-D 302
`option-key-p` function 393
`origin` generic function 710
ostype 603
`output-stream` class 427
oval
erase 731
fill 732
frame 730
invert pixels 732
paint 731

P

packages 34–35, 628–629
inspect current 370
set 34, 35
page setup 15
`paint-arc` generic function 738
`paint-oval` generic function 731
`paint-polygon` generic function 756
`paint-rect` generic function 727
`paint-region` generic function 748
`paint-round-rect` generic function 734

parentheses, inserting 50
part-color generic function 272
  dialog-item 199
  menu 101
  menubar 91
  menu-item 114
  table-dialog-item 230
  window 170
part-color-list generic function 273
  dialog-item 200
  menu 102
  menubar 93
  menu-item 115
  window 171
Pascal language
  calling Macintosh Common Lisp 532
  calling sequence 560
  null pointer 538, 559
  Pascal types and MCL equivalents 572
  true and false 604
  VAR arguments 559, 594
Paste command 33
paste generic function 16, 116, 181, 216, 494
Paste menu item 16
pathnames 299–329
  creating 302
  definition 297
  escape character 307
  MCL functions related to 303–305
  namestring conversion 299
  numeric arguments 298
  parsing 305–306
  printing 298
  quoting special character 307
  reading 298
  set pathname 327–329
  specification 302
pen-hide generic function 713
pen-mode generic function 716
pen-mode keywords 82
pen-normal generic function 718
pen-pattern generic function 715
pen-position generic function 713
pen-show generic function 713
pen-shown-p generic function 713
pen-size generic function 714
pen-state generic function 718
%copy-float function 531
%gen-trap function 601
%get-byte function 544
%get-cstring function 549
%get-long function 547
%get-ostype function 549
%get-ptr function 548
%get-signed-byte function 544
%get-signed-long function 547
%get-signed-word function 545
%get-string function 549
%get-unsigned-byte function 544
%get-unsigned-long function 547
%get-unsigned-word function 546
%get-word function 546
%hget-byte function 545
%hget-long function 547
%hget-ptr function 548
%hget-signed-byte function 545
%hget-signed-word function 546
%hget-unsigned-long function 548
%hget-word function 546
%hput-byte function 550
%hput-long function 551
%hput-ptr function 552
%hput-word function 551
%incf-ptr macro 558
%inc-ptr function 558
%int-to-ptr function 558
%null-ptr macro 559
%null-ptr-p function 559
%path-from-fsspec function 417
%ptr-to-int function 558
%put-byte function 550
%put-cstring function 553
%put-full-long function 551
%put-long function 551
%put-ostype function 553
%put-ptr function 552
%put-string function 552
%put-word function 550
%setf-macptr function 541
%stack-block macro 542
%set-toplevel function 654
%stack-block vs. rlet 542
%vstack-block macro 542
%word-to-int function 531
pictures
  draw 753
  get 752
  kill 753
  start 752
  stop and return 752

pixel, getting value 758
pointerp function 555
pointers 141
  into a buffer 446
point-h function 66
point-in-click-region-p generic function 148
point-in-rect-p function 724
point-in-region-p function 746
points 708
  global point to local point 757
  local point to global point 757
  map 759
  scale point 758
points-to-rect function 724
points 64–68
  adding 68
  compare with EQL 65
  constructing from coordinates 67
  definition 64
  encoded form 64
  find view containing 146
  global point to local point 757
  horizontal coordinate of 66
  integer, not fixnum 64
  local point to global point 757
  map 758
  relative placement 66
  scale point 758
  string representation of 65
  subtracting points 68
  vertical coordinate of 66
point<= function 66
point-string function 65
point-to-angle function 725
point-to-cell generic function 233
point-v function 66
polygons
  erase 756
  fill 756
  frame 755
  get 754
  invert 756
  kill 754
  map 761
  move 755
  paint 756
  start 754
  stop and return 754
pop-up-menu class 234
pop-up menus 234
PortRect 708
pref macro 586
preload-all-functions function 631
press-button generic function 207
primary method 674
print 15
print-call-history function 346
print-db macro 373
Print Dialog Source menu item 288
printing 493
  print options 19
printing variables 629
print options 15, 19, 21
Print Options menu item 19
print-documents-handler generic function 422
provide function 310
purge-functions function 630
pushed button 220
pushed-radio-button generic function 220
put-scrap function 407

Q

queued-reply-handler generic function 419
QuickDraw 130, 131, 180, 205, 707
  Color QuickDraw command set 263
QuickDraw graphics 705–761
quit xxxii, 15
quit-application-handler generic function 421
quit function xxxii
quoted characters 50

R

radio-button-cluster generic function 219
radio-button-dialog-item class 190, 218
radio-button dialog items 218–221
  creating instances 219
radio-button-pushed-p generic function 221
radio-button-push generic function 220
radio-button-unpush generic function 220
raref macro 590
rarset macro 591
reader macros 623, 626

reader methods 671
reading current s-expression 339
readtable, inspect current 370
`real-color-equal` function 262, 265
`real-font` function 71
record field
  definition 575
  structure 575
record field types (mactypes) 566, 571–574
  length 578, 579
`record-fields` function 573
`record-info` function 574
`record-length` macro 573
records 574–594
  changes from previous version of Macintosh Common Lisp 565
  copying 592
  create efficient record 580
  create record with indefinite extent 583
  create temporary record 580
  define record type 576
  definition 565, 574
  dispose of 584
  examples 570
  find fields 573
  find info about all fields 574
  find info about one field 574
  find length 573
  find record field type 573
  find record type 572
  listing record field types 593
  listing record types 593
  menu record handle 103
  printing values of fields 594
  record field, definition 575
  return contents of fields 584, 586, 587
  returning value of field 592
  setting value of field 593
  variant fields 579
record types 571–574
  define 576
  find 572
  inspect 370
  listing record field types 593
  listing record types 593
rectangle record, storage 709
rectangles
  angle number 725
  empty, checking 726
  equality between rectangles 725
  erase 728
  fill with pattern 729
  frame 726
  intersection 722
  invert pixels 728
  map 760
  move 721
  paint 727
  point in rectangle, checking 724
  points defining rectangle 724
  shrink or expand 722
  union 723
`rect-in-region-p` function 747
`redraw-cell` generic function 226
regions
  allocate new region 741
  close region record 743
  copy or create 742
  difference 745
  dispose 742
  empty 742
  empty, checking 747
  equality between regions, checking 747
  erase 748
  exclusive-or of two regions 746
  fill 749
  frame 748
  intersection 745
  invert pixels 749
  map 761
  move 744
  open region record 743
  paint 748
  point in region, checking 746
  rectangle, set to 742
  rectangle in region, checking 747
  shrink or expand 744
  union 745
register trap call 597
`register-trap` macro 600
`reindex-interfaces` function 569
`remove-dialog-items`. *See* `remove-subviews` generic function
`remove-key-handler` generic function 382
`remove-menu-items` generic function 99

remove-self-from-dialog (obsolete function) 202. *See also* remove-view-from-window generic function
remove-subviews generic function 137, 189
remove-view-from-window generic function 132, 133, 189, 202, 203, 382
rename-file function 320
replace 16
require function 309
require-trap-constant macro 568
require-trap macro 568
require-type function 644
restart 18
restart-case 189
Restarts menu item 18
Return, in Listener 6
return-from-modal-dialog macro 258
Revert menu item 15
RGB records
  binding to variable 267
  returning 266
  returning color from 267
rgb-to-color function 267
Right Arrow keystroke 44
rlet macro 580
  examples 570
room function 346
rotate-killed-strings function 501
rotating yank 49
rounded rectangles
  erase 735
  fill 736
  frame 733
  invert pixels 735
  paint 734
rref macro
  plus setf is rset 588
rset macro 588, 589
  rref plus setf 589

S

same-buffer-p function 449
save-application function 618, 648
Save As menu item 15
save-copy-as generic function 116, 181
Save Copy As menu item 15
Save menu item 15
scale-point function 758
scrap-handler class 409
screen drawing 707
  first QuickDraw point 84
  pen-mode keywords 82
  pixels per inch 84
  style encoding 83
  width and height 83
scroll-bar-changed generic function 242
scroll-bar-dialog-item class 236
scroll-bar dialog items
  creating instances 236
  initial setting, set 241
  length 238
  length, set 238
  maximum setting 238
  maximum setting, set 239
  minimum setting 239
  minimum setting, set 239
  page size 239
  scroll box, existence, checking 241
  scrollee 240
  scroll size 240
  setting 240
  track thumb, existence, set 241
  width 242
  width, set 242
scroll-bar-length generic function 238
scroll-bar-max generic function 238
scroll-bar-min generic function 239
scroll-bar-page-size generic function 239
scroll-bar-scrollee generic function 240
scroll-bar-scroll-size generic function 240
scroll-bar-setting generic function 240
scroll-bar-track-thumb-p generic function 241
scroll-bar-width generic function 242
scrolling 37
scroll-position generic function 232
scroll-rect generic function 750
scroll-to-cell generic function 232
search 16
search files 19
Search Files menu item 19
Search menu item 16
select-all generic function 16, 47, 116, 181, 216, 494
Select All menu item 16
select-backtrace function 346
selected-cells generic function 232

select entire buffer 47
selection-range generic function 486
selection, types allowed in tables 254
select-item-from-list function 254
sequence-dialog-item class 190, 243
sequence dialog items, creating instances 243
set-allow-returns generic function 215, 476
set-allow-tabs generic function 215, 476
set-cell-font generic function 230
set-cell-size generic function 229
set-choose-file-default-directory function 329
set-clip-region generic function 711
set-command-key generic function 110
set-current-compiler-policy function 643
set-current-file-compiler-policy function 643
set-current-key-handler generic function 211, 381
set-cursor function 404
set-dead-keys (obsolete mcl function) 41
set-default-button generic function 209
set-dialog-item-action-function generic function 195
set-dialog-item-font. *See* set-view-font generic function
set-dialog-item-handle generic function 203
set-dialog-item-size. *See* set-view-size generic function
set-dialog-item-text generic function 197, 211
set-empty-region function 742
set-event-ticks function 397
set-file-create-date function 321
set-file-write-date function 321
set-fore-color generic function 270
set-fred-display-start-mark generic function 479
set-fred-hscroll generic function 486
set-fred-last-command generic function 511
set-fred-margin generic function 491
set-fred-package generic function 490
set-gc-event-check-enabled-p function 637
set-grafport-font-codes function 79
set-internal-scrap generic function 407, 409
set-local macro 357
set-mac-default-directory function 311
set-mac-file-creator function 323
set-mac-file-type function 323
set-mark function 448
set-menubar function 88, 89
set-menu-item-action-function generic function 105, 108
set-menu-item-check-mark generic function 111, 118
set-menu-item-style generic function 112, 118
set-menu-item-title generic function 109
set-menu-item-update-function generic function 113
set-menu-title generic function 95
set-mini-buffer generic function 503
set-origin generic function 708, 710
set-part-color generic function 273
  dialog-item 200
  menu 102
  menubar 92
  menu-item 115
  table-dialog-item 230
  window 171
set-pen-mode generic function 716
set-pen-pattern generic function 715
set-pen-size generic function 714
set-pen-state generic function 718
set-record-field function 593
set-rect-region function 742
set-scroll-bar-length generic function 238
set-scroll-bar-max generic function 239
set-scroll-bar-min generic function 239
set-scroll-bar-scrollee generic function 240
set-scroll-bar-setting generic function 241
set-scroll-bar-track-thumb-p generic function 241
set-scroll-bar-width generic function 242
set-selection-range generic function 487
set-table-dimensions generic function 228
set-table-sequence generic function 244
setting a font 33

setup-undo generic function 498, 499
setup-undo-with-args generic function 500
set-view-container generic function 132
set-view-font-codes generic function 78, 169, 199
set-view-font generic function 74, 167, 198, 488
set-view-nick-name generic function 146
set-view-position generic function xxviii, 143, 161, 189
set-view-scroll-position generic function 145
set-view-size generic function 144, 163, 189, 196, 385
set-visible-dimensions generic function 228
set-window-filename generic function 489
set-window-layer generic function 155, 174
set-window-package generic function 35
set-window-position. *See* set-view-position generic function
set-window-size. *See* set-view-size generic function
set-window-title generic function 166
set-window-zoom-position generic function 175, 176
set-window-zoom-size generic function 177, 178
set-wptr-font-codes function 80
Shadow-Edge-Box dialog box 289
sharp-sign comments 50
sharp-sign dollar-sign reader macro 568
sharp-sign underbar reader macro 567
Shift–Down Arrow keystroke 47
shift-key-p function 393
Shift–Left Arrow keystroke 46
Shift–Page Up keystroke 48
Shift–Right Arrow keystroke 46
Shift–Up Arrow keystroke 47
short-method-combination class 606
simple bit vectors, print readably 629
simple vectors, print readably 629
simple-view class 126
simple views
  creating instances 127
  definition 122
Single-Edge-Box dialog box 289
slot 663
slot-definition-name generic function 617
slot definition objects 617
slots
  adding to class 663
  allocating 666
  associating value with, using setf 664
  finding value of 664
source code, editing definition 18
special characters, names 626
specializer
  generic functions with methods on specializer 614
  methods on specializer 613
specializer class 607
specializer-direct-generic-functions generic function 614
specializer-direct-methods generic function 613
stack allocation 542
Stack Backtrace 355
stack trap call 595, 597
stack-trap macro 598
standard-accessor-method class 606
standard-class class 607
standard-generic-function class 607
standard-method class 606
standard-method-combination class 606
standard-object class 606
standard-reader-method class 606
standard-writer-method class 606
start-picture generic function 752
start-polygon generic function 754
static-text-dialog-item class 190, 209
static-text dialog items
  creating instances 210
Steele, Guy L. xxii, 659, 764
step macro 359
stepper 358–359
Stop icon 250
stream-abort generic function 439
stream class 426
stream-clear-input generic function 436
stream-close generic function 438
stream-column generic function 434, 504
stream-direction generic function 428
stream-eofp generic function 436
stream-force-output generic function 428
stream-fresh-line generic function 435
stream-line-length generic function 434

`stream-listen` generic function 436
`stream-peek` generic function 434
`stream-reader` generic function 431
streams 426–439
  clear pending input from 436
  close 438
  close abnormally 439
  current column 434
  definition of class 426
  end, check for 436
  functionality 426
  increase speed by buffering 428
  line length 434
  outputting character 428
  print newline in 435
  read further characters from 436
  read next character 429
  reopen after close 438
  unread character from 430
  write all pending output 428
`stream-tyi` generic function 429
`stream-tyo` generic function 428, 720
`stream-untyi` generic function 430
`stream-writer` generic function 431
`stream-write-string` generic function 435
strings, print readably 629
`string-width` function 72
`structure-class` class 607
structured directories 313–316
`structure-object` class 606
`structurep` function 645
structures, print readably 629
`structure-typep` function 645
`subtract-points` function 68
`subviews` generic function 133
superclasses 621
suspend 18
Suspend command 11
Sussman, Gerald 765
syntax xxix–xxxi
system requirements xxxi

T

Tab keystroke 49
`table-dialog-item` class 190, 221, 223
table dialog items 221–234
  creating instances 224
  definition 221
`table-dimensions` generic function 227
`table-hscrollp` generic function 233
tables. *See* table dialog items
`table-sequence` generic function 244
`table-vscrollp` generic function 234
tagged pointer 621
tag values 621–623
tail recursion 638
`target` function 158
Tatar, Deborah 764
terminating a command 60
text, selecting 46
`throw-cancel` macro 248
`time` macro 372
Tool dialog box 289
Tools menu 18
`top-inspect-form` function 348, 371
`toplevel` function 653
`toplevel-loop` function 654, 655
Touretzky, David 765
`trace` macro 360
`trace-tab` function 364
tracing 359, 360–368
  MCL functions associated with tracing 363–365
transposition 51
trap call
  register 597
  stack 595, 597
traps 565–574, 594–604
  create named trap 569
  define behavior 569
  dispatch table 566
  examples 570, 577
  trap macro 566
  trap word 566
  update index files 567
  updating interface index files 569
turnkey dialogs 248
`tyi` (obsolete function) 439
`tyo` (obsolete function) 439

U

`unadvise` macro 368
`uncompile-function` function 356
`_waitnextevent` trap 655
undo 496, 497, 498
`undo` generic function 16, 116, 181, 216
Undo menu item 16
`undo more` generic function 16

undo-more generic function 116, 181, 216
Undo More menu item 16
union-rect function 723
union-region function 745
unlock-file function 323
untrace macro 363
update-cursor function 403
updating 766
usual- (obsolete Object Lisp function), equivalent 670
Use Dialogs menu item 287
user-pick-color function 268
uvectorp function 622, 623
uvectors 622, 623
  definition 621

V

validate-corners generic function 140
validate-region generic function 141
validate-view generic function 140
variant fields 579
versions of Macintosh Common Lisp. *See* compatibility
view, set position in container xxviii
view-activate-event-handler generic function 148, 189, 203, 380
view class 128
view-click-event-handler generic function 149, 189, 194, 195, 377, 387
view-container generic function 132, 189
view-contains-point-p generic function 147, 148
view-convert-coordinates-and-click generic function 150
view-corners generic function 138
view-cursor generic function 179, 401, 402
view-deactivate-event-handler generic function 148, 189, 204, 380
view-default-font generic function 167
view-default-position generic function 143, 165
view-default-size generic function 144, 165
  dialog-item 204
view-draw-contents generic function 149, 150, 200, 388, 389
view-focus-and-draw-contents generic function 151, 196
view-font-codes generic function 77, 168, 198
view-font generic function 74, 166, 189, 198, 488
view-key-event-handler generic function 211, 379
  editable-text-dialog-item 212
view-mini-buffer generic function 478, 503
view-mouse-enter-event-handler generic function 401, 403
view-mouse-leave-event-handler generic function 401, 403
view-mouse-position generic function 189, 391
view-named generic function 135, 189
view-nick-name generic function 146, 189
view-position generic function 142, 161, 189
views
  can take dialog items 188
  class hierarchy 124
  click region, checking 148
  container 132
  container set 136
  container, remove from 137
  contains point, checking 147
  creating instances 128
  deactivate 148
  default position 143
  default size 144
  definition 121, 122, 123
  focused 130, 131
  font codes 77
  font codes set 78
  implementation 122
  MCL expressions relating to views and simple views 126–152
  mouse clicks, checking 148
  nickname 146
  nickname set 146
  point in view 146
  point in view, checking 147
  returned clicked subview 137
  scroll position 145
  scroll position set 145
  set position in container 143
  size 144
  size set 144
  subviews 133

view-scroll-position generic function 145
view-size generic function 144, 163, 189
  dialog-item 196
view-subviews generic function 133
view-window function 142
visible-dimensions generic function 228
volume-number function 324
volumes, MCL operations related to 324–327

W

widmax 73, 76
wildcards 317
Wilensky, Robert 765
windoid class 183
windoids 124, 183
window-activate-event-handler. *See* view-activate-event-handler generic function
window-active-p generic function 173
window-buffer. *See* fred-buffer generic function
window-can-do-operation generic function 182, 495
window-can-undo-p (obsolete MCL function) 182, 495
window-can-undo-p generic function 216, 494
window class 153
window-click-event-handler. *See* view-click-event-handler generic function
window-close-event-handler generic function 387
window-close generic function 116, 160, 181
window-cursor generic function 180
window-deactivate-event-handler. *See* view-deactivate-event-handler generic function
window-default-zoom-position generic function 175, 176
window-default-zoom-size generic function 177
window-defs-dialog generic function 116, 181
window-do-first-click generic function 387
window-drag-event-handler generic function 386
window-drag-rect generic function 179
window-draw-grow-icon generic function 390
window-ensure-on-screen generic function 172
window-eval-buffer generic function 181
window-eval-selection generic function 116, 181
window-eval-whole-buffer generic function 116
window-event generic function 398
window-filename generic function 489
window-font. *See* view-font generic function
window-grow-event-handler generic function 385
window-grow-rect generic function 178
window-hardcopy generic function 116, 181, 493
window-hide generic function 172
window-hpos. *See* fred-hpos generic function
window-key-up-event-handler generic function 384
window-layer generic function 174
window-line-vpos. *See* fred-line-vpos generic function
window-menu-item class 117
window menu items 115–118
  creating instances 117
window-mouse-position. *See* view-mouse-position generic function
window-mouse-up-event-handler generic function 385
window-needs-saving-p generic function 116, 182
window-null-event-handler generic function 383, 384
window-object function 180
window-on-screen-p generic function 173
window-package generic function 35
window-position. *See* view-position generic function
window-revert generic function 116, 181, 492
windows 152
  activate 148, 175
  active, checking 173
  can take dialog items 188
  close 15, 160

windows *(continued)*
close event handler 387
color, find 170
color, return color list 171
color, set 171
colors 270, 271, 272, 275
create for selected file 15
creating instances 153
cursor 180
cursor shape 179
cursor show 480
deactivate 148
default zoom position 176
default zoom size 177, 178
definition 122, 124, 152
drag event handler 386
ensure on screen 172
floating 183–184
font 166
font codes 77, 168
font codes set 78, 169
font default 167
font spec set 167
font, use specific 131
functions, advanced 175
GrafPort, use specific 130, 180
grow event handler 385
hide 172
implementation 122
layer 174
layer set 174
layer set to front 175
manipulation 9
MCL functions related to windows 153
menu items 115
modified 489
pointer 141
position 161
position default 164, 165
position set 161
printing 493
resize subviews 164
resizing 179
save 182
saving state information 37
select 175
set package 34, 35
setting background color 270, 272
setting foreground color 270, 271
setting position of Listener 5
windows *(continued)*
setting size of Listener 6
show on screen 172
shown, checking 172
size 163, 178
size default 164, 165
size set 163
switch position of active and target 56
title 165
title set 166
unsaved 32
visible, checking 173
wptr pointing at window object 180
zoom position 175
zoom size 177
zoom size set 178
`window-save-as` generic function 56, 116, 181, 492
`window-save` generic function 56, 116, 181, 492
`window-select-event-handler` generic function 384
`window-select` generic function 175
`window-set-not-modified` generic function 489
`windows` function 156, 370
`window-show-cursor` generic function 480
`window-show` generic function 172
`window-shown-p` generic function 172
`window-size`. *See* `view-size` generic function
`window-size-parts` generic function 164
Windows menu 20
`window-start-mark`. *See* `fred-display-start-mark` generic function
`window-title` generic function 165
`window-update`. *See* `fred-update` generic function
`window-update-cursor` generic function 129, 402, 405
`window-update-event-handler` generic function 139, 388
`window-vpos`. *See* `fred-vpos` generic function
`window-zoom-event-handler` generic function 386
`window-zoom-position` generic function 175
`window-zoom-size` generic function 177
Winston, Patrick 765

`with-aedescs` macro 416
`with-back-color` macro 272
`with-cstrs` macro 554
`with-cursor` macro 402, 403, 404
`with-dereferenced-handles` macro 557
`with-focused-dialog-item` macro 205
`with-focused-view` macro 129, 130, 181, 708
`with-font-focused-view` macro 131
`with-fore-color` macro 271
`without-interrupts` special form 180, 358, 401, 405
`with-pointers` macro 557
`with-port` macro 180
`with-pstrs` macro 554
`with-returned-pstrs` macro 554
`with-rgb` macro 267
wptr
  font codes 80
  window object 180
`wptr-font-codes` function 80
`wptr` generic function 141
writer methods 671

X, Y, Z

x-coordinate of points 64
`xor-region` function 746
yank 49
  rotating 49
y-coordinate of points 64
`y-or-n-dialog` function 250
`zone-pointerp` function 556

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and Microsoft Word software. Proof and final pages were created on an Apple LaserWriter II NTX printer. Line art was created using Adobe Illustrator. PostScript®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica ®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writer: Sarah Smith

Illustrator: Sandee Karr

Production Supervisor: Janet M. Anders

Special thanks to Gary Byers, Alice Hartley, David Lamkins, Steven Mitchell, William St. Clair, and Andrew Shalit, and to our skilled and helpful group of alpha and beta testers