LMI

LAMBDA

[illegible]

[illegible]ase [illegible] Installation [illegible]

Editing the Lambda's Site Files

[illegible]ware and Interprocessor Communication

Common Lisp Notes

SYSTEM MAP for Release 2.0
** indicates location of tab divider in binder

These manuals are part of your Lambda documentation, but are not part of a binder.

Intro to Lambda
ZetaLISP-Plus Commands

Here are the binders and their contents:

BASICS:
- **LMI Lambda Technical Summary
- **LMI Lambda Field Service Manual
- **NuMachine Installation and User Manual

RELEASE NOTES:
- **Release 2.0 Overview & Notes
- **Release 2.0 Inst & Conversion
- **Editing Lambda Site Files
- **Tape Software & Streams
- **Common LISP Notes

LISP 1: The LISP Machine Manual, Part 1
- **Introduction
- Primitive Object Types
- Evaluation
- Flow of Control
- Manipulating List Structure
- **Symbols
- Numbers
- Arrays
- Strings
- **Functions
- Closures
- Stack Groups
- Locatives
- Subprimitives
- Areas
- **The Compiler
- Macros
- The LOOP Iteration Macro
- **Defstruct

LISP 2: The LISP Machine Manual, Part 2
- **Objects, Message Passing, and Flavors
- **The I/O System
- Naming of Files
- The Chaosnet
- **Packages
- Maintaining Large Systems
- Processes
- Errors and Debugging
- **How to Read Assembly Language
- Querying the User
- Initializations
- Dates and Times
- Miscellaneous Useful Functions
- **Indices

LISP 3:
- **Introduction to the Window System
- **The Window System Manual
- **ZMAIL Overview
- **ZMAIL

EDITORS:
- **ZMACS Introductory Manual
- **ZMACS Reference Manual
- **Mince

UNIX 1:
- **NuMachine Release and Update Information
- **NuMachine Operating System
- **UNIX Programmer's Manual, V. 1: Section 1
- ** Sections 2-8

UNIX 2:UNIX Programmer's Manual, Vol. 2
- **The UNIX Time-sharing System
- UNIX for Beginners - Second Edition
- A Tutorial Introduction to the UNIX Text Editor
- Advanced Editing On Unix
- An Introduction to the UNIX Shell
- Typing Documents on the UNIX System
- A Guide to Preparing Documents with -ms
- Tbl-A Program to Format Tables
- NROFF/TROFF User's Manual
- A TROFF Tutorial
- **The C Programming Language Reference Manual
- Recent Changes to C
- Lint, A C Program Checker
- Make-A Program for Maintaining Computer Programs
- **UNIX Programming-Second Edition
- A Tutorial Introduction to ADB
- Yacc: Yet Another Compiler-Compiler
- Lex-A Lexical Analyzer Generator
- **A Portable Fortran 77 Compiler
- RATFOR-A Preprocessor for a Rational Fortran
- The M4 Macro Processor
- SED-A Non-Interactive Text Editor
- Awk-A Pattern Scanning and Processing Language (2d. ed.)
- DC-An Interactive Desk Calculator
- BC-An Arbitrary Precision Desk-Calculator Language
- An Introduction to Display Editing with Vi
- **The UNIX I/O System
- On the Security of UNIX
- Password Security: A Case History

HARDWARE 1:
- **NuMachine Technical Summary
- **SDU Monitor User's Manual
- SDU General Description
- **Mouse Manual
- **LMI Printer Software Manual
- **VR-Series Monitor
- Z29 Monitor

HARDWARE 2:
- **Tape Drive
- **Disk Drive
- **Kermit

OPTIONS:
- **(varies according to options purchased)
- Prolog
- Interlisp
- Fortran Installation Memo
- Scribble
- Ethernet Multibus
- Medium Res Color System
- MTI System

LMI Release 2.0 Package

May 1, 1985

24-0100340-0001

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

SYSTEM MAP for Release 2.0
** indicates location of tab divider in binder

These manuals are part of your Lambda documentation, but are not part of a binder.

Intro to Lambda
ZetaLISP-Plus Commands

Here are the binders and their contents:

BASICS:
**LMI Lambda Technical Summary
**LMI Lambda Field Service Manual
**NuMachine Installation and User Manual

RELEASE NOTES:
**Release 2.0 Overview & Notes
**Release 2.0 Inst & Conversion
**Editing Lambda Site Files
**Tape Software & Streams
**Common LISP Notes

LISP 1: The LISP Machine Manual, Part 1
**Introduction
Primitive Object Types
Evaluation
Flow of Control
Manipulating List Structure
**Symbols
Numbers
Arrays
Strings
**Functions
Closures
Stack Groups
Locatives
Subprimitives
Areas
**The Compiler
Macros
The LOOP Iteration Macro
**Defstruct

LISP 2: The LISP Machine Manual, Part 2
**Objects, Message Passing, and Flavors
**The I/O System
Naming of Files
The Chaosnet
**Packages
Maintaining Large Systems
Processes
Errors and Debugging
**How to Read Assembly Language
Querying the User
Initializations
Dates and Times
Miscellaneous Useful Functions
**Indices

LISP 3:
**Introduction to the Window System
**The Window System Manual
**ZMAIL Overview
**ZMAIL

EDITORS:
**ZMACS Introductory Manual
**ZMACS Reference Manual
**Mince

UNIX 1:
**NuMachine Release and Update Information
**NuMachine Operating System
**UNIX Programmer's Manual, V. 1: Section 1
** Sections 2-8

UNIX 2:UNIX Programmer's Manual, Vol. 2
**The UNIX Time-sharing System
UNIX for Beginners - Second Edition
A Tutorial Introduction to the UNIX Text Editor
Advanced Editing On Unix
An Introduction to the UNIX Shell
Typing Documents on the UNIX System
A Guide to Preparing Documents with -ms
Tbl-A Program to Format Tables
NROFF/TROFF User's Manual
A TROFF Tutorial
**The C Programming Language Reference Manual
Recent Changes to C
Lint, A C Program Checker
Make-A Program for Maintaining Computer Programs
**UNIX Programming-Second Edition
A Tutorial Introduction to ADB
Yacc: Yet Another Compiler-Compiler
Lex-A Lexical Analyzer Generator
**A Portable Fortran 77 Compiler
RATFOR-A Preprocessor for a Rational Fortran
The M4 Macro Processor
SED-A Non-Interactive Text Editor
Awk-A Pattern Scanning and Processing Language (2d. ed.)
DC-An Interactive Desk Calculator
BC-An Arbitrary Precision Desk-Calculator Language
An Introduction to Display Editing with Vi
**The UNIX I/O System
On the Security of UNIX
Password Security: A Case History

HARDWARE 1:
**NuMachine Technical Summary
**SDU Monitor User's Manual
SDU General Description
**Mouse Manual
**LMI Printer Software Manual
**VR-Series Monitor
Z29 Monitor

HARDWARE 2:
**Tape Drive
**Disk Drive
**Kermit

OPTIONS.
**(varies according to options purchased)
Prolog
Interlisp
Fortran Installation Memo
Scribble
Ethernet Multibus
Medium Res Color System
MTI Systeel

Introduction

This is your LMI Release 2.0 documentation. It is organized to reflect basic information about the software, general conditions of its use, and specific new features.

With it, we are upgrading some other features of your documentation to make it more convenient for you to use.

- Chief among these are some new introductory manuals to the Lambda system. *ZMail Overview* and *Introduction to the Window System* bridge the gap between previously available reference materials and your needs as a new user.
- *ZetaLISP-Plus Commands*, our pocket-sized reminder book, summarizes the most frequently used commands in ZMACS, the HELP facility, and the Debugger.
- With this release we are also distributing WindowMaker (tm), a new LMI facility allowing you to create windows graphically and edit automatically produced code. WindowMaker ends the tedium of writing multiple-window environments and allows you to customize your programming environment in minimal time.
- KERMIT, an industry-standard uploading and downloading facility, is now available on the Lambda. With this release you will receive LMI and full CUCCA documentation on KERMIT.
- In order to make your tape and disk drive documentation more readily available to you, we have included new, shorter documentation on these features.
- If you have chosen to receive the Microcompiler, our new facility for compiling directly LISP to microcode, your Microcompiler documentation will also be included in this package.

Your Release 2.0 information consists of the following:

- *Release 2.0 Overview*—**Read this first** to see the advantages and new capabilities of Release 2.0.
- *Release 2.0 Notes*—System release notes, covering all features of the system in detail.
- *Installation*—How to install your new system.
- *Release 2.0 Conversion*—Converting older ZetaLISP code to Release 2.0.
- *Editing the Lambda's Site Information*—Customizing your site for your software upgrade or new Lambda.
- *Lambda Tape Software*—Full information on tape functions.
- *Interprocessor Communication: The Extended STREAMS Interface*—How to communicate among processors in your Lambda.

Note for upgrade customers only (new customers have these changes incorporated in their manual sets):

In order to make these manuals more easily available to you, we request that you make the following changes to your document set.

1. In your RELEASE NOTES binder, discard the *System 94 Notes, System 98 Notes,* and *Release 1.2 Notes.* (Your set may not contain all of these notes.)

   The new RELEASE NOTES binder organization is as follows:

- New Release Notes table-of-contents sheet (striped cover stock sheet, beginning "Release 2.0 Overview and Notes")
- Red tab
- *Release 2.0 Overview*
- *Release 2.0 Notes*
- Yellow tab
- *Release 2.0 Installation*
- *Release 2.0 Conversion*
- Green tab
- *Editing the Lambda's Site Files*
- Blue tab
- *Lambda Tape Software*
- *Interprocessor Communication: The Extended Streams Interface*
- Purple tab
- *Common LISP Notes*

2. Please include the following in your LISP 3 binder:
   - New table-of-contents sheet, beginning "Introduction to the Window System", should be substituted for old.
   - *The WindowMaker*, documentation on LMI's new automatic window-making facility, should be inserted.
   - The *ZMail Overview* should be inserted.

   If you are a new or upgraded customer as of May 1, you will receive a copy of the revised *Introduction to the Window System* shortly.
3. Please discard all the contents of HARDWARE 2 and HARDWARE 3, except the "LMI Lambda" index tabs. In HARDWARE 2, insert the following:
   - New table-of-contents, beginning "Tape Drives"
   - Red tab
   - Tape Drives package
   - Yellow tab
   - Disk Drives package
   - Green tab
   - KERMIT package
   - Blue tab
   - Purple tab

   HARDWARE 3 will not currently be used.

Thank you for your help in making LMI documentation work better for you. If you have suggestions and comments on LMI documentation, please contact me directly, via your LMI electronic mail account (my username is **swrs**), by telephone via LMI Customer Service, or by US mail at LMI, 1000 Massachusetts Ave., Cambridge MA 02138 USA.

(Dr.) Sarah Smith

Director of Documentation, LMI

Release 2 Overview

Published by LMI 1000 Massachusetts Avenue, Cambridge MA 02138 USA

Introduction

Release 2 offers subtantial improvements over Release 1 in almost every aspect; it is faster, has more features, and is less susceptible to bugs. The most important advance of Release 2 on the Lambda is the adoption of 25-bit pointers; now the address space is twice as large as before, so that user programs now have up to 16000K words more space to use. Execution speed has been increased. Release 2 contains improvements in the user interface (notably in the rubout handler and Zmacs, the editor). In addition, Common LISP is supported in full.

Common LISP

The most noticeable software change for Release 2 is the addition of support for Common LISP. Although Common LISP and ZetaLISP-Plus are very close, there are differences; you have a choice of what incompatible Common LISP functions you would like to use. Here are the general differences:

- Constructs (functions and variables) tend to have more regular names. (The names of many ZetaLISP-Plus system variables are the same as those in Maclisp and Franz LISP.) For example, in Common LISP, **base** becomes ***print-base***.
- Some functions and situations are more well-defined.
- Some Common LISP features are oriented towards conventional architectures. (On the LISP Machine they are legal, but superfluous.)

Common LISP and ZetaLISP-Plus exist side by side in LMI software. For incompatible functions with the same name in both dialects, and when the input syntax is slightly different, the machine interprets the dialect that the form is in according to the time when the form is entered. (This is implemented with readtables.) When the machine is running code, instead of declaring a "mode" for Common LISP, one tells the editor or the LISP Listener what dialect is desired.

- Programmers can use new, upward-compatible Common LISP functions from ZetaLISP-Plus without special arrangement.
- Incompatible Common LISP functions can be used from ZetaLISP-Plus programs by explicitly referencing the Common LISP incompatible **(CLI)** package.
- Incompatible ZetaLISP-Plus functions can be used from Common LISP, if needed, by explicitly referencing the **GLOBAL** package.
- Both the old and the new, Common LISP names of variables can be used; for ZetaLISP-Plus, no "preferred" name is enforced.

This set of features make it possible to port an existing ZetaLISP-Plus program to Common LISP function by function, file by file.

What does Common LISP bring to ZetaLISP-Plus ?

Many aspects of Common LISP can be adopted by programmers without using incompatible functions. The following is a list of what is immediately available for programmers in ZetaLISP-Plus:

- Better names for certain functions and variables.
- Long awaited printer/reader features. A wider range of structures can be read and printed by the LISP function **(READ)**.
- Compatibility with other LISP implementations (the next NIL, Spice LISP, Standard LISP).
- Lexical scoping. Many locally special declarations are no longer needed. Local function and macro definitions are now possible.
- Upward compatible changes to many existing ZetaLISP-Plus functions. (One of the more important is the tighter definition of **SETF**.)
- A more comprehensive type system.
- Character objects, rational and complex numbers.
- Generic sequence functions that accept both vectors (one-dimensional arrays) and lists. Many of these functions express common programming idioms that are usually not defined as part of LISP.

Other Improvements

Besides the adoption of Common LISP, Release 2 features some new functions, many bug fixes, and improved implementations of important system facilities.

- **DEFSUBST** is fixed; certain bugs associated with the previous implementation have gone away.
- Translation for logical hosts can now be automatically updated.
- The rubout handler has been improved. For example, one can edit the end of one's input, and have it be re-input, if syntactically possible, with a single keystroke.
- The default font **CPTFONT** is more readable. The lower-case letters have been enlarged slightly, so LISP code in lower case should be more readable.
- More fonts have a wider range of characters. Earlier on, only a few fixed-width fonts actually had glyphs for every printing character in the LISP Machine character set. However, many of the text-oriented fonts (the **HL**/Helvetica series, the **TR**/Times Roman series) have had more characters added to them. For example, backquote (the character ` should now be available in all text fonts.
- Notifications are handled differently. Now, only the appropriate flavors of windows will print out notifications on ~~on~~ themselves. Most windows will handle notifications by telling you about them (via the documentation window at the bottom of the screen); there are commands to view notifications at your leisure with a ⟨TERMINAL⟩ command.

Editor and ZMail Improvements

- The most useful improvement to the editor (meaning Zmacs, which implements both **Zwei** and **ZMail**) is that "undoability" of editing is now allowed for all modifications. Before, only a few operations had methods for undo the changes; now, any change to text can be undone. The editing history is maintained on a per-section basis, so that you can now undo changes selectively in one unit of your buffer. Usually, that unit will be a LISP defining form (**defun**, **defmethod**, and so on) or a paragraph.

A common application for such generalized 'undoability' is program modification. You no longer need to save away changed sections of programs while modifying them, but can simply undo the changes that are local to a section.

- In **Zwei**, some commands have been renamed to more cleverly exploit command completion.
- **Zwei** also has commands for supporting Common LISP. (These commands the attribute ?verb? list of the buffer you are editing, to allow to use either dialect on a per-buffer basis.) **Zwei** understands the slight differences between Common LISP and ZetaLISP-Plus syntax.
- The 'modified' flag (*) in editor modelines has been moved to the left so that a modified buffer is more noticeable. Formerly, it was sometimes hard to discern a modified buffer if one was editing a file with a long name.
- **ZMail** performance has been improved; in addition, several bugs have been fixed. There are new profile variables for more customization, and a new Undigestify command (for reading mailing list digests) has been provided.

Using Release 2

Because the internals of Release 2 are so different from Release 1, all user files (including init files and ZMail init files) will need to be recompiled. The details are specified in the *Release 2 Conversion Notes*. The compiler, of course, will pick up the errors, but you will probably want to read the conversion advice to ease the transition. Note that many old constructs are still supported, even though newer ones are preferred.

Documentation

Because Release 2 has so many changes, there is a good amount of documentation to accompany it. Other documents describe some aspects of Release 2 in more detail; this is an overview of the major features of Release 2.

- *Common LISP Release Notes*: Describe the new features from Common LISP. Almost ~~of~~ all the document is now contained in the latest version of the *LISP Machine Manual*, but the *Notes* provide a convenient way to peruse the many features. These *Notes* are already provided in your manual set.
- *Release 2 Notes*: A comprehensive summary of Release 2 changes, with special attention to the package system and the **DEFSTRUCT** facility. All programming, user interface, editor, and site maintenance changes are documented here in detail.
- *Release Conversion Notes*: General advice for converting from Release 1 (Lambda System 1, CADR System 94) to this Release.
- *LISP Machine Manual, Sixth Edition*: The new edition, available as the LISP I and LISP II binders of the LMI documentation set. (It is also available separately, in one volume, as an orange-covered paperback.) The manual corresponds to the current release.

- *Common LISP:* Written by Guy Steele and available from Digital Press, this is the definitive document of Common LISP. It is not necessary, however, to acquire this book to use Common LISP with LMI software.

In the document package you receive with this release is documentation corresponding to further features included in Release 2.

Release 2 Notes

24-0100330-0001

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

This document corresponds to Release 2.0. It supersedes the *System 98 Release Notes.*

LMI Lambda is a trademark of Lisp Machine, Incorporated.

Principal writers of this document were Richard M. Stallman and Richard Mlynarik of MIT, and Robert Krajewski of LMI. Formatted with BoTeX.

Table of Contents

Introduction

This manual describes these changes to the system in Release 2:

- COMMON LISP Support: New functions; usage on the LMI Lambda.
- Incompatible Changes: Some incompatible changes are due to COMMON LISP. Many of the others involve the increase in size of the LAMBDA's address space, or rationalizing the behavior of some constructs (like **eval-when**) that sometimes had hard-to-understand conseqeunces.
- Compatible LISP Programming Changes: This covers certain basic changes that are neccessary for and compatible with COMMON LISP. Improvements have been made to the rubout handler, the "system" system, the file system interface, networks, and miscellaneous low-level constructs in ZETALISP.
- Window System Changes
- User Interface Changes: This covers the way the user interacts with the machine (usually the window system) and ZMail, which is a new part of the LAMBDA software in Release 2.
- Editor Changes
- Defstruct Changes
- Package Changes
- Site File Changes: If you are responsible for maintaining site information, you should read this.

The changes for the **defstruct** facility and the package system are noteworthy enough to be placed in their own chapters. Many changes related to COMMON LISP are documented in the *Common Lisp Release Notes*; the new edition of the *Lisp Machine Manual* also documents these changes.

Although the new edition of the *Lisp Machine Manual* has been produced only recently, there are some omissions and changes which are included in this document.

In this document, fonts are used to highlight words and text in a number of ways:

arg Here, the text mentions *arg*, which is an argument to the current definition. This is applicable for the documentation of functions, macros, operations, and special forms. *arg* could also stand for a value that was passed in some some pattern, as in **(:option** *arg***)**.

cons Here, we are mentioning a LISP construct in text.

c-X This same font is also used for mention names of characters, or sequences of keystrokes.

FOO Usually, this kind of font is used in examples that are set off from the rest of the text. If this font is used inline, it is usually to emphasise the way something could be entered into the Lisp Machine.

The "bucky" shift keys have the names **Control**, **Meta**, **Super**, and **Hyper**. The character formed by typing the **X** key while holding down the **Control** and **Super** keys can be written as **Control-Super-X**. Sometimes, this is abbreivated to use only the first letters of the bucky bits: **c-s-X**. If the **Shift** key is also used, the character is represent as **Control-Super-Shift-X** or **c-s-sh-X**, common

1. Common LISP Is Supported

Most of the differences between Common LISP and the traditional LISP machine dialect of LISP are compatible extensions. These extensions are available in all programs.

There are some incompatibilities between COMMON LISP and the traditional Lisp machine system, however. For the sake of existing programs, in most cases the system still works the traditional way.

In Release 2.0, the Lisp Machine will boot with ZETALISP being the default in the initial Lisp Listener. To use make a Lisp Listener use COMMON LISP reader syntax and functions by default, do

(setq *readtable* si:common-lisp-readtable)
or equivalently, **(common-lisp t)**

To use traditional ZETALISP syntax and functions, do

(setq *readtable* si:standard-readtable) or **(common-lisp nil)**

To make a file read in using COMMON LISP syntax, and use incompatible COMMON LISP functions when neccessary, put **Readtable: Common-Lisp** or **Readtable: CL** in the attribute list (the **-*-** line) of the file. To make a file always read using traditional syntax and functions, use **Readtable: Traditional** or just **Readtable: ZL**. The attribute **Syntax** is a synonym for **Readtable.**

The new editor command **m-X Set Readtable** is the recommended way to change the readtable attributes of a file (and the editor buffer). See section 8.11.1, page 82 for more information.

The readtable variable is bound at the top level of each process, so setting it applies only to the current process. Lisp Listeners check the variable before each form, so it works to set the variable while operating in the listener.

The current value of ***readtable*** is important for two reasons:

- First, there are some (small) differences between the syntax of COMMON LISP and ZETALISP.
- The readtable also has information about the symbols that name functions that are incompatible between the two dialects. The **CL** readtable will substitute **cli:listp** for **listp** at read time, for example. Because the support is implemented this way, there is no special variable that needs to tell the **Lambda** which way to behave at run time, in the case of an incompatibility.

A few things in COMMON LISP are not yet supported completely:

* transcendental functions of complex numbers
* "alternative" definitions (as macros) of nonstandard special forms
* **inline** and **notinline** declarations.

Some features of COMMON LISP are not yet supported exactly per the COMMON LISP spec:

* **&rest** arguments are not valid beyond the dynamic extent of a function; incorrect but legal values will result if such an argument is returned out of a function. To return a **&rest** argument as a true list, use the function **copy-list**.
* Zero-dimensional arrays may not be displaced or indirected.
* A number of COMMON LISP special forms are actually ZETALISP macros, a situation which could confuse some program-analyzing tools. **special-form-p** is COMMON LISP compatible, however; one should use that predicate and dispatch when needed before checking if a form is a macro-call.

* There are COMMON LISP names that are ZETALISP special forms and thus are not redefinable.
* If one makes a free reference to a variable in the interpreter, but does not declare it special, one gets thrown into the error handler. Currently, there is a proceed option that will subsequently allow this to happen without an error occuring. This proceed option will go away after Release 2.
* The ZETALISP implementation uses the tag values (**catch** and **throw** tags) of **t**, **nil**, and **0** for internal purposes. User programs should refrain from using these ~~tags~~.
* In order ~~to~~ for files with font changes in them to be read correctly, reader macros must use the functions **si:xr-xrtyi** and **si:xr-xruntyi** instead of **read-char** and **unread-char**. Use of the special functions will be made unneccessary in a future release.

2. Incompatible Changes

This chapter describes various changes to ZETALISP that are likely to affect user programs. Most of the COMMON LISP changes have either been taken in through the **cli** package and readtable mechanism or have been documented elsewhere.

2.1 Tail Recursion

The variable **tail-recursion-flag** has no effect on the behavior of the system in Release 2. It is unlikely that it will ever be reinstated.

2.2 Returning Storage

Stricter conventions need to be observed when using **return-storage** and **return-array** than are alluded to in the *Lisp Machine Manual.* It implies that returning the storage and then clobbering the pointer (using **without-interrupts**) is adequate protection against improper reference. In fact, the PDL-buffer management in the microcode makes this not so. The only guaranteed technique is the

```
(return-storage (prog1 pointer (setq pointer nil)))
```

idiom, which the compiler optimizes into code that actually clobbers *pointer before* it calls *return-storage.* It is probably not a terribly good idea to call these functions from the interpreter. (Actually, with the new garbage collector on the way, it's probably not a good idea to call them at all.)

2.3 Clarification on Fill Pointers

Two clarifications need to be made about fill pointers in ZETALISP:

- Apparently, the documentation for **fill-pointer** has been wrong since Release 1. The documentation (including the most recent edition of the **Lisp Machine Manual**) states that **fill-pointer** returns **nil** if it argument (a vector) does not have a fill pointer. However, it has always signalled an error. (The condition flavor is **sys:array-has-no-leader** because, in ZETALISP, fill pointers are implemented as element zero of the array's leader.)

  Interestingly enough, the actual behavior of **fill-pointer** agrees with what COMMON LISP specifies. So, only the documentation will change.
- Fill pointers are only defined for vectors. Arrays that are not vectors, of course, may have leaders, but element zero of such a leader will *not* be considered a fill pointer by any system array function; nor will the function **array-has-fill-pointer-p** return **t** for such an array.

2.4 Changes Related to Common Lisp

The following changes have been made to ZETALISP for compatibility with COMMON LISP.

2.4.1 Decimal Radix Has Become the Default

Base **10.** is now the default. This is a COMMON LISP change. However, it is still possible to specify the radix for each file individually. To avoid any difficulties, place **Base: 8;** in the attribute list (the -*- line) of any file which is supposed to be in octal.

To get back the old behaviour, do

```
(setq *print-base* 8. *read-base* 8. *nopoint nil)
```

2.4.2 Ratio Reading and Printing

Ratios used to be always read and printed using decimal notation when using ZETALISP syntax. Thus, **#5r-10\12** (or **#5r-10/12** in COMMON LISP syntax) now represents "minus five sevenths."

2.4.3 Case-Sensitive and -Insensitive String Comparison Functions.

The function **equal** now considers the strings **"A"** and **"a"** to be distinct. Use **equalp** if you wish to ignore case in the comparison. This is a COMMON LISP change.

Because **equal** has changed, the ZETALISP functions **member**, **assoc**, **rassoc**, **remove**, **delete**, and **find-position-in-list-equal** are affected when strings are involved. (Note that the epynomous COMMON LISP functions use **eql** as the default comparison function.) Also affected are hash tables which use **equal** as the comparison function.

char-equal and **string-equal** always ignore case. To consider case in comparing characters or strings this way, use **char=** for characters and the new function **string=** for strings.

Here are some ways to compensate for the change in **equal** when you have been using strings as "keys" in lists (as sets), association lists, or in **equal**-based hash tables.

— In lists:

Release 1: `(member key *known-words*)`
Release 2: `(cli:member key *known-words* :test #'string-equal)`

— In association lists:

Release 1: `(assoc person nickname-alist)`
Release 2: `(cli:assoc person nickname-alist :test #'string-equal)`

— In hash tables

Release 1: `(defvar *things* (make-equal-hash-table :size 42))`
Release 2:
```
 (defvar *things* (make-equal-hash-table
                     :size 42 :comparison-function #'string-equal))
```

samepnamep now considers case significant.

The functions of the **string-search** series now take an extra optional argument which says whether to consider case.

alphabetic-case-affects-string-comparison Variable

The old flag **alphabetic-case-affects-string-comparison** is now used only by the **%string-search** and **%string-equal** microcode primitives. These primitives now consider font significant as well as case when the flag is non-**nil**.

2.4.4 'COMPILE No Longer Needed in PROGN

Any **progn** encountered at top level by the compiler is now handled by treating each element as if it had been found at top level. Macros that used to expand into **(progn 'compile** *forms...*) can now expand into just **(progn** *forms...*)

2.4.5 Arrays Stored in Row-Major Order

Arrays used to be stored in column-major order. Now, they are stored in row-major order, which means that successive locations differ in the last subscript. The value of **sys:array-index-order** is now **t**; it was **nil** in Release 1. The change is *irreversible* and cannot be affected by changing the value of this variable. The change in storage layout does not affect user programs except when they do one of these four things:

1. Access screen arrays of windows using **aref**. Since the TV hardware has not been changed, the horizontal dimension is still the one that varies fastest in memory, which means it is now the second dimension rather than the first. The function **ar-2-reverse** (and its related versions for setting and getting locatives) was introduced in Release 1 so that code which was really using two-dimensional arrays in x/y terms would work no matter what the status of the index order was. If you used this function in such an application, you should have no problems.
2. Use multidimensional displaced arrays or arrays displaced to multidimensional arrays.
3. Deal with large multidimensional arrays and want to optimize paging behavior. For example, this piece of code will run with acceptable paging behavior in Release 1, but "pessimally" in Release 2 because it touches the elements of the array that are the farthest apart (in the first dimension) in the innermost loop.

```
(defvar *space* (make-array '(100 100 100)))

(defun make-space-centered ()
  (dotimes (iz 100)
    (dotimes (iy 100)
      (dotimes (ix 100)
        (setf (aref *space* ix iy iz)
              (make-space-vector (- ix) (- iy) (- iz)))))))
```

   Notice that this example is written starting with an outer loop concerned with the z-axis. Most programmers would probably use the opposite approach if they did not care about paging performance at all, since it is "natural" to nest loops according the order of the indices as they are written.
4. Store multidimensional arrays in QFASL files. A QFASL file records the elements of an array in the order they appear in storage. Therefore, if an array is dumped in an earlier system and loaded into Release 2, it will appear to be transposed.

The functions **ar-2-reverse**, **make-pixel-array** and others are provided to make it easier for you to change your code so that it works in both Release 2 and older system versions. See these functions in the *Lisp Machine Manual.*

2.4.6 &KEY Arguments

It is no longer ever an error to omit a keyword argument defined with **&key**. **&Optional** now has no effect on the treatment of **&key** arguments. This change is for COMMON LISP.

2.4.7 Common Lisp Package Conventions

For more information about changes to the package system, see section 5.1, page 57. The changes documented here simply give a very cursory overview of the most obvious visible changes.

2.4.7.1 Keywords

The **user** package is now just like all other packages in requiring that colons be used in front of keyword symbols. For example, you can no longer write just **tyi** instead of **:tyi** if your program is in **user**.

All symbols in the **keyword** package – that is to say, symbols that you write with a colon, such as **:string-out** –are now automatically set up to evaluate to themselves. Thus, you can now write

```
(send stream :tyi)
```
instead of
```
(send stream ':tyi)
```

This Common LISP change ought not to invalidate any reasonable programs.

2.4.7.2 Referring to Packages

In COMMON LISP you must refer to an internal symbol of another package by using two colons (`::`). ZETALISP does not actually require you to use this construct, and you are free to access internal symbols with a plain colon in a package prefix. You can also suppress local package nicknames with `#:`. As of Release 2, the situation is:

foo:bar Refers to external symbol of package **foo** (but actually you can use it for any symbol in **foo**).

foo::bar Refers to an internal symbol of package *foo*, in strict Common LISP.

foo#:bar Refers to external (really, any) symbol in the package whose global name or nickname is **foo**, ignoring any local nickname *foo* for any other package.

#:bar Makes an uninterned symbol named **bar**.

Currently, internal symbols in other packages are indicated with `::`. However, only the COMMON LISP readtable enforces the distinctions between external and internal symbols.

2.4.8 Local SPECIAL Declarations to Change in Meaning

For the sake of Common LISP, a **special** declaration within a function will have to be present in the construct (**let**, **prog**, etc.), which binds a variable in order to make the binding be special. Thus, for example,

```
(defun foo (a)
  (declare (special b))
  (let (b)
    ...))
```
will no longer make **b** special. Instead, you must write

```
(defun foo (a)
  (let (b)
    (declare (special b))
    ...))
```

where the local declaration appears just inside the construct that binds the variable in question.

A further unfortunate consequence of this is that **local-declare** cannot be used any more to make a binding special, as in

```
(defun foo (a)
  (local-declare ((special b))
    (let (b)
      ...)))
```

because this too would fail to put the declaration just inside the **let**.

To facilitate the changeover, this change has not actually been made. Local **special** declarations will still affect code just as they used to. However, any code that depends on this will get a warning reminding you to fix the code. The actual change will occur in a future system version.

Note that **local-declares** of **special** around an entire function, affecting arguments of the function, will continue to work. Also, if you are just examining or setting the variable, as in

```
(local-declare ((special a))
  ... (+ a 5) ...)
```

and not rebinding it, then your code will not be affected.

2.4.9 SELECTQ now uses EQL as its test function.

selectq formerly performed all its comparisons using **eq**. Since everything that is **eq** is also **eql**, and the only things which are **eql** but not **eq** are flonums, bignums and ratios (which should never have been used as tests for **selectq** in the past for this very reason) there should be no effect on any existing code. **selectq** and the COMMON LISP macro **case** are thus now identical.

2.4.10 CATCH and THROW

catch and **throw** used to be defined in a way which was compatible with Maclisp. **(catch** *form* **tag)** used to be what **(catch 'tag** *form***)** is now, and **(throw** *form* **tag)** used to be what **(throw 'tag** *form***)** is now. Since Maclisp itself has been issuing warnings for years saying to ~~use~~ ***catch**, this should cause no problems.

The implementation-related restrictions and general weirdness associated with the values from **catch** (a/k/a ***catch**) in older system versions have been fixed; **catch** now returns all the values from the last form executed (if no **throw** occurs) or else the values supplied by the second argument to **throw**.

In Release 2, **throw** can pass multiple values to **catch**: **catch** used to return exactly four arguments, of which the first one was a single value given to **throw**; the other three had complicated meanings. Now, **catch** returns any number of values: either the values thrown, or the values of the last form inside the **catch**, if no **throw** was done.

To throw more than one value, make the second subform of a ***throw** something which returns multiple values. Thus,

```
(catch 'foo (throw 'foo (values 'a 'b)))
```

returns the two values **a** and **b**.

In addition, **catch-all** now returns all the values of the body or all the values thrown, plus three more: the tag, action and count, *a la* ***unwind-stack**. (Yes, it is peculiar for a function to return *n* values followed by three specific ones, but it has to work that way.)

If you want to receive all these values, you should use **catch-all** within a **multiple-value-list** and then use **(butlast list 3)** to get the values thrown or returned and **(nleft 3 list)** to get the three specific values.

2.4.11 EVALHOOK/APPLYHOOK Incompatible Change

Evalhook and applyhook functions are now passed two additional arguments, which describe the interpreter environment that the evaluation or application was going to take place in. See the section on evaluation in the *Common Lisp Release Notes* for more information.

2.4.12 Changes to FORMAT control argument

~X (*HeX*)

Usage: ~*width,padchar,commachar***X** — Prints its argument in hexadecimal (analogous to **~O**, **~B** and **~D**). This command used to be used to insert spaces into the output.

Use ~*number-of-spaces***@T** to achieve the same result as the old ~*number-of-spaces***X** directive.

~F (*Floating point*)

Usage: ~*width,decimal-places,scale,overflowchar,padchar***F** — Prints a floating-point number in nonexponential notation. Multiplies by 10^{scale} before printing if *scale* is specified. Prints in *width* positions, with *decimal-places* digits after the decimal point. Pads on left with *padchar* if necessary. If the number doesn't fit in *width* positions, and *overflowchar* is specified, this command just fills the *width* positions with that character.

This directive used to just take one optional prefix control arg, which specified how many mantissa digits to print. This is the same as *decimal-places*+2 for the new **format**. Use ~,*n+2***F** to achieve the same result as the old ~*n***F** directive.

~E (*Exponential*)

Usage: ~*width,decimal-places,exponent-places,scale,overflowchar,padchar,exptchar***E** — Prints a floating-point number in exponential notation. Prints in *width* positions, with *exponent-places* digits of exponent. If *scale* (default is 1) is positive, prints *scale* digits before point, *decimal-places-scale*+1 after. If *scale* is zero, prints *decimal-places* digits after the point, and a zero before if there's room. If *scale* is negative, prints *decimal-places* digits after the point, of which the first -*scale* are zeros. If *exptchar* is specified, it is used to delimit the exponent (instead of "e" or whatever.) If *overflowchar* is specified, then if the number doesn't fit in the specified *width*, or if the exponent doesn't fit in *exponent-places* positions, the field is filled with *overflowchar* instead.

This directive used to just take one optional prefix control arg, which specified how many mantissa digits to print. This is the same as *decimal-places*+2 for the new **format**. Use ~,*n+2***E** to achieve the same result as the old ~*n***E** directive.

~G (*Generalized floating-point*)

Usage: ~*width,decimal-places,exponent-places,scale,overflowchar,padchar,exptchar***G** — Like **~E**, but if the number fits without an exponent, it is printed without one.

This command used to be used to go to a particular argument. Use ~*argument-number*@* to achieve the same result as the old ~*argument-number*G directive.

2.4.13 New Treatment of Square Roots

sqrt *number* Function

Return the square root of *number*, returning a complex number if needed. In Release 1, if *number* was negative, a condition of the flavor **sys:negative-sqrt** would be signalled. However, since this error never occurs in Release 2, the condition flavor has been flushed.

2.5 Pointer Fields Now 25 Bits; Flag Bit Gone

Each typed data word in LISP machine memory used to have one bit called the "flag bit", which was not considered part of the contents of the word. This is no longer so. There is no longer a flag bit; instead, the pointer field of the word is one bit larger, making it 25 bits in all.

This extra bit extends the range of integers that can be represented without allocation of storage, and also extends the precision of small-floats by one bit.

On the LMI Lambda processor, the maximum size of virtual memory is doubled. This is the primary reason for the change. Unfortunately, the CADR mapping hardware is not able to use the extra bit as an address bit, so the maximum virtual memory size on a CADR is unchanged.

The functions **%24-bit-plus**, **%24-bit-difference** and **%24-bit-times** still produce only 24 bits of result. If you wish to have a result the full size of the pointer field, however wide that is, you should use the functions **%pointer-difference** and **%pointer-times** (the last is new), and **%pointer-plus** to do addition. (The new **%pointer-** functions are documented in more detail in the *Lisp Machine Manual*.)

The functions **%float-double**, **%divide-double**, **%remainder-double** and **%multiply-fractions** use the full width of the pointer field.

The values returned by **sxhash** have not changed. They are always positive fixnums less than 2^{23}.

Because of the change in pointer format, short-floats now have 17 bits of mantissa, 7 bits of exponent magnitude, and 1 bit of exponent sign. (Short floats used to have 16 bits of mantissa.)

2.6 EVAL-WHEN Rationalized

The treatment of **(eval-when (load)** *forms...***)** by the compiler is now identical to the treatment of forms encountered with no **eval-when**. They are put into the file to be evaluated on loading, or compiled if they are **defuns**, and any macros defined are made available for expansion during the compilation.

As a consequence, you can no-op an **eval-when** by supplying **(load eval)** as its first argument. It is then equivalent in all cases to no **eval-when** at all.

Nested **eval-whens** now effectively intersect their list of times to evaluate. As a result,

```
(eval-when (compile load eval)
  compile-time-forms...
  (eval-when (load eval)
    forms...))
```

treats the *forms* in the ordinary manner, overriding the special treatment given to the *compile-time-forms*.

(eval-when (compile) (eval-when (load) *ignored-forms...***))**

does not do anything with the *ignored-forms*.

2.7 Change to SI:FULL-GC-INITIALIZATION-LIST

The **si:full-gc-initialization-list** initializations are now run before the garbage collection in **si:full-gc**, rather than after. A new initialization list, **si:after-full-gc-initialization-list**, is run after. The old list which now runs before GC can be requested with the keyword **:full-gc** in **add-initialization**, and the new list which runs after can be requested with **:after-full-gc**.

This change is for greater compatibility with Symbolics systems.

2.8 PROGV With More Variables Than Values

The function **progv** accepts a list of variables and a list of values. In the past, if the list of variables was longer, **nil** was used in place of the missing values. Now, in this case, the extra variables which have no corresponding values will be made "unbound." This is a COMMON LISP change.

2.9 %PAGE-STATUS Change

The subprimitive **%page-status** now returns the entire first word of the page hash table entry for a page, if the page is swapped in; or **nil** for a swapped-out page, or for certain low-numbered areas (which are all wired, so their pages' actual statuses never vary). The argument is an address in the page you are interested in–data type is irrelevant. The **%%pht1-** symbols in **SYS: SYS; QCOM LISP** are byte pointers you can use for decoding the value.

2.10 Character Bits Moved

The Control, Meta, Super, and Hyper bits now occupy a new position in character codes. This is so that they will not overlap the field used by the character's font number.

You can continue to use the byte pointers **%%kbd-control** to examine and set the bits; these byte pointers have different values now but your code will work anyway. No change to the source is needed.

2.11 Time Functions Return Exact Year

The functions **decode-universal-time**, **time:get-time** and **get-decoded-time** now return the correct year number (a number greater than 1900.) rather than the year number modulo 1900.

2.11.1 Primitive Printer Functions Changed

The functions **si:print-object** and **si:print-list** no longer accept the *slashify-p* argument. Instead, they look at the current value of ***print-escape***.

si:print-object *object prindepth stream* &optional *which-operations* Function
si:print-list *list prindepth stream* &optional *which-operations* Function

These are the primitive printer functions in the system. The recommended way to change the style of printed representation of all objects in the system is to **advise** these functions.

2.12 New List Matching Constructs

The syntax of **select-match** has been changed so as to avoid use of the construct **#?**. This is to avoid defining the construct **#?**, leaving it free for users to define. In addition, new instructions test extremely quickly whether a list has certain elements and then extract the others.

As before, **select-match** takes an expression for an object to be tested followed by any number of clauses to try. Each clause contains a pattern, a conditional form, and more forms that make the body of the clause. The first clause whose pattern matches the object and for which the conditional form produces a non-NIL value is the chosen clause, and its body is executed. The last clause can be an **otherwise** clause.

The change is that the pattern is now an expression made with the ' character, with commas indicating a variable in the pattern. For example, in

```
(select-match foo
  (`(a ,b ,b) (symbolp b) (print b))
  (otherwise (print foo)))
```

the first clause matches if **foo** is a list of three elements, the first being the symbol **a**, and the second and third being the same symbol. The second clause matches anything that slips through the first.

select-match and **list-match-p** also accept logical combinations of patterns, using **and**, **or**, and **not** at top level. Note that matching specifications for patterns actually containing the symbols **and**, **or**, and **not** will not conflict with the use of this feature, since **select-match** and **list-match-p** are special forms which interpret their arguments specially.

```
(defun hack-add-sub (x)
  (select-match x
    ((or `(+ ,y 0) `(- ,y 0) `(+ y)) t
     y)
    ((not (or `(+ . ,ignore) `(- . ,ignore)))
     (ferror nil "You lose"))
    (t x)))
```

Note that variables used in the patterns (such as **y** in the example above) are bound locally by the **select-match**.

You can get the effect of a single **select-match** pattern with **list-match-p**:

list-match-p *list pattern* Macro

Returns **t** if the value of list matches pattern. Any match variables appearing in pattern will be set in the course of the matching, and some of the variables may be set even if the match fails.

2.13 BREAK Arguments Changed

The function **break** is being changed to accept a format string and format arguments. It used to take an unevaluated first argument, normally a symbol, and simply print it.

To make the changeover easier, **break** evaluates its first argument by hand, unless it is a symbol–then its pname is used as the format string. However, the compiler issues a warning if you use **break** in the old way.

2.14 Macro Expander Functions Take Two Arguments

A macro's expander function used to be passed only one argument, the macro call to be expanded. Now it is passed a second argument as well. It is an "environment" object, and it is used to record the local macro definitions currently in effect.

Since many old macros are still compiled to accept only one argument, **macroexpand-1** is smart and will pass only one argument in such a case. So there is no need to alter or recompile your macro definitions now.

However, if you have anything else that calls macro expander functions directly, it must be changed to do what **macroexpand-1** does. The easiest way is to write

(call *expander-function* **nil** *form* **:optional** *environment***)**

If you define a macro using **macro** (instead of **defmacro**), you should change the arglist yourself to accept a second optional argument, even if it is just **ignore**.

Another change to these functions is that they return a second value which is **t** if any expansion was done.

2.15 SETF and LOCF Definitions Done Differently

You no longer use **setf** and **locf** properties to define how to do **setf** or **locf** on some kind of form. Instead, you use the macro **defsetf** to define how to **setf** it, and you do

(deflocf *function* ...**)**

to define how to do **locf** on it. See the section in the *Common Lisp Release Notes* that talks about **setf**.

One exception: **(defprop foo si:unsetfable setf)** still works, by special dispensation. Likewise for **si:unlocfable**. However, it is preferable to say, in the case of a function that should not allow **setf**, to say

(defsetf *function***)** or **(defun** *function* **si::nosetf)**

2.16 Y-OR-N-P And YES-OR-NO-P Arguments Changed

y-or-n-p *format-string* &rest *format-arguments* Function
yes-or-no-p *format-string* &rest *format-arguments* Function

These two functions now take just a format string and format arguments. They no longer accept the stream to use as an argument; they always use the value of ***query-io***.

If you used to pass two arguments, you must now bind ***query-io*** around the call instead.

2.17 COPY-FILE Takes Keyword Arguments

copy-file *filename new-name* &key (*error* **t**) (*copy-creation-date* **t**) (*copy-author* **t**) *report-stream* (*create-directories* **:query**) (*characters* **:default**) (*byte-size* **:default**) Function

Copies the file named *filename* to the file *new-name*.

characters and *byte-size* specify what mode of I/O to use to transfer the data. *characters* can be **t** to specify character input and output, **nil** for binary, **:ask** meaning ask the user which one, **:maybe-ask** meaning ask if it is not possible to tell with certainty which method is best, or **:default** meaning to guess as well as possible automatically.

If binary transfer is done, *byte-size* is the byte size to use. **:default** means to ask the file system for the byte size that the old file is stored in, just as it does in **open**.

The *copy-author* and *copy-creation-date* arguments say whether to set those properties of the new file to be the same as those of the old file. If a property is not copied, it is set to your login name (for the machine on which the target file resides) or the current date and time.

report-stream, if non-**nil**, is a stream on which a message should be printed describing the file copied, where it is copied to, and which mode was used.

create-directories says what to do if the output filename specifies a directory that does not exist. It can be **t** meaning "create the directory", **nil** meaning "treat it as an error", or **:query** meaning ask the user which one to do.

error, if **nil**, means that if an error happens then this function should just return an error indication.

If *filename* contains wildcards, multiple files are copied. The new name for each file is obtained by merging *new-name* (parsed into a pathname) with that file's truename as a default. The mode of copy is determined for each file individually, and each copy is reported on the report-stream if there is one. If error is **nil**, an error in copying one file does not prevent the others from being copied.

The value returned is a list with one element for each file which was to be copied. Each element is either an error object, if an error occurred copying that file (and error was **nil**), or a list (*old-truename new-truename characters*). The two truenames are those of the file copied and the newly created copy. characters is **t** if the file was copied in character mode. The value can also be just an error object, if an error happened in making a directory listing to find out which files to copy (for a wildcard pathname).

2.18 MAKE-PATHNAME Change

The meaning of the defaults argument to **make-pathname** is changed. Now all pathname components that are not specified or specified as **nil** are defaulted from the defaults, if you give defaults. If you do not give defaults, then the host alone defaults from ***default-pathname-defaults***, as it used to.

3. Compatible LISP Programming Changes

3.1 All Objects Except Symbols and Lists Are Constants

All arrays, instances, fefs, characters, closures, etc. now evaluate to themselves. Evaluating such objects used to be an error; this new behavior therefore cannot hurt anything. Keywords (see section 2.4.8, page 8), which are symbols in the **keyword** package, also evaluate to themselves.

The only kinds of objects that currently can evaluate to anything but themselves are symbols and lists. However, it is not guaranteed that no other kind of object will ever be defined to evaluate to other than itself.

3.2 Nonlocal GO and RETURN

You can now **go** or **return** from an internal **lambda** expression to the containing function. Example:

```
(prog ()
   (mapc #'(lambda (x) (if (numberp x) (return T)))
         inputs))
```

returns **t** if any element of **inputs** is a number. So does

```
(prog ()
  (mapc #'(lambda (x) (if (numberp x) (go ret-t)))
        inputs)
  (return nil)
 ret-t
  (return t))
```

3.3 Common Lisp Control Constructs BLOCK and TAGBODY

block takes a block name and a body:

(block *name body*...)

and executes the *body*, while allowing a **return-from** *name* to be used within it to exit the **block**. If the *body* completes normally, the values of the last body form are the values of the **block**.

A **block** whose *name* is **nil** can be exited with plain **return**, as well as with **(return-from nil)**.

block can be thought of as the essence of what named **progs** do, isolated and without the other features of **prog** (variable binding and **go** tags).

Every function defined with **defun** whose name is a symbol contains an automatically generated **block** whose name is the same as the function's name, surrounding the entire body of the function.

tagbody, on the other hand, is the essence of **go** tags. A **tagbody** form contains statements and tags, just as a **prog**'s body does. A symbol in the **tagbody** form is a tag, while a list is a statement to be evaluated. The value returned by a **tagbody** is always **nil**. **tagbody** does not have anything to do with **return**.

prog is now equivalent to a macro

```
(macro prog (form)
     (let* ((name (and (symbolp (cadr form)) (cadr form)))
              (vars (if name (caddr form) (cadr form)))
              (body (if name (cdddr form) (cddr form))))
       (if name
            '(block ,name
               (block nil
                 (let ,vars
                      (tagbody . ,body))))
          '(block nil
              (let ,vars
                (tagbody . ,body))))))
```

if we ignore the added complication of **progs** named **t** and **return-from-t**.

3.4 LEXPR-FUNCALL And APPLY Now Synonymous.

apply now accepts any number of arguments and behaves like **lexpr-funcall**. **lexpr-funcall** with two arguments now works the way **apply** used to, passing an explicit rest-argument rather than spreading it. This eliminates the old reasons why **lexpr-funcall** was not the best thing to use in certain cases, and paves the way for **apply** to translate into it. **lexpr-funcall** is now considered somewhat obsolete

3.5 :ALLOW-OTHER-KEYS As A Keyword Argument

:allow-other-keys has a special meaning as a keyword when passed to a function that takes **&key** arguments. If followed by a non-**nil** value, it prevents an error if any keyword is not recognized. Thus, given the function

```
(defun foo (&key a b) (list a b))
```

you would get an error if you do `(foo :a 5 :c t)` because **:c** is not recognized. But if you do

```
(foo :a 5 :c t :allow-other-keys t)
```

you get no error. The **:c** and its argument are just ignored.

3.6 GET and GETHASH with Three arguments.

get and **gethash** now take an optional third argument, which is a default value to be returned as the value if no property or hash table entry is found.

3.7 New Macros TYPECASE, PUSHNEW

There is now a **typecase** macro, compatible with COMMON LISP. See the *Lisp Machine Manual* for details.

pushnew pushes an element onto a list only if it was not there (using **cli:member**) before.

(pushnew *elt place***)**

compat

is equivalent to

```
(or (cli:member elt place)
    (setf place (adjoin elt place)))
```

except that *elt* and *place* are evaluated only once. The value returned by **pushnew** is the new list. The keywords **:key**, **:test**, and **:test-not** are accepted by **pushnew**; they get passed to along to **cli:member** to change the test for the "newness" of *elt*.

3.8 Microcoded Functions Interruptible

Many microcoded functions, including **last**, **memq**, **assq** and **get**, are now interruptible. This means in particular that if you pass a circular list to any of them you can now abort successfully.

3.9 Selecting a Returned Value

The function **nth-value** makes it convenient to select one of the values returned by a function. For example, **(nth-value 1 (foo))** returns the second of **foo**'s values. **nth-value** operates without consing in compiled code if the first argument's value is known at compile time.

nth-value *value-number expression* — Special form

Evaluates *expression*, then returns one of its values as specified by *value-number* (with **0** selecting the first value).

3.10 New types NON-COMPLEX-NUMBER and REAL

(typep *x* **(non-complex-number** *low high***))** returns **t** if *x* is a non-complex number (ie a floating-point number, a ratio or an integer) between *low* and *high*, the limits as usual being inclusive normally,or exclusive if they consist of a list of one element. Note that complex-numbers with an imaginary part of 0 are never of the type **non-complex-number**, since they are always of type **complex**. To account for this additional case, there is another new type, **real**, which is defined such that **(typep** *x* **(real** *low high***))** returns **t** if *x* is a either a non-complex number between *low* and *high*, or a complex number with a zero imaginary part and a real part lying between *low* and *high*.

3.11 Remainder, Log Functions Extended

“ *x y* — Function

remainder *x y* — Function

In Release 1, the remainder function only took integer (fixnum & bignum) arguments. In Release 2, it takes any sort of numeric arguments, and returns whatever is necessary to represent the exact result, as per the **Common Lisp** specification.

log *n* &optional (*base* **(exp 1)**) — Function

Return the base *base* logarithm of *n*, where *base* defaults to *e*. Previously, **log** only took one argument, and the base always *e*.

3.12 New Arithmetic Condition

Because COMMON LISP has a such a rich structure of numeric types, there are now cases (especially in the transcendental functions) where raising a number to a power may produce an undefined result.

sys:illegal-expt (sys:arithmetic-error) Condition

The condition **sys:illegal-expt** is signalled whenever an attempt is made to raise a number to a power in some case where the result is not defined. The condition supports the following operations:

:base-number
: The base of the exponentiation (the first argument to **expt**, for example).

:power-number
: The power of the exponentation (the second argument to **expt**, for example).

3.13 Macro Changes

Because of COMMON LISP, some subtle changes have occurred in the behavior of macros in interpreted code. Macro expander functions now take another argument, the lexical environment, to account for macros which need to be aware of the local macro definitions.

3.13.1 All Macros Are Displacing In Interpreted Code

All macros are displacing when encountered by **eval**. **defmacro-displace**, and so on, are now synonyms for **defmacro**, and so on. This is not exactly a compatible change for the interpreter. It was always made clear in the **Lisp Machine Manual** that part of a *compiled* function's behavior would be affected by the state of the macros it used at *compile*-time, no matter if the macro was displacing or not. On the other hand, it would matter in the interpreter whether the macro *was* displacing or not. If a macro defined with **macro** (not a COMMON LISP construct, by the way) or **defmacro** changed between invocations of a interpreted function that used it, the change would be seen by the function, because the macro would get expanded every time it is encountered by the interpreter. On the other hand, when a macro call uses a displacing macro, it is really expanded only once: the first time it is seen. So, if the macro changes, the changes will not noticed by the interpreter if it encounters a macro call which it has already expanded.

Note that this behavior is closer to being analogous to the compiler, but not exactly so. In order for that to be true, the interpreter would have to expand the macros in function-making forms (the **def** family and **lambda**) immediately. In general, COMMON LISP implementations are free to expand macros whenever they see fit, so users should be wary of depending on the implementation to notice changes in their macros when using interpreter.

3.13.2 MACROEXPAND-ALL

The function **macroexpand-all** is called like **macroexpand**. It expands macro definitions not only at the top level of a form but also in its subexpressions. It is never confused by a macro name, appearing at the start of a list, that is not a subexpression.

macroexpand-all *form* &optional *environment* Function

Expands macro definitions at all levels in *form* and returns the result. *environment* is used for finding local **macrolet** macro definitions; it is like the second argument to **macroexpand** (see previous page).

Only one value is returned.

3.13.3 New Function DEFF-MACRO

deff-macro &quote *function-spec* &eval *definition* Special form

defines *function-spec* as *definition*, just like **deff**. The difference comes in compiling a file, where the compiler assumes that **deff-macro** is defining a macro and makes the definition available for expansion during this compilation. **deff**, on the other hand, is just passed through to be evaluated when the file is loaded. To use **deff-macro** properly, *definition* must be a list starting with **macro** or a suitable subst function (a list starting with **subst** or a compiled function which records an interpreted definition which is a list starting with **subst**).

3.13.4 DEFINE-SYMBOL-MACRO

define-symbol-macro has not been implemented in LMI/MIT ZETALISP.

The effect of **(define-symbol-macro foo (print 'huh))** would be that evaluating the symbol **foo** would execute **(print 'huh)**. "Binding" such a symbol with **let** would probably have undefined or counterintuitive behavior.

If users find this useful or necessary for compatibility with Symbolics systems, it will be implemented.

3.14 Named Structure Operations

You can now **funcall** a named structure to invoke a generic operation on it, just as you would a flavor instance. In fact, you can have code which operates on named structures and flavor instances indiscriminately, if you make sure that the named structures you are using support whichever operations you plan to use.

For example,

```
(send *package* :describe)  ; Use send here to make it clear.
```

invokes the **:describe** operation on the current package, just as

```
(named-structure-invoke :describe *package*)
```

would do.

Invoking a named structure has not been made ultra-fast, but that can bedone in a future microcode release.

3.14.1 DEFSELECT and Named Structures

defselect, by default, defines the function to signal an error if it is called with a first argument not defined in the **defselect** (except for **:which-operations**, which is defined implicitly by **defselect**).

If you use **defselect** to define the handler function for a named structure type, and you use this default behavior, you will get errors at times when the system invokes operations that you may not know or care about, such as **:sxhash** or **:fasload-fixup**.

To avoid this problem, specify **ignore** as the default handler in the **defselect**. **ignore** accepts any arguments and returns **nil**. Also, **defselect-incremental** (see page 23) may be useful when defining a set of operations on a named structure.

3.14.2 Named Structure Operation :FASLOAD-FIXUP

The named structure operation **:fasload-fixup** is invoked by **fasload** whenever a named structure is created according to data in a QFASL file. This operation can do whatever is necessary to make the structure properly valid, in case just reloading it with all its components is not right. For most kinds of structures, this operation need not do anything; it is enough if it does not get an error.

3.15 DEFSUBST Preserves Order of Evaluation

It used to be the case that if a **defsubst**'s body used an argument more than once, or used its arguments out of order, the forms supplied as arguments would be evaluated multiple times or in the wrong order. This has been fixed. The arguments passed to a **defsubst** function will be evaluated exactly once, in the order they are written.

For example, after **(defsubst foo (a b) (cons b a))**, the reference **(foo x (setq x y))** used to turn into **(cons (setq x y) x)**, which is incorrect since it uses the new value of **x** twice. To be correct, the old value of **x** should be used for the second argument to **cons**.

Now, the expansion will be something effectively like

```
(let ((temp x)) (cons (setq x y) temp))
```

3.16 CAR-SAFE, Etc.

car-safe *x* Function
car-safe is like **car** when operating on a list. If *x* is not a list, **car-safe** returns **nil**. **car-safe** never gets an error.

cdr-safe *x* Function
cddr-safe *x* Function
nth-safe *n x* Function
nthcdr-safe *n x* Function
These are other functions which are analogous to **car-safe**. If *x* is not a cons, **nil** is returned.

3.17 Global Value Functions

There are now functions to use to examine or set the global binding of a variable, as opposed to the binding currently in effect. The global binding is the one that is in effect in all processes or stack groups that have not rebound the variable.

They work by forking off another process and examining or setting the variable in that process. The bindings of your own process are *not* visible in the other process, and that process establishes no bindings of its own, so references to the symbol there access the global binding.

symeval-globally *symbol* Function
Returns the global binding of *symbol*.

setq-globally *unevaluated-symbol value unevaluated-symbol value...* Function
Sets the global binding of each symbol to the corresponding *value*.

set-globally *symbol value* Function
Sets the global binding of *symbol* to *value*. *symbol* is an evaluated argument.

makunbound-globally *symbol* Function

Makes the global binding of **symbol** be void.

boundp-globally *symbol* Function

Returns **t** if the global binding of *symbol* is not void.

These functions are used primarily so that init files can set variables that are bound by the **load** function, such as **package** or **base**. If your init file does

```
(setq package (find-package 'foo))
```

this will be nullified as soon as **load** exits and its binding of **package** goes away. If you do

```
(setq-globally package (find-package 'foo))
```

the current binding established by **load** is actually not changed, but when the **load** exits and the global binding is in effect again, **foo** will be the current package.

3.18 LOCATION-MAKUNBOUND Takes Two Arguments

location-makunbound now takes a second, optional argument. This argument supplies a pointer value to use in the void marker that is stored.

A void location actually contains a pointer with data type **dtp-null**. This pointer is supposed to point to the object whose value or function definition is void. In the case of a symbol's value cell or function cell, the object would be the symbol itself.

location-makunbound makes the location point to whatever object you supply as the second value.

3.19 DEFSELECT-INCREMENTAL

defselect-incremental *fspec default-handler* Special form

With **defselect-incremental** you can define a **defselect** that starts out empty and has methods added to it incrementally with individual **defuns**.

You do **(defselect-incremental** *fspec default-handler***)** to define fspec as a select-method function that has no methods except the standard ones (**:which-operations**, **:operation-handled-p**, and **:send-if-handles**).

Then, to define the individual methods, use **defun** on function specs of the form **(:select-method** *fspec operation***)**. Note that the argument list of the **defun** must explicitly provide for the fact that the operation will be the first argument; this is different from what you do in an ordinary **defselect**. Example:

```
(defselect-incremental foo ignore)
; The function ignore is the default handler.
(defun (:select-method foo :lose) (ignore a)
  (1+ A))
```

defines FOO just like

```
(defselect (foo ignore)
  (:lose (a) (1+ a)))
```

The difference is that reevaluating the **defselect** gets rid of any methods that used to exist but have been deleted from the **defselect** itself. Reevaluating the **defselect-incremental** has no such effect, and reevaluating an individual **defun** redefines only that method.

3.20 :NO-ERROR Clauses in CONDITION-CALL

The last clause in a **condition-call** or **condition-call-if** may now be a **:no-error** clause. This looks and works about the same as a **:no-error** clause in a **condition-case**: it is executed if the body returns without error. The values returned by the body are stored in the variables that are the elements of the list that is the first argument of the **condition-call**, and the values of the last form in the clause are returned by the **condition-call** form itself.

3.21 Top Level Forms Specially Treated In The Compiler

Following is a partial list of symbols, which, when appearing as the first element of a top-level form, will cause that form to be treated specially by the compiler. Only those whose meanings have changed, or require clarification, are listed here.

progn Treat all following forms as if they also were at top level. Note that in Maclisp and in Release 1 and earlier, it was necessary for the first form of the body to be **'compile** for this to happen. This curious behaviour has been eliminated.

proclaim The arguments are evaluated, and relevant proclamations (such as **special**, **notinline**) are used in the remainder of the compilation. This is as if the form were contained within a **(eval-when (eval compile load) ...)**

export import in-package make-package shadow
shadowing-import unexport unuse-package use-package
These perform their relevant actions as if the form contained within a `(eval-when (eval compile load) ...)`.

require Ditto; this is relevant for COMMON LISP modules.

To cause a form not to be treated specially at top-level by the compiler, enclose it in an **eval-when**. Eg:

```
(eval-when (load)  ; don't want this package to be consed up when we're just compiling
  (make-package "lossage" :use nil :size 69))
```

3.22 Compiler Optimization Changed

Many compiler optimizers have been reimplemented, and should often produce better code. The most visible change is that any form is only optimized once, no matter where it appears. (In earlier systems, a form could sometimes be optimized twice, which could produce duplicate compiler warnings) In addition, the order in which optimizations are carried out has changed. All the arguments to a function are optimized before the call to the function on those arguments, unless the "function" is a macro or special form, in which case it is expected to take responsibility for doing its own optimizations.

3.23 TV:BITBLT-CLIPPED

tv:bitblt-clipped is just like **tv:bitblt**, except that if you specify transfers that include points outside the bounds of either the source or destination array, only the part of the transfer that is within the bounds of both arrays will take place.

The height and width you specify must be positive.

3.24 %BLT-TYPED. Proper Use of Pointer Subprimitives

%blt-typed is called just like **%blt** and does about the same thing: it copies any number of consecutive memory words from one place in memory to another. The difference is that **%blt** is only properly used on data that contains no pointers to storage, while **%blt-typed** is only properly used on boxed data.

Both **%blt** and **%blt-typed** can be used validly on data that is formatted with data types (boxed) but whose contents never point to storage. This includes words whose contents are always fixnums or small flonums, and also words that contain array headers, array leader headers, or FEF headers. Whether or not the machine is told to examine the data types of such data makes no difference since, on examining them, it would decide that nothing needed to be done.

For unboxed data (data that is formatted so as not to contain valid data type fields), such as the inside of a numeric array or the instruction words of a FEF, only **%blt** may be used. If **%blt-typed** were used, it would examine the data type fields of the data words and would probably halt due to an invalid data type code.

For boxed data that may contain pointers, only **%blt-typed** may be used. If **%blt** were used, it would appear to work, but problems could appear mysteriously later because nothing would notice the presence of the pointer there. For example, the pointer might point to a bignum in the number consing area; moving it with **%blt** would fail to copy it into a nontemporary area. Then the pointer would become invalidated the next time the number consing area was emptied out. There could also be problems with lexical closures and with garbage collection.

%p-store-tag-and-pointer should be used only for storing into boxed words, for the same reason as **%blt-typed**: the microcode could halt if the data stored is not valid boxed data.

%p-dpb and **%p-dpb-offset** should be used only when the word being modified does not contain a pointer. It may be an unboxed word, or it may be a boxed word containing a fixnum, small-flonum or array header. The same goes for **%p-deposit-field** and **%p-deposit-field-offset**.

Here are some new subprimitives that test values for pointerhood.

%pointerp *object* Function

returns non-**nil** if *object* points to storage. For example, **(%pointerp "foo")** is **t**, but **(%pointerp 5)** is **nil**.

%p-pointerp *location* Function

returns non-**nil** if the contents of the word at *location* points to storage. This is similar to **(%pointerp (contents** *location***))**, but the latter may get an error if *location* contains a forwarding pointer, a header type, or an void marker. In such cases, **%p-pointerp** will correctly tell you whether the header or forward points to storage.

%p-pointerp-offset *location offset* Function

similar to **%p-pointerp** but operates on the word *offset* words beyond *location*.

%p-contents-safe-p *location* Function

returns non-**nil** if the contents of word *location* are a valid Lisp object, at least as far as data type is concerned. It is **nil** if the word contains a header type, a forwarding pointer, or an unbound marker. If the value of this function is non-**nil**, you will not get an error from **(contents** *location***)**.

%p-contents-safe-p-offset *location offset* Function
similar to **%p-contents-safe-p** but operates on the word *offset* words beyond *location*.

%p-safe-contents-offset *location offset* Function
returns the contents of the word *offset* words beyond *location* as accurately as possible without getting an error.

- If the data there are a valid Lisp object, it is returned exactly.
- If the data are not a valid Lisp object but do point to storage, the value returned is a locative which points to the same place in storage.
- If the data are not a valid LIsp object and do not point to storage, the value returned is a fixnum with the same pointer field.

%pointer-type-p *data-type* Function
returns non-**nil** if the specified data type is one which points to storage. For example, **(%pointer-type-p dtp-fix)** returns **nil**.

3.25 Growing the Stack

When the PDL (stack) overflows, a condition is signalled, and the process usually falls in the debugger. If a function is going to use up a lot of stack space, then the function **eh:require-pdl-room** can be used to grow the stack, and thus avoid the debugger.

eh:require-pdl-room *regpdl-space specpdl-space* Function
Makes the current stack group larger if necessary, to make sure that there are at least *regpdl-space* free words in the regular pdl, and at least *specpdl-space* free words in the special pdl, not counting what is currently in use.

3.26 Flavor Changes

3.26.1 Delaying Flavor Recompilation

si:*dont-recompile-flavors* Variable
Normally the system recompiles combined methods automatically when you make a change that requires this. If you plan to make more than one change, you might wish to recompile only once. To do this, set the variable **si:*dont-recompile-flavors*** non-**nil** before you make the changes. Then set it back to **nil**, and use **recompile-flavor** to perform the appropriate recompilations.

3.26.2 Method Combination Improvements

When using method combination types such as **:list**, **:progn**, **:append** and **:pass-on**, which formerly allowed any number of untyped methods and nothing else, you can now use the method combination type keyword as a method type. For example, when using **:or** combination for operation **:doit**, you can now define a method **(myflavor :or :doit)** as well as **(myflavor :doit)**. The method is combined the same way whichever name you use. However, when the operation is invoked, all the typed methods are called first, followed by all the untyped methods.

There is no longer a limit of three values passed back from the primary method when **:after** methods are in use. As many values as the primary method chooses to return will be passed back to the ultimate caller.

3.26.3 Undefinition

undefflavor *flavor-name* Function

Removes the definition of *flavor-name*. Any flavors that depend on it are no longer valid to instantiate.

3.26.4 New DEFFLAVOR options

Several new keyword options for **defflavor** have been added for Release 2.

:instance-area-function *function*

This feature can control in which area flavor instances are consed, on a per-flavor basis, by giving a flavor an instance-area function. This is a function which will be called whenever the flavor is instantiated, and expected to return the area to cons in (or **nil**, if it has no opinion). The function is passed one argument, the init-plist, so if you want to have an init option for the caller to specify the area, the instance-area function can use **get** to get the value the caller specified.

The instance-area function is inherited by flavors which use this one as a component.

:required-init-keywords *init-keywords...*

This option specifies that each of the keywords in *init-keywords* must be provided when trying to make an instance of this flavor. Then, whenever the flavor (or any flavor that depends on it) is instantiated, it will be an error if any of those init keywords fails to be specified. For example, after

```
(defflavor foo (a) :inittable-instance-variables
   (:required-init-keywords :a))
```

it is an error to do **(make-instance 'foo)** since the **:a** keyword is not provided.

:instantiation-flavor-function *function-name*

This allows a flavor to compute what flavor **make-instance** will actually use. When a flavor which uses this option is passed to **make-instace**, it calls a function to decide what flavor it should really instantiate (not necessarily the original flavor).

When **(make-instance 'foo** *keyword-args***)** is done, the function specified is called with two arguments: the flavor name specified (**foo** in this case) and the init plist (the list of keyword args). It should return the name of the flavor that should actually be instantiated.

Note that the instantiation flavor function applies only to the flavor it is specified for. It is not inherited by dependent flavors.

:run-time-alternatives *clauses...*

:mixture *clauses...*

A run-time-alternative flavor is a way to define a collection of similar flavors, all built on the same base flavor but having various mixins as well, and choose which one to instantiate based on init options. (This is implemented using the **:instantiation-flavor-function** feature.)

A simple example would be:

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
    (:big big-foo-mixin))
  (:init-keywords :big))
```

Then, **(make-instance 'foo :big t)** will get you an instance of a flavor whose components are **big-foo-mixin** as well as **foo**. But **(make-instance 'foo)** or **(make-instance 'foo :big nil)** will get you an instance of **foo** itself. The clause **(:big big-foo-mixin)** in the **:run-time-alternatives** says to incorporate **big-foo-mixin** if **:big**'s value is **t**, but not if it is **nil**.

You can have several clauses in the **:run-time-alternatives**. Each one is processed independently. Thus, you could have keywords **:big** and **:wide** independently control two mixins and get four possibilities.

You can test for values of a keyword other than just **t** or **nil**. The clause

```
(:size (:big big-foo-mixin) (:small small-foo-mixin)
       (nil nil))
```

allows the value for the keyword **:size** to be **:big**, **:small**, or **nil** (or omitted). If it is **nil** or omitted, no mixin is used (that's what the second **nil** means). If it is **:big** or **:small**, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values you check for can be anything, but **eq** is used to compare them.

You can also have the value of one keyword control the interpretation of others by inserting clauses within clauses. After the place where you put the mixin name or **nil** for no mixin, you can put other clauses which specify keywords and their interpretation. These other clauses are acted on only if the containing alternative is chosen. For example, the clause

```
(:etherial (t etherial-mixin)
           (nil nil
       (:size (:big big-foo-mixin) (:small small-foo-mixin)
         (nil nil))))
```

says to consider the **:size** keyword only if **:etherial** is **nil**.

3.27 File System Changes

3.27.1 Additional Arguments to FS:PARSE-PATHNAME

fs:parse-pathname *thing* &optional *with-respect-to defaults* (*start* **0**) *end junk-allowed* Function

Parses *thing* into a pathname and returns it. *thing* can be a pathname, a string or symbol, or a Maclisp-style namelist. If it is a pathname, it is returned unchanged, and the other arguments do not matter. *with-respect-to* can be **nil** or a host or a host-name.

- If it is not **nil**, the pathname is parsed for that host and it is an error if the pathname specifies a different host.

- If *with-respect-to* is **nil**, then defaults is used to get the host if none is specified. *defaults* may be a host object in this case.

start and *end* are indices specifying a substring of thing to be parsed. They default to 0 for start and **nil** (meaning end of thing) for end.

- If junk-allowed is non-**nil**, parsing stops without error if the syntax is invalid, and this function returns **nil**. The second value is the index in thing at which parsing stopped, which is the index of the invalid character if there was invalid syntax.
- If junk-allowed is **nil**, invalid syntax signals an error.

3.27.2 Merging Pathname Components

fs:merge-pathname-components *pathname* &optional *defaults* &key *default-name always-merge-name default-type always-merge-type default-version always-merge-version* Function

This function extends the functionality of both the commonlisp function **merge-pathnames** and the old Lisp Machine function **fs:merge-pathname-defaults**.

merge-pathname-components defaults components that are of *pathname* which are **nil**, and returns the defaulted pathname. *defaults* is a pathname or a defaults-list to get defaults from. If non-**nil**, *default-name, default-type* and *default-version* respectively are used as the defaults for the name, type and version components if those components are not supplied by *pathname*. Otherwise, those components are defaulted from *defaults* in the usual manner. *always-merge-name, always-merge-type* and *always-merge-version* respectively mean that the version and type components should always be merged in (from either *default-xxx* or from *defaults*) even if the relevant component is already specified by *pathname*.

(merge-pathnames pathname defaults default-version) is thus equivalent to:

```
(merge-pathnames-components pathname default
                            :default-version default-version
                            :always-merge-version t)
```

since COMMON LISP specifies that the *default-version* argument to *merge-pathnames* is merged into the resulting even if *pathname* already had a version component.

fs:merge-pathname-components differs from **fs:merge-pathname-defaults** in that it performs *only* the merging operation of filling **nil** components of one pathname with (possibly **nil**) components from the defaults, whereas **fs:merge-pathname-defaults** will *never* return a pathname with a **nil** name or type component.

fs:merge-pathname-defaults is thus a function useful for defaulting a pathname that the user has just entered for some purpose, such as to be read. **fs:merge-pathname-componments** will perform a single merging (and *may* return a pathname which is not accceptable for performing file operations upon — such as a pathname with a name of **nil**.) It is useful for programs which need to manipulate filenames in an exact manner (such as the file server) and do not want any user-oriented heuristics happening "behind its back." It ignores such variables as ***always-merge-type-and-version** and ***name-specified-default-type***, which **fs:merge-pathname-defaults** uses. **merge-pathnames** is a simpler version of **fs:merge-pathname-components** which COMMON LISP implementations understand.

A typical use of **fs:merge-pathname-components** is

```
(setq qfasl-file
      (fs:merge-pathname-components qfasl-file lisp-file
                                    :always-default-version t
                                    :default-type :qfasl
                                    :always-default-type t))
```

which will produce a file whose version is the same as that of **lisp-file** and whose type is always **qfasl**, and whose other components are the (perhaps **nil**) results of merging the components of **lisp-file** with **fasl-file**.

Some examples:

```
(setq pn1 (make-pathname :host twenex-host :name "FOO" :version 259))
  => #⊂FS::TOPS20-PATHNAME "TWENEX:FOO.⇌.259"⊃
(setq pn2 (make-pathname :host twenex-host :device "DP" :type :TEXT))
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:⇌.TEXT.⇌"⊃

(fs:merge-pathname-components pn1 pn2)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn1 pn2 :default-version 5)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn1 pn2 :default-version 5
                              :always-merge-version t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.5"⊃
(fs:merge-pathname-components pn1 pn2 :default-version 5
                              :default-type :lisp
                              :always-merge-version t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.LISP.5"⊃

(fs:merge-pathname-components pn2 pn1)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn2 pn1 :always-merge-type t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.⇌.259"⊃ ; merges in null type!
(fs:merge-pathname-components pn2 pn1 :default-type :lisp)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn2 pn1 :default-type :lisp
                              :always-merge-type t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.LISP.259"⊃
```

3.27.3 Logical Hosts

Logical hosts can now have their translations specified by pattern matching, instead of using just literal directory names. A translation now consists of a pair of pathnames or namestrings, typically containing wildcards. Unspecified components in them default to **:wild**. The *from*-pathname of the translation is used to match against the pathname to be translated; if it matches, the corresponding *to*-pathname is used to construct the translation, filling in its wild fields from the pathname being translated as in the **:translate-wild-pathname** operation.

Most commonly the translations contain pathnames that have only directories specified, everything else wild. Then the other components are unchanged by translation.

Each translation is specified as a list of two strings. The strings are parsed into pathnames and any unspecified components are defaulted to **:wild**. The first string of the pair is the source pattern; it is parsed with logical pathname syntax. The second string is the target pattern, and it is parsed with the pathname syntax for the specified physical host.

For example, suppose that logical host **FOO** maps to physical host **BAR**, a Tops-20, and has the following list of translations:

```
(("BACK;" "PS:<FOO.BACK>")
 ("FRONT;* QFASL" "SS:<FOO.QFASL>*.QFASL")
 ("FRONT;" "PS:<FOO.FRONT>"))
```

Then all pathnames with host **FOO** and directory **BACK** translate to host **BAR**, device **PS** and directory **<FOO.BACK>** with name, type and version unchanged. All pathnames with host **FOO**, directory **FRONT** and type **QFASL** translate to host **BAR**, device SS, directory **<FOO.QFASL>** and type **QFASL**, with name and version unchanged. All other pathnames with host **FOO** and directory **FRONT** map to host **BAR**, device **PS** and directory **<FOO.FRONT>**, with name, type and version unchanged. Note that the first translation whose pattern matches a given pathname is the one that is used.

Another site might define **FOO**'s to map to a Unix host **QUUX**, with the following translation list:

```
(("BACK;" "//nd//foo//back//")
 ("FRONT;" "//nd//foo//front//"))
```

This site apparently does not see a need to store the **QFASL** files in a separate directory. Note that the slashes are duplicated to quote them for Lisp; the actual namestrings contain single slashes as is usual with Unix.

If the last translation's source pattern is entirely wild, it applies to any pathname not so far handled. Example:

```
(("BACK;" "//nd//foo//back//")
 ("" "//nd//fool//*//"))
```

fs:add-logical-pathname-host *logical-host physical-host translations* Function
fs:set-logical-pathname-host *logical-host* &key *physical-host translations* Function

Both create a new logical host named *logical-host*. Its corresponding physical host (that is, the host to which it should forward most operations) is *physical-host*. *logical-host* and *physical-host* should both be strings. *translations* should be a list of translation specifications, as described above. The two functions differ only in that one accepts positional arguments and the other accepts keyword arguments. Example:

```
(add-logical-pathname-host "MUSIC" "MUSIC-10-A"
      '(("MELODY;" "SS:<MELODY>")
        ("DOC;" "PS:<MUSIC-DOCUMENTATION>")))
```

This creates a new logical host called **MUSIC**. An attempt to open the file

`MUSIC:DOC;MANUAL TEXT 2`
will be re-directed to the file
`MUSIC-10-A:PS:<MUSIC-DOCUMENTATION>MANUAL.TEXT.2`

(assuming that the host **MUSIC-10-A** is a TOPS-20 system).

fs:make-logical-pathname-host *name* Function

Requests that the definition of logical host *name* be loaded from a standard place in the file system: namely, the file **SYS: SITE:** *name* **TRANSLATIONS**. This file is loaded immediately with **load**, in the **fs** package. It should contain code to create the logical host; normally, a call to **fs:set-logical-pathname-host** or **fs:add-logical-pathname-host**, above.

The same file is automatically reloaded, if it has been changed, at appropriate times: by **load-patches**, and whenever site information is updated.

3.27.4 :DEVICE-WILD-P, etc., Pathname Operations

The operation **:device-wild-p** operation on a pathname object is defined to return non-**nil** if the pathname's device component contains a wildcard.

:directory-wild-p, **:name-wild-p**, **:type-wild-p** and **:version-wild-p** are similar, for their respective pathname components.

3.27.5 WITH-OPEN-FILE-SEARCH

with-open-file-search is a new macro for opening a file and trying various pathnames until one of them succeeds. The pathnames tried differ only in their type components.

with-open-file-search (*streamvar* (*operation defaults auto-retry*) *types-and-pathname . options*) &body *body* Macro

Tries opening various files until one succeeds; then binds streamvar to the stream and executes body, closing the stream on exit.

types-and-pathname should evaluate to two values, the first being a list of types to try and the second being a pathname, called the base pathname. Each pathname to try is made by merging the base pathname with the defaults defaults and one of the types. options should evaluate alternately to keywords and values that are passed to **open**.

If all the names to be tried fail, a **fs:multiple-file-not-found** error is signaled. *operation* is provided just so that the **:operation** operation on the error object can return it. It is usually the user-level function for that the **with-open-file-search** is being done.

If *auto-retry* is non-**nil**, an error causes the user to be prompted for a new base pathname. The entire set of types specified is tried anew with this pathname.

3.27.6 New :PROPERTIES operation on file streams

Sending a **:properties** message to a file stream returns two values: a property list, like the kind which is a element of the list returned by **fs:directory-list**, and a list of settable properties. There is the usual optional *error-p* argument, as well. This operation uses a new **PROPERTIES** command in the Chaosnet file protocol, so it may not work with servers running old software.

3.27.7 Creating Links

fs:create-link *link-name link-to* &key (*error* **t**) Function

Creates a link named *link-name* that points to a file named *link-to*. An error happens if the host specified in *link-name* does not support links (or because of any of the usual problems that can happen in creating a file).

3.27.8 New :SUBMIT option for opening files

A new **:submit** option available in **open** and other constructs that use **open** (such as **with-open-file** and friends). When this option is t and the direction is **:output**, the file is submitted for batch processing on the host. The **:submit** option is currently effective on VMS and Twenex Chaosnet FILE servers.
An example:

```
(defun retrieve-twenex-file (F)
  "Submit a batch job that will retrieve the file F"
  (setq f (fs:parse-pathname F))
  (with-open-file (s (send (send (send (send f :host)
                                        :sample-pathname)
                                  :homedir)
                           :new-pathname :name "_retrieve"
                           :type "ctl" :version :newest)
                  :direction :output
     :temporary t :submit t)
    (format s "@retrieve ~A~%" (send f :string-for-host))))
```

3.27.9 File-Reading Special Forms

These two special forms are a straightforward aid in writing code that reads Lisp forms from a file, while obeying the attribute list of the file. (The attribute list file's pathname object or generic pathname object is *not* updated with this special form.)

fs:reading-from-file (*form file*) *body...* Macro
fs:reading-from-file-case (*form file error*) *clauses...* Macro

The following form prints out the result of evaluating each form in the file:

```
(fs:reading-from-file (form file)
   (format t "Values from ~S are: " form)
   (format:print-list t "~S" (multiple-value-list (eval form))))
```

The body of the form is executed for every form in the file. **fs:reading-from-file-case** is a cross between **fs:reading-from-file** and **with-open-file-case**, except there's an additional argument *error* argument (which is bound to the error object) for use in the clauses.

```
(fs:reading-from-file-case (form file error)
    ((fs:file-not-found
       (format t "~&Options file ~A not found, using default values."
                 file))
         (fs:file-error
           (format t "~&Error: ~A" (send error :report-string)))
    (:no-error (process-option form))))
```

Here, the **:no-error** clause, which must be present and consists of any number of forms, is executed for every form in the file. (But any error clause would be executed just once.) The value of the error variable is not defined in the **:no-error** clause

The following function does the actual work of getting the attribute list of a stream.

fs:extract-attribute-bindings *stream* Function

returns two values: a list of variables, and a corresponding list of values to bind them to, to set up an environment to read data from *stream* in accordance with *stream*'s attribute list.

3.27.10 VMS Default Device

The "primary device" for VMS hosts now defaults **USRD$** rather than**SYS$SYSDSK**. However, it is also possible specify what the default device using the **:host-default-device-alist** option in the site description; see section 9.3.2.2, page 85 for more details.

3.27.11 Improved File Error Handling

When there is an error accessing a file and the system asks for a new pathname, you now have the option of entering the debugger instead. Simply type **End**.

3.28 String Changes

string-length can give you the length of anything **string** can coerce into a string. In Release 2, it would not accept characters or symbols.

make-string *length* &key *initial-element* &allow-other-keys Function

The "other" keyword which is the most interesting to use here is **:fill-pointer**. (Only the **:initial-element** keyword is supported in COMMON LISP.)

Clarification: note that **string-append** and the related functions do not create strings with fill pointers.

3.29 New Keyword Arguments to MAKE-PLANE

The arguments *initial-dimensions* and *initial-origins* are now accepted. You can use them to specify which part of the infinite plane the initially allocated storage should be for.

make-plane *rank* &key *type default-value extension initial-dimensions initial-origins* Function

Creates and returns a plane. *rank* is the number of dimensions. The keyword arguments are

type The array type symbol (e.g. **art-1b**) specifying the type of the array out of which the plane is made.

default-value
The default component value as explained above.

extension The amount by which to extend the plane, as explained above.

initial-dimensions
nil or a list of integers whose length is *rank*. If not **nil**, each element corresponds to one dimension, specifying the width to allocate the array initially in that dimension.

initial-origins
: **nil** or a list of integers whose length is *rank*. If not **nil**, each element corresponds to one dimension, specifying the smallest index in that dimension for which storage should initially be allocated.

Example:

```
(make-plane 2 :type 'art-4b :default-value 3)
```

creates a two-dimensional plane of type **art-4b**, with default value **3**.

3.30 New Resource Features

A new option to **defresource** called **:deinitializer** has been added. The value is either a function of one argument, or a form containing a reference to the variable **object**. The deinitializer is called when an object is deallocated. There are two storage-related reasons for specifying a deinitializer:

1. Sometimes, a resource may have pointers to objects that are only valid (with respect to the Lisp Machine storage conventions) when the object is allocated. When the object is deallocated, some objects to which it might point may no longer be around. This situation only arises when using dangerous features such as pointer-making subprimitives or temporary areas.

   Even when an object of a resource is deallocated, the garbage collector can still find it. Thus, "dangerous" pointers should be thrown away by the deinitializer.
2. An object of a resource might the only object to point to another big object that should otherwise be freed by the garbage collector.

In either case, the deinitializer will deference the objects to which it points by setting slots of itself to **nil**.

There are also two new operations on resources:

map-resource *function resource-name* &rest *extra-args* Function

Operates with function on each object created in resource resource-name.

Each time function is called, it receives three fixed args, plus the extra-args. The three fixed args are:

- an object of the resource;
- **t** if the object is currently allocated ("in use")
- the resource data structure itself.

deallocate-whole-resource *resource-name* Function

Deallocates each object in resource *resource-name*. This is equivalent to doing **deallocate-resource** on each one individually. This function is often useful in warm-boot initializations.

3.31 Flushed Processes

A flushed process now has the symbol **si:flushed-process** as its wait function. This function is equivalent to **false** in that it always returns **nil**, but it is distinguishable from **false**. Thus, flushed processes can reliably be distinguished from those that have done **process-wait-forever**.

3.32 Indenting Format Directive

format output within a ~→ ... ~← construct is printed with each line indented to match the indentation that was current when the ~→ was reached.

3.33 Input Read Function Changes

3.33.1 READLINE and Friends

readline and **readline-trim** have been extended to return a second value. This value is **t** if end-of-file was encountered.

Note that end-of-file can still be an error if encountered at the beginning of the line, and this is still controlled by the *eof-option* argument. But if the function does return, the second argument always says whether there was an end-of-file.

The new function **readline-or-nil** is like **readline-trim** except that it returns **nil** rather than "" if the input line is empty or all blank.

3.33.2 New Function READ-DELIMITED-STRING

read-delimited-string &optional *delimiter stream eof rubout-handler-options buffer-size* Function

Reads input from *stream* until a delimiter character is reached, then returns all the input before but not including the *delimiter* as a string. *delimiter* is either a character or a list of characters that all serve as delimiters. It defaults to the character **End**. *stream* defaults to the value of ***standard-input***.

If eof is non-**nil**, then end-of-file on attempting to read the first character is an error. Otherwise it just causes an empty string to be returned. End-of-file once at least one character has been read is never an error but it does cause the function to return all the input so far.

Input is done using rubout handling and echoing if stream supports the **:rubout-handler** operation. In this case, *rubout-handler-options* are passed as the options argument to that operation.

buffer-size specifies the size of the string buffer to allocate initially.

Three values are returned:

- the string of input read;
- a flag which is **t** if input ended due to end of file;
- and the delimiter character which terminated input (or **nil** if end of file was reached).

:run-time-alternatives can also be called **:mixture**, for compatibility with other systems.

3.33.3 :STRING-LINE-IN Stream Operation

:string-line-in is a new standard input stream operation, supported by all the input streams provided by the system. It fills a user-supplied buffer with text from the stream until either the buffer is full, end of file is reached, or a **Return** is found in the input. If input stops due to a **Return**, the **Return** itself is not put in the buffer.

Thus, this operation is nearly the same as **:string-in**, except that **:string-in** always keeps going until the buffer is full or until end of file.

:string-line-in returns three values:

- The index in the buffer at which filling stopped. (If the buffer has a fill pointer, it is set to this value as well.)
- **t** if end of file was reached.
- **t** if the line is not complete; that is, input did not encounter a **Return** character. In that case, there may be more text in the file belonging to the same line.

3.33.4 PROMPT-AND-READ Improvements

There are several new options you can give to **prompt-and-read**, and some existing options now take arguments. Remember that the first argument to **prompt-and-read** is an option that is either a keyword or a list of a keyword followed by arguments (alternating keywords and values). The rest of the arguments are a string and additional args passed to **format** to print the prompt.

Here are the options which have been changed incompatibly:

:eval-form-or-end
: Is changed so that, if the user types just **End**, it returns **:end** as the second value. It used to return **#\end** as the second value in that case. The first value will still be **nil**.

:eval-form-or-end :default *object*

:eval-form :default *object*
: If the user types **Space**, meaning use the default, the second value will now be **:default** rather than **#\Space**. The first value will still be *object*, the default.

Here are the options that now take additional arguments:

:pathname :defaults *default-list* **:version** *default-version*
: A pathname is read, and returned using **fs:merge-pathname-defaults**: *default-list* is passed as the second argument, and *default-version* is passed as the fourth argument.

:number :input-radix *radix* **:or-nil** *nil-ok-flag*
: Reads a string terminated by **Return** or **End**, and parses it into a number using radix *radix* if the number is a rational. The number is returned. If *nil-ok-flag* is non-**nil**, then you may also type just **Return** or **End**, and **nil** is returned.

Here are the new options:

:character Reads a single character and return a fixnum representing it.

:date :never-p *never-ok* **:past-p** *past-required*
: Reads a string terminated by Return or End and parses it as a date/time. The universal time number representing that date/time is returned. If past-required is non-**nil**, the date must be before the present time, or else you get an error and must rub out and use a different date. If never-ok is non-**nil**, then you may also type "never"; **nil** is returned.

:expression
: Is the same as **:read**: read a LISP object using **read** and return it.

:expression-or-end

Reads a LISP object using **read**, but alternately allows just **End** to be typed and returns the two values **nil** and **:end**.

:pathname-or-nil

Reads a file name and returns a pathname object, but if the user types just End then returns **nil** instead. The pathname is defaulted with **fs:merge-pathname-defaults.**

:pathname-or-nil :defaults *default-list* **:version** *default-version*

A pathname is read, and returned using **fs:merge-pathname-defaults**: *default-list* is passed as the second argument, and *default-version* is passed as the fourth argument.

3.33.5 The Rubout Handler

There are some new options for use in controlling the rubout handler; some other options are changed. The new options are **:no-input-save, :activation, :command** and **:preemptable.** The changed options are **:do-not-echo, :pass-through** and **:prompt.**

Recall that the options are the first argument to the **:rubout-handler** stream operation; the remaining arguments being the parsing function and arguments to call it with. The options argument is an alist; each element should look like one of these patterns:

(:no-input-save t)

Does not save this batch of input in the input history when it is done. **yes-or-no-p** specifies this option.

(:full-rubout *value***)**

Causes immediate return from the **:rubout-handler** operation if the buffer ever becomes empty due to deletion of text.

Two values are returned: **nil** and value.

The debugger uses this option so it can erase **"Eval:"** from the screen if you rub out all the characters of the form to be evaluated.

(:initial-input *string***)**

Starts the buffer with *string*.

(:initial-input-pointer *n***)**

Starts by placing cursor *n* chars from the beginning of the buffer. This is used with **:initial-input.**

(:activation *fn args...***)**

Activates if certain characters are typed in. An activation character causes the buffered input to be read immediately, and moves the editing pointer to the end of the buffer.

fn is used to test whether characters are activators. It is called with an input character (never a blip) as the first arg and *args* as additional args. If fn returns non-**nil**, the character is an activator.

The activation character does not go in the buffer itself. However, after the parsing function has read the entire contents of the buffer, it reads a blip **(:activation** *char numeric-arg***)** where *char* is the character that activated and *numeric-arg* is the numeric argument that was pending for the next rubout handler command.

(:do-not-echo *chars...***)**

Poor man's activation characters. Like **:activation** except: the characters that should activate are listed explicitly, and the character itself is returned, rather than a blip, after all the buffered input.

(:command *fn args...***)**

Makes certain characters preemptive commands. A preemptive command returns instantly to the caller, of the **:rubout-handler** operation, regardless of the input in the buffer. It returns two values: a list (**:command** *char numeric-arg*) and the keyword **:command**. Any buffered input remains in the buffer for the next time input is done. In the meantime, the preemptive command character can be processed by the command loop.

In testing for whether a character should be a preemptive command, this works just like **:activation**.

(**:preemptable** *value*)

Makes all blips act as preemptive commands. If this option is specified, the rubout handler returns immediately when it reads a blip, leaving buffered input for next time. Two values are returned: the blip that was read, and value.]

(**:pass-through** (*char doc*) ...)

Defines editing commands to be executed by the parsing function itself. Each *char* is such a command, and *doc* says what it does. *doc* is printed out by the rubout handler's help command. If any of these characters is read by the rubout handler, it is returned immediately to the parsing function regardless of where the input pointer is in the buffer. The parsing function should not regard the character as part of the input.

There are two reasonable things that the parsing function can do:

- print some output
- **:force-kbd-input**

If output is printed, the **:refresh-rubout-handler** operation should be invoked afterward. This causes the rubout handler to redisplay so that the input being edited appears after the output that was done. If input is forced, it will be interpreted as rubout handler commands.

There is no way to act directly on the buffered input because different implementations of the rubout handler store it in different ways.

(**:prompt** *fn-or-string*)

Directs prompting for the input being read. If *fn-or-string* is a string, it is printed; otherwise it is called with two args: the stream, and a character that is an editing command that says why the prompt is being printed.

(**:reprompt** *fn-or-string*)

Same as **:prompt** except used only if the input is reprinted for some reason after editing has begun. The **:reprompt** option is not used on initial entry. If both **:prompt** and **:reprompt** are specified, **:prompt** is used on initial entry and **:reprompt** thereafter.

A new convenient way to invoke the rubout handler on a stream if the stream supports it is to use **with-input-editing**.

with-input-editing (*stream options*) *body...* Macro

Invokes the rubout handler on stream, if it is supported, and then executes body. body is executed in any case, within the rubout handler if possible. body's values are returned by **with-input-editing**. However, if a preemptive command is read, **with-input-editing** returns immediately with the values being as specified above under **:command** or **:preemptable**.

options are used as the rubout handler options.

sys:parse-error Condition
sys:read-error (**parse-error**) Condition

All rubout handlers now check for the condition name **sys:parse-error** when they decide whether to handle an error. They used to check for **sys:read-error**. All the errors signaled by the system that have the condition name **sys:read-error** now have **sys:parse-error** as well, so no change in behavior should be apparent. However, you can signal an error that has **sys:parse-error** but not **sys:read-error** if you wish (say, if the error happens in some function other than **read**).

sys:parse-error is also the condition name that the compiler looks for in its efforts to continue from errors that happen while reading text to be compiled.

sys:parse-ferror *format-string* &rest *format-args* Function

The function **sys:parse-ferror** is a convenient way to signal such an error, if you do not want any additional condition names besides **sys:parse-error** and the ones it implies. If **sys:parse-ferror** is called while reading text to be compiled, it will return **nil** automatically.

3.34 Readtables

Because of the adoption of COMMON LISP, some of the Lisp reader syntax internals have been changed or extended. In addition, a mechanism has been added for named readtables, which may be helpful in more easily supporting languages with different syntaxes from ZETALISP

3.34.1 Syntax Descriptions

Remember, that even though the following changes have been documented as a result of COMMON LISP, the syntax descriptions and the way there are modified are not accessed exactly this way in COMMON LISP itself.

si:set-syntax-from-description *char description* &optional *readtable* Function

There are new syntax descriptions that you can pass to this function:

si:escape A quote-one-character character. In the ZETALISP readtable / is such a character. In the COMMON LISP readtable, \ has this syntax description.

si:multiple-escape
A quote-several-characters character. In the ZETALISP readtable | is such a character.

si:character-code-escape
Is followed by a character's octal code. In the ZETALISP readtable ⊗ is such a character.

si:digitscale
A character for shifting an integer by digits. In the ZETALISP readtable ^ is such a character.

si:bitscale A character for shifting an integer by bits. In the ZETALISP readtable _ is such a character.

si:non-terminating-macro
A macro character that is not recognized if a token is already in progress. In the ZETALISP readtable # is such a character. (It is also a dispatching macro, but that is another matter.) The correct way to make a character be a macro is with **set-macro-character**, not with this description.

The syntax descriptions **si:slash** and **si:circlecross** are still implemented but it is preferable to use **si:escape** or **si:character-code-escape**. The syntax **si:verticalbar** is no longer defined; use **si:multiple-escape**. Unfortunately, it is no longer possible to define **si:doublequote**, since Doublequote (") is now just a macro character.

3.34.2 Named Readtables

To aid in the support for COMMON LISP in Zetalisp, readtables were given names so that they could be referred to symbolically. The **readtable** and **syntax** file attributes use this feature to distinguish COMMON LISP files from Zetalisp files. Named readtables may be useful for similar applications. (Note that for a readtable to be accessible from the file attribute list, one of its names must readable as a symbol – so it should have one short name with no whitespace in it.)

There are two ways to get a named readtable:

- The first way is to use the readtable compiler, to make a readtable from scratch. In the section of the readtable definition file where the options (**:opts**) go, use the **:names** option.

```
(:OPT :NAMES '("Lisp Machine COBOL" "COBOL"))
```

- The second way is to copy another readtable, and give it some names. You can actually override another readtable's name by pushing your readtable in front of it on **si:*all-readtables***, so be careful about this feature, which may or may not always be the right thing for an application.

```
(defvar *strange-table* :unbound
  "For slightly modified Common Lisp syntax")

(defun set-up-strangeness ()
  (let ((rt (copy-readtable nil)))
    (setf (si:rdtbl-names rt) '("strange Common Lisp" "STRANGE"))
    (push rt si:*all-readtables*)
    (setq *strange-table* rt)))
```

The COMMON LISP readtable has, among other names, **CL** and **Common-Lisp** for nicknames. The standard Zetalisp readtable can be found with the names **T**, **Traditional**, **ZL**, and **Zetalisp**.

si:find-readtable-named *name create-p* Function

Find or possibly create a readtable named *name* If there is a readtable which has a name **string-equal** to it, we return that readtable. Otherwise, we may create such a readtable, depending on *create-p*

nil	
:error	Get an error.
:find	Return **nil**
:ask	Ask whether to create a readtable named *name* which is a copy of the current readtable (***readtable***), and returns it if so.
t	Create the readtable (a copy of ***readtable***) and return it.

si:*all-readtables* Variable

This is a list of all readtables *except* those created with **copy-readtable**, which does not automatically put new readtables on this list.

si:rdtbl-names *readtable* Function

The accessor for the names (strings) of the readtable, the first name being the one printed out at the beginning of Lisp interaction loops. The rather constrained name of the function is due to historical reasons.

3.35 Fasdumping Functions Record Package

These functions:

- **dump-forms-to-file**
- **compiler:fasd-symbol-value**
- **compiler:fasd-font**
- and **compiler:fasd-file-symbols-properties**

now always record, in the QFASL file created, the name of the package in which the file was written. This makes sure that the symbols used when the file is loaded will be the same as when it was dumped.

In **dump-forms-to-file**, you can specify the package to use by including a **:package** attribute in the **attribute-list** argument. For example, if that argument is the list **(:package :si)** then the file is dumped and loaded in the **si** package. If you do not specify a package, the file is dumped and loaded in **user**. With the other three functions, the file is always dumped and loaded in **user**.

3.36 Process Queues

A process queue is a kind of lock, that can record several processes that are waiting for the lock and grant them the lock in the order that they requested it. The queue has a fixed size. If the number of processes waiting remains less than that size, then they will all get the lock in the order of requests. If too many processes are waiting, then the order of requesting is not remembered for the extra ones.

si:make-process-queue *name size* Function
Makes a process queue object named name, able to record size processes. size includes the process that owns the lock.

si:process-enqueue *process-queue* &optional *lock-value who-state* Function
Attempts to lock *process-queue* on behalf of *lock-value*. If *lock-value* is **nil** then the locking is done on behalf of **current-process**.

If the queue is locked, then *lock-value* or the current process is put on the queue. Then this function waits for that lock value to reach the front of the queue. When it does so, the lock has been granted, and this function returns.

who-state appears in the who line during the wait. It defaults to **"Lock"**.

si:process-deqeueue *process-queue* &optional *lock-value* Function
Unlocks process-queue. *lock-value* (that defaults to the current process) must be the value that now owns the lock on the queue, or an error occurs. The next process or other object on the queue is granted the lock and its call to **si:process-enqueue** will therefore return.

si:reset-process-queue *process-queue* Function
Unlocks the queue and clears out the list of things waiting to lock it.

si:process-queue-locker *process-queue* Function
Returns the object in whose name the queue is currently locked, or **nil** if it is not now locked.

3.37 New Function SI:PATCH-LOADED-P

compat

si:patch-loaded-p *major-version minor-version* &optional (*system-name* **"SYSTEM"**) Function

Returns **t** if the changes in patch number *major-version.minor-version* of system *system-name* are loaded. If major-version is the major version of the system currently loaded, then the changes in that patch are loaded if the current minor version is greater than or equal to *minor-version*. If the currently loaded major version is greater than major-version, then it is assumed that the newer system version contains all the improvements patched into earlier versions, so the value is **t**.

3.38 Date Formats

time:*default-date-print-mode* Variable

This defines the default way to print the date for functions in the **time** package that accept a print-mode argument, which currently include:

* **time:print-time**
* **time:print-universal-time**
* **time:print-brief-universal-time**
* **time:print-date**
* **time:print-universal-date**
* **time:print-current-time**
* **time:print-current-date**

Following is a description of the possible values, using ZETALISP syntax.

:dd//mm//yy
Prints out as 27/10{/66}

:dd//mm//yyyy
27/10{/1966}

:mm//dd//yy
10/27{/66}

:mm//dd//yyyy
10/27{/1966}

:dd-mm-yy 27-10{-66}

:dd-mm-yyyy
27-10{-1966}

:dd-mmm-yy
27-Oct{-66}

:dd-mmm-yyyy
27-Oct{-1966}

:dd/ mmm/ yy
27 Oct{ 66} – Note that the print name of this symbol really does contain a space; backslash would be used to enter the symbol in COMMON LISP syntax.

:dd/ mmm/ yyyy
27 Oct{ 1966}

:ddmmmyy
27Oct{66}

:ddmmmyyyy
27Oct{1966}

:yymmdd 661027

:yyyymmdd
19661027

:yymmmdd
{66}Oct27

:yyyymmmdd
{1966}Oct27

:yy-mmm-dd
{66-}Oct-27

:yyyy-mmm-dd
{1966-}Oct-27

:yy-mm-dd {66-}10-27

:yyyy-mm-dd
{1966-}10-27

These last four, and all the **yyyy** ones are new since the manual.

The default value is **:mm//dd//yy**. If one wishes to customize this for a site (usually, a site not in the United States), simply put a **setq** of **time:*default-date-print-mode*** to the appropriate value something in the **SYS: SITE; SITE LISP** file.

The time parser now accepts ISO format dates. **1980-3-15** means 15 March, 1980; **1980-MAR-15** means 15 March, 1980.

3.39 Network Changes

Some of the site changes (section 9.3.2.2, page 85) are also network-related.

3.39.1 Host Network Operations

si:parse-host *string error-p* (*unknown-ok t*) Function

si:parse-host's third argument, *unknown-ok*, now defaults to **t**. That means ~~that~~ if it can't find the host on **si:host-alist**, it tries contacting a host table server to see if *it* knows about the host. If the server contact does not, an error is signalled (or **nil** is returned) as usual. The change was made to minimise the penalty for not loading the latest site files. (Maintaining up-to-date site information can be a problem at large installations.)

The list of hosts that may be contacted on the Chaosnet for this service are listed in the site option **:chaos-host-table-server-hosts**.

:network-addresses Operation on **si:host**

The operation **:network-addresses**, on a host object, returns an alternating list of network names and lists of addresses, such as

```
(:chaos (3104) :arpa (106357002))
```

You can therefore find out all networks a host is known to be on, using **getf**.

:network-address *network* &optional *smart-p* Operation on **si:host**

Returns a network address, if possible, for the host on *network*. The network address returned is the primary one (determined ultimately by the order found in the host table source) unless *smart-p* is non-**nil**; then, some optimal address as defined by *network* is returned.

The actual format of the network address is left unspecified; it is usually the "unparsed" form which is passed to the network entry point functions.

:unparsed-network-address *network* &optional *smart-p* Operation on **si:host**

Like **:network-address**, but returns an unparsed network address (a string), where the string representation is defined by the network.

:internet-connect *socket protocol* &key *timeout* (*ascii-translation* **t**) (*direction* **:bidirectional**) Operation on **si:host**

This is the current interface for using Internet in LMI ZETALISP, to connect to the host at *socket* using *protocol*, a keyword. Currently, the only legal value is **:tcp**. *timeout*, in sixtieths of a second, currently defaults to some reasonable value. The remaining keyword arguments are only applicable when the Internet protocol requested is **:tcp**. *direction* can be one of symbols acceptable to **chaos:open-stream**: **:input**, **:output**, or **:bidirectional**. Currently, *ascii-translation* defaults to **t**, since most TCP servers and protocols are oriented to the ASCII character set.

3.39.2 New Error Condition SYS:NO-SERVER-UP

sys:no-server-up (sys:connection-error) Condition

The error condition **sys:no-server-up** is signalled by certain requests for a service from any available network host, when no suitable host is currently available.

3.39.3 Some Chaosnet Functions Renamed

Some functions in the **chaos** package have had their names changed. This is so we can avoid having two advertised system functions with the same name in different packages. The old names still work.

Old Name	New Name
chaos:finish	chaos:finish-conn
chaos:close	chaos:close-conn
chaos:finished-p	chaos:conn-finished-p

3.39.4 Chaosnet Listening Streams

Now you can listen for a Chaosnet connection and open a stream at the same time. To do this, call **chaos:open-stream** with **nil** as the host argument. You must still pass a non-**nil** contact-name argument. The function will return a stream to you as soon as someone attempts to connect to that contact name.

At this time, you must accept or reject the connection by invoking the stream operation **:accept** or **:reject**. **:reject** takes one argument, a string to send back as the reason for rejection. Before you decide that to do, you can use the **:foreign-host** operation to find out where the connection came from.

3.39.5 New Chaos Routing Inspector Functions

These two functions make use of the **DUMP-ROUTING-TABLE** protocol, documented in the new edition of the *Lisp Machine Manual.* They are primarily for inspecting the operation of the network and the localisation of bridging and routing problems.

chaos:show-routing-table *host* &optional (*stream* ***standard-output***) Function

Prints out the routing table of *host* onto *stream.*

chaos:show-routing-path &key (*from* **si:local-host**) *to* (*stream* ***standard-output***) Function

Shows how packets will flow from *from* to *to*, using the routing information supplied by *from* and any intervening bridges to figure out the path.

For example, **(chaos:show-routing-path :from "charon" :to "nu-1")** may produce the following output:

```
MIT-CHARON will bounce the packet off MIT-SIPB-11 at cost 81.
MIT-SIPB-11 will bounce the packet off MIT-INFINITE at cost 63.
MIT-INFINITE will bounce the packet off MIT-BYPASS at cost 51.
MIT-BYPASS will bounce the packet off MIT-OZ-11 at cost 37.
MIT-OZ-11 will bounce the packet off XI (XX-Network-11) at cost 23.
Direct path from XI (XX-Network-11) to host MIT-NU-1 on subnet 32 at
interface 1.
```

3.40 Infix Expressions.

You can now include infix expressions in your Lisp code. For example,

```
#◇X:Y+CAR(A1[I,J])◇
```

The ◇ character is **Altmode**.

is equivalent to

```
(setq x (+ y (car (aref a1 i j))))
```

#◇ begins an infix expression, and ◇ ends it.

The atomic terms of infix expressions include

- symbols: use " to quote special characters.
- numbers: any valid Lisp real or imaginary number is accepted. Complex numbers can be constructed by addition or subtraction.
- strings: the same as in ordinary Lisp syntax.
- raw Lisp data: ! followed by any Lisp expression, as in

```
#◇ FOO . !(CAR BAR) ◇ => (list* foo (car bar))
```

Combining operations:

Highest precedence

a[*i*]	(AREF a *i*)	
a[*i,j*]	(AREF a *i j*)	*and so on*

```
  examples
    X[I,J+3]    =>    (AREF X (+ J 3))
    (GET-MY-ARRAY(FOO))[I]    =>    (AREF (GET-MY-ARRAY FOO) I)

    f(a)           (f a)
    f(a,b)         (f a b)      and so on
  examples
    CAR(X)    =>    (CAR X)

    (exp)        exp    parentheses control order of evaluation
  examples
    (X+1)*Y    =>    (* (+ X 1) Y)

    (e1, e2)          (PROGN e1 e2)      and so on
  examples
    (X:5, X*X)    =>    (PROGN (SETQ X 5) (* X X))

    [elt]          (LIST elt)
    [e1,e2]        (LIST e1 e2)     and so on
  examples
    [!'X,Y,Z]    =>    (LIST 'X Y Z)

Precedence 180 on left, 20 on right
    a : b               (SETF a b)
  examples
    X: 1 + Y: Z+5    =>    (SETQ X (+ 1 (SETQ Y (+ Z 5))))

Precedence 140
    a^b               (EXPT a b)    right associative
  examples
    X ^ N ^ 2    =>    (EXPT X (EXPT N 2))

Precedence 120
    a* *b               (* a b)
    a* *b * *c        (* a b c)      and so on
    a/ *b               (// a b)
    a/ *b / *c        (// a b c)     and so on

Precedence 100
    - a                 (- a)
    a+ *b               (+ a b)
    a+ *b + *c        (+ a b c)  and so on
    a- *b               (- a b)
    a- *b - c         (- a b c)      and so on

Precedence 95
    a. b                (LIST* a b)
    a. b . c          (LIST* a b c)     and so on
    a@ b                (APPEND a b)
    a@ b  c           (APPEND a b c)    and so on
```

```
Precedence 80
    a ε b                    (MEMQ a b)
    a= b                     (= a b)
    a= b = c            (= a b c)         and so on
    <, >, ≠, ≥, ≤ are like =.

Precedence 70
    NOT a                    (NOT a)

Precedence 60
    a AND b                  (AND a b)
    a AND b AND c            (AND a b c)    and so on

Precedence 50
    a OR b                       (OR a b)
    a OR b OR c                  (OR a b c)   and so on

Precedence 45 for c, 25 for a and b.
    IF c THEN a                  (IF c a)
    IF c THEN a ELSE b           (IF c a b)
```

It is easy to define new operators. See **SYS: IO1; INFIX LISP**.

3.41 Bug Reports for User Systems

To make it easier to collect bug reports about a system, there is now a **:bug-reports** option to **defsystem**. Two values are supplied: the name of the topic, and a documentation string. The topic name is usually the name of the system. The documentation string appears in the mouse documentation line when the user sends a bug report from ZMail. For example:

```
(defsystem foo
  ...
  (:bug-reports "FOO" "Tell about a bug in the FOO system")
  ...
  )
```

For this to really work, there must be a mailing address named **bug-foo** on the bug report host (that is, the host named by the site option **:host-for-bug-reports**).

This feature does not work with the **Control-M** debugger command, because the error handler presets the address according to the value of the string returned by sending the **:bug-report-recipient-system** message to the error instance.

4. DEFSTRUCT

This describes changes to the **defstruct** feature as implemented in Release 2.

The compatible changes to **defstruct** as discussed in this section of the manual are:

- New Options
- Documentation for Structures
- Slot Options
- Changes to the **:include** option
- **defstruct** Tries to Determine an Appropriate Array Type
- New Predefined Structure Types
- COMMON LISP Support

One change is that **defstruct** no longer generates any sort of **eval-when**. If you want the expansion of a **defstruct** to be inside an **eval-when**, simply write an **eval-when** around the **defstruct**.

4.1 New Options

The following are now accepted by **defstruct** in addition to the options described in the *Lisp Machine Manual.*

:callable-constructors

Giving this option a value of **t** (i.e. by writing (**:callable-constructors t**)) causes constructors for this structure to be functions, rather than macros, as they used to be. This, however, means that code like the following, which works with a macro-defined constructor, will usually cause an error if it is a function:

```
(make-foo a 1 b 'bee)
```

The syntax to use for callable constructors is like that for **&key** functions (which is actually how they are defined):

```
(make-foo :a 1 :b 'bee).
```

Macro-defined constructors now accept keywords for slot-names also. Just to facilitate changing the kind of constructor you use, it is probably best to always use this syntax. However, an irresolvable incompatibility exists in the way the two types of constructors handle the constructor options such as *:times* and **:make-array**. When **:callable-constructors** is **nil**, they *should not* be quoted, and when it is **t**, they *must* be quoted. For example, in the first case we would say:

```
(make-frobboz :slot-1 'foo :make-array (:leader-length 2))
```

With callable constructors the **:make-array** argument must be quoted:

```
(make-frobboz :slot-1 'foo :make-array '(:leader-length 2))
```

:subtype This option is valid only when used with structure-types that include **:subtype** among their **:defstruct-keyword** keywords (see below). Such types include things like **:array** and **:array-leader**, for which a subtype of the primary array-type is a meaningful concept. In the case of arrays, this could be used to make a structure of this type use a specific array-type, rather than the default **art-q**. The subtype can also be implicitly specified through the **:type** option. Types such as **:list** or **:fixnum-array** do not have any any meaningful subtypes, and hence do not support the **:subtype** option. It is an error to use **:subtype** with such types.

:type This is by no means a new option, but its syntax has been extended. Previously, this option could be used only in the form (**:type** *defstruct-type*). It is now possible to write (**:type** (*defstruct-type subtype*)), the effect being like specifying both (**:type** *defstruct-type*) and (**:subtype** *subtype*). For example:

```
(defstruct (foo (:type (:array ART-4B))) A B)
  or
(defstruct (foo (:type (:vector (mod 16)))) a b)
```

using a COMMON LISP type defines a structure with two slots, each of which can contain only fixnums in the range [0,15]. This is a COMMON LISP change, but is worthwhile to use in any case as this syntax is more transparent and cleaner than the present technique of writing:

```
(defstruct (foo (:type :array) (:make-array (:type art-4b))) a b)
```

:print-function

The argument to this option is a function of three arguments, which will print an object of the type being defined. This function will be called with three arguments – the structure to be printed, the stream to print it on, and the current printing depth (which should be compared with ***print-level***). The function is expected to observe the values of the various printer-control variables. Example:

```
(defstruct (bar :named
  (:print-function
    (lambda (struct stream depth)
      (format stream "#<This is a BAR, with ring-ding index ~S>"
      (zap struct)))))
    "The famous bar structure with no known use."
    (zap 'yow) random-slot)

(MAKE-BAR) => #<This is a BAR, with ring-ding index YOW>
```

This option is similar in application to the existing option **:print**. Its introduction is a COMMON LISP change.

4.2 Documentation for Structures

defstruct now interprets a string occurring after the structure name and options as documentation for this structure. The documentation can be accessed by:

(documentation *structure-name* **'structure)**

and changed by **setf**ing such a form.

4.3 Slot Options

Slots within a structure may now include one or more slot options. The extended syntax for defining slots is either:

```
slot-name
```

or

```
(slot-name (default-init
              (slot-option-1 option-value-1
               slot-option-2 option-value-2 ...)))
```

or

```
((slot-name-1 byte-spec-1 (default-init-1
                            (slot-option-1-1 option-value-1-1 ...)))
 (slot-name-2 byte-spec-2 (default-init-1
                            (slot-option-2-1 option-value-2-1 ...)))
 ...)
```

Here are the currently defined *slot-options*:

:read-only *flag*
: Specifies that this slot mat not be **setf**ed if *flag* is non-**nil**. The contents of this slot are not supposed to be changed after you construct the structure.

:type *type*
: Declares that this slot is expected to be of a given type. The LISP machine compiler does not use this for any assumptions, but sometimes the information enables **defstruct** to deduce that it can pack the structure into less space by using a numeric array type.

:documentation *documentation-string*
: Makes *documentation-string* the documentation for the slot's accessor function. It also goes in the **defstruct-slot-description-documentation** for this slot. Example:

```
(defstruct (eggsample :named :conc-name
              (:print-function #'(lambda (s stream ignore)
       (format stream "#<Eggsample ~S ~S ~s>"
(eggsample-yolk s)
(eggsample-grade s)
(eggsample-albumen s)))))
     (yolk 'a :type symbol :documentation "First thing you need in
     an eggsample.")
     (grade 3 :type (mod 4))
     (albumen nil :read-only t))
=> eggsample
(setq egg (make-eggsample :albumen 'white))
=> #<Eggsample A 3 WHITE>
(setf (eggsample-yolk <c-sh-d>
   EGGSAMPLE-YOLK: (EGGSAMPLE)
   "First things you need in an eggsample."
(setf (eggsample-yolk egg) 19.5)
=> 19.5      ; no type checking !
egg
=> #<Eggsample 19.5 3 WHITE>
(setf (eggsample-albumen egg) 'eggsistential)
=> >>ERROR: SETF is explicitly forbidden on
```

```
(EGGSAMPLE-ALBUMEN EGG)
While in the function SI::UNSETFABLE ← SI::LOCF-APPLY
   ← SI::SETF-1
...
```

4.4 Changes to the :INCLUDE Option

4.4.1 DEFSTRUCT

defstruct now accepts slot-options in the specification for included slots. This extended syntax is illustrated here:

```
(defstruct one :conc-name :named
  (slot-1 0 :type fixnum :documentation "The very first" :read-only t)
  (slot-2 'bar)
  slot-3)

(defstruct (two :conc-name :named
           (:include one (slot-1 6 :documentation "The second first")
                  (slot-3 '(a b) :type cons :read-only t)))
   (slot-3 5))
```

two will be a structure whose first slot has default value 6, has the documenation, and is read-only and of type **fixnum**, these last two attributes being inherited from the included structure. The third slot will have a default value of **'(a b)**, should be a cons, and is read-only.
The following example will cause an error:

```
(defstruct (loser :named :conc-name
                  (:include one (slot-1 0 :read-only nil :type symbol))))
```

This is because (i) the slot is specified to be not read-only, when the included slot was, and (ii) the slot was given a type that is not a subtype of the included slot type.

4.4.2 New Slot-Accessor Functions Generated

Previously no accessor called **two-slot-1** was generated in the example above, and you had to access that slot using the function **one-slot-1**. Now such accessors are generated for all the included slots, using the conc-name of the including structure. Note that the accessors need not necessarily be the same as the accessors used in the included structure. That is, they may have different documentation, or be read-only.

4.5 DEFSTRUCT Tries to Determine an Appropriate Array Type

If all the slots to **defstruct** are given **:type** slot-options and the structure is based on an array that can be of a specialised type (such as **:array**, **:typed-array**, **:grouped-array** or **:vector**) and no **:subtype** is explicitly given, then **defstruct** will attempt to find the most storage-efficient array-type (subtype) for the structure. Example:

```
(defstruct (foo)
    (eh 3 :type (mod 7))
    (be 0 :type (mod 1)))
```

will define a structure that makes arrays of type **art-4b**. This feature can be overridden by explicitly giving a **:subtype**, or by just not giving all the slot-types.

4.6 New Predefined Structure Types

The system now has a number of new predefined structure types:

:typed-array
: This is the same as **:array**, for use with **:named-typed-array.**

:named-typed-array
: This is an named array type with which you can specify a subtype restricting the type of elements. The named structure symbol is always put in leader slot 1.

:named-fixnum-array
: Named **:fixnum-array**; the named-structure-symbol is stored in the leader.

:named-flonum-array
: Named **:flonum-array**; the named-structure-symbol is stored in the leader.

:vector
: Same as **:typed-array**. This is used for COMMON LISP.

:named-vector
: Same as :named-typed-array. This is the default for COMMON LISP structures.

:phony-named-vector
: This is what you get in COMMON LISP if you say **(:type :vector)** and **:named**.

Examples:

```
(defstruct (foo (:type (:vector (mod 4)))) a)
(defstruct (foo (:type (:vector art-fat-string))) a)
(defstruct (bar (:type :fixnum-array) :named) x y z)
```

4.7 Common Lisp Support

There now exists a macro **cli:defstruct** to support the COMMON LISP **defstruct** feature. The only difference between **cli:defstruct** and regular **defstruct** is that the COMMON LISP version has different defaults for certain options:

:conc-name
: Defaults to **name-**, where *name* is the defstruct being defined. (Normally, it is **nil** by default.)

:predicate
: Defaults to **t**, producing a predicate called *name*-**p**, if no predicate name is requested by the user. (Default is normally **nil**.)

:callable-constructors
: Defaults to **t** (normally **nil**).

:alterant
: Defaults to **nil**, i.e. no alterant macro is defined (traditionally a macro called **alter-***name* is defined).

If you do not specify **:type**, you get **:named-vector**, which makes a named structure. You get a predicate by default. You may specify how to print the structure.

If you do specify **:type**, you never get a named structure. You either get a plain list or a plain vector. You do not get a predicate by default, and you may not request one. You may not specify how to print.

If you specify **:named** along with **:type**, you do not get a named structure. You get either type **:named-list** or type **:phony-named-vector**. Both of these types store the structure type in the structure somewhere, and both of them allow you to define a predicate that looks there to see whether an object appears to be a structure of the sort you defined. Neither type is recognizable by **typep**, and anyone randomly creating a list or vector with the right thing in it at the right place will find that it satisfies the predicate.

4.8 Changes to DEFSTRUCT-DEFINE-TYPE Options

4.8.1 New Per-Type Method of Declaring DEFSTRUCT Options

defstruct used to check whether a keyword appearing as an option was valid by checking whether the keyword had a non-**nil** **si:defstruct-description** property. The problem with this technique is that keywords that are appropriate to only one type of structure are accepted by **defstruct** as options for other structures for which they are meaningless. (For example, the **:times** option for grouped arrays has no meaning for other currently-defined structure types.) The new way to achieve this functionality is via the **:defstruct-keywords** option to **defstruct-define-type**, which has the same syntax as the old **:keywords** option, for example, (**:defstruct-keywords** *keyword-1 keyword-2* ...). A typical use is the following, which is the actual definition of the **:grouped-array** type:

```
(defstruct-define-type :grouped-array
  (:cons-keywords :make-array :times :subtype)
  (:defstruct-keywords :make-array :times :subtype)
  (:defstruct (description)
    (defstruct-hack-array-supertype description))
  (:cons (arg description etc) :alist
    (lispm-array-for-defstruct
      arg
      #'(lambda (v a i) '(aset ,v ,a ,i))
      description etc nil nil nil
      (or (cdr (or (assq :times etc)
                   (assq :times
                         (defstruct-description-property-alist))))
     1)
      nil))
  (:ref (n description index arg)
    description ; ignored
    (cond ((numberp index)
      '(aref ,arg ,(+ n index)))
          ((zerop n)
      '(aref ,arg ,index))
  (t '(aref ,arg (+ ,n ,index))))))
```

The **:cons-keywords** specifies the valid keywords that can be supplied to a constructor for this type. **:defstruct-keywords** (which happens to be the same in this case) specifies valid keywords to appear in the structure definition of a grouped array, making

```
(defstruct (foo (:type :grouped-array) :times 7) a b)
```

a valid defstruct, while

```
(defstruct (foo (:type :grouped-array) :typo 7) a b)
```

and

```
(defstruct (foo (:type :array) :times 7) a b)
```

signal an error.

The old type-independent method of saying

```
(defprop :make-array t :defstruct-option)
```

is obsolete, although still supported so that programs using this continue to work.

4.8.2 :KEYWORDS Option to Renamed :CONS-KEYWORDS

This has been done because **defstruct-define-type** now knows about more than one type of keyword relevant to the structure, namely **:cons-keywords** and **:defstruct-keywords**, which are relevant to the construction and definition respectively of structures of a given type. Previously, there were no **:defstruct-keywords**, and so there was no ambiguity in calling this option plain **:keyword**. As this is largely a change for consistency's sake, the old syntax continues to be supported.

5. The New Package System

A new package system has been created; it is essentially that of COMMON LISP with some added compatibility features. Its highlights are:

- Symbols in a package are now marked as either internal or external. Only the external symbols are inherited by other packages.
- Packages are no longer arranged in a hierarchy; inheritance is no longer required to be transitive. Now you can specify exactly which other packages' external symbols are to be inherited by a new package.
- **keyword** and **user** are now two distinct packages. No symbol is shared between **keyword** and **global**, so that **compile** and **:compile** are two distinct symbols, and so are **nil** and **:nil**. You must now be careful to use the correct symbol (keyword or global) in your code, whereas it used to make no difference.
- All package names are now global in scope; they mean the same thing regardless of which package is current. It is also possible to define local nicknames, in effect in only one package, but this is usually not done.
- Package prefixes can now contain **#**: in place of just :. They also sometimes contain two colons in a row.

These things have not changed in the new package system:

- A package is still an object used by **intern** to map names into symbols. At any time one package is current; it is the value of ***package***, and is used by default in **intern** and **read**. Packages can still have their own ("local") symbols while inheriting additional symbols from other packages.
- **read** still looks up symbols in the current package by default. It still allows you to specify another package with a package prefix, a package name followed by a colon, as in **si:full-gc**.
- There is still a package called **global**, which contains the fundamental function and variable symbols of LISP, such as **eval**, **cond**, **setq**, **t** and **package**. By default, new packages inherit from this package alone.
- There is still a keyword package whose symbols are normally referred to with a package prefix that is just a colon, as in **:noselective**.
- Nearly all the old documented functions for operating on packages still work, though not always exactly in the same way.

5.1 Specific Incompatibilities

Here are the specific incompatibilities between the old and new package systems.

- **list** and **:list** are now two distinct symbols. No symbol is now shared between the **global** package and the **keyword** package. This means that in many cases where a colon prefix used to make no difference, it is now significant. You must be careful to use package prefixes when you want the keyword symbol. The documentation has made the distinction even when it did not matter. If you are lucky, you followed the documentation as if you did not realize that **list** and **:list** were the same symbol, and your old code will still work.
- Files loaded into the **user** package will not work if they omit the colon on keyword symbols, as they were formerly allowed to do. See the section "The USER Package", below, for more information. With luck, these problems will be infrequent.

- **pkg-subpackages** no longer exists. There is no way to simulate the old meaning of this function, since there is no equivalent of "subpackages" close enough to the old concept.
- **pkg-super-package** does still exist, but it uses a heuristic. Its new definition manages to satisfy most aspects of this function's old contract, but not quite all. If you define a package with **package-declare**, **pkg-super-package** will still return the same package that it used to return. But for packages defined in other, newly available ways, there may be no unique way of defining the "superpackage". The **global** package will probably be returned as the "superpackage" in this case.
- **pkg-refname-alist** still exists and its value is used in roughly the same way. However, it is no longer the case that most package names are found there. In fact, these lists will normally be **nil**.
- Some hairy undocumented features of **package-declare** are no longer supported.
- **apropos**, **who-calls** and **what-files-call** take different keyword arguments. They used to accept keywords **:superiors** and **:inferiors** to specify whether to look in the superpackage and subpackages of the specified package. Now that packages do not have superpackages and subpackages, the keywords have been changed to **:inherited** and **:inheritors**.
- Package names are now treated much like symbol names with regard to case. In package prefixes, letters are converted to upper case unless quoted with a slash or vertical bar, so it does not matter what case you use. In functions that accept a package name to look up a package, the string or symbol you specify is compared, with case being significant. Thus, if you use a string, the string must contain upper-case letters if the package name does. If you supply a symbol, you can type the symbol in upper or lower-case because **read** converts the characters of the symbol to upper case anyway.

5.2 The Current Package

package Variable
package Variable

These are now synonymous names for a variable whose value is the current package. ***package*** is the COMMON LISP name for **package**.

packagep *anything* Function

Returns **t** if *anything* is a package.

pkg-bind (*package*) *body...* Macro

Executes body with ***package*** bound to *package*.

pkg-goto *package* Function

Sets ***package*** to *package*, but only if *package* is suitable. A package that automatically exports new symbols is not suitable and causes an error without setting ***package**. This is because typing expressions with such a package current would create new external symbols and interfere with other packages that **use** this one.

pkg-goto-globally *package* Function

Sets the global binding of ***package*** (in effect in all processes that do not bind ***package***) to *package*. An error occurs if *package* automatically exports new symbols. Note that the LISP read-eval-print loop binds ***package***, so such loops are not affected by the global binding. Conversely, doing **pkg-goto** inside a LISP read-eval-print loop would not change the global binding. **load** also binds the current package, so in order to change the global binding from your init file, you must use this function.

5.3 Finding All Packages

all-packages Variable

all-packages is a new variable whose value is a list of all packages.

list-all-packages Function

The function **list-all-packages**, with no arguments, returns the same list. This is a standard COMMON LISP construct. Strangely, ***all-packages*** is not.

5.3.1 Package Naming

A package has one name, also called the primary name, and can have, in addition, any number of nicknames. All of these names are defined globally, and all must be unique. An attempt to define a package with a name or nickname that is already in use is an error.

Either the primary name of a package or one of its nicknames counts as a name for the package. All of the functions described below that accept a package as an argument will also accept a name for a package (either a string or a symbol whose pname is used). Arguments that are lists of packages may also contain names among the elements. However, for transportable COMMON LISP, one must not use this feature.

When the package object is printed, its primary name is used. The name is also used by default when printing package prefixes of symbols. However, when you create the package you can specify that one of the nicknames should be used instead for this purpose. The name to be used for this is called the prefix name.

Case is significant in package name lookup. Usually package names should be all upper case. **read** converts package prefixes to upper case except for quoted characters, just as it does to symbol names, so the package prefix will match the package name no matter what case you type it in, as long as the actual name is upper case: **TV:FOO** and **tv:foo** refer to the same symbol. In the functions **find-package** and **pkg-find-package**, and others that accept package names in place of packages, if you specify the name as a string you must give it in the correct case:

(find-package "TV") finds the **tv** package
(find-package "tv") finds nothing

You can alternatively specify the name as a symbol; then the symbol's pname is used. Since **read** converts the symbol's name to upper case, you can type the symbol in either upper or lower case: **(find-package 'TV)** and **(find-package 'tv)** both find the **tv** package since both use the symbol whose pname is **"TV"**.

Each package has a list of local nicknames, which are mapped into packages. These local nicknames serve as additional names for those other packages, but only when this package is current, and only for the sake of package prefixes in **read**. It is permissible to define a local nickname that is the same as the name of some existing package; this is useful for "redirecting" symbol references with package prefixes to packages other than the ones named in the code.

Relevant functions:

package-name *package* Function

Returns the name of *package* (as a string).

package-nicknames *package* Function

Returns the list of nicknames (strings) of *package*.

package-prefix-print-name *package* Function

(Not in COMMON LISP) Returns the name to be used for printing package prefixes that refer to *package*. Note that COMMON LISP does not have such a feature.

rename-package *package new-name* &optional *new-nicknames* Function

Makes *new-name* be the name for *package*, and makes *new-nicknames* (a list of strings, possibly **nil**) be its nicknames. An error is signalled if the name or any of the nicknames is already in use.

find-package *name* &optional *use-local-names-package* Function

Returns the package that *name* is a name for, or **nil** if there is none. If *use-local-names-package* is non-**nil**, the local nicknames of that package are checked first. Otherwise only actual names and nicknames are accepted. *use-local-names-package* should be supplied only when interpreting package prefixes. The use of the second argument is not transportable COMMON LISP.

If a package is supplied as *name*, it is returned.

If a list is supplied as *name*, it is interpreted as a specification of a package name and how to create it. The list should look like (*name super-or-use size*). If *name* names a package, it is returned. Otherwise a package with name *name* is created with **make-package** (see page 60) and then returned. *size* is specified as the size. *super-or-use* should be either the name of a single package, to be specified as the *super* argument to **make-package**, or a list of package names, to be specified as the *use* argument to **make-package**.

pkg-find-package *name* &optional *create-p* Function

(Not in COMMON LISP) *name* and *use-local-nickname-pkg* are passed to **find-package** (see page 60). If that returns a package, **pkg-find-package** returns the same package. Otherwise, a package may be created, according to the value of **create-p**. These values are allowed:

nil	An error is signaled if an existing package is not found.
t	A package is always created.
:find	**nil** is returned.
:ask	The user is asked whether to create a package.

If a package is created, it is done by calling **make-package** with *name* as the only argument. This function is not quite for compatibility only, since certain values of *create-p* provide useful features.

5.4 Creation and Destruction of Packages

While **package-declare** still works, the standard way to create a package now is the new function **make-package** or the defining construct **defpackage**. To eliminate one, use **kill-package** (see page 63).

make-package *name* &key *nicknames use super shadow export prefix-name size invisible import shadowing-import import-from relative-names relative-names-for-me* Function

Creates a new package with name *name* (a string) and nicknames *nicknames* (a list of strings). It is initially made large enough to hold at least *size* symbols before needing expansion. The package is returned as the value.

The following keyword arguments are accepted:

:use	A list of packages or names for packages from which the new package should inherit or a single name or package. It defaults to just the **global** package.
:super	If this is non-**nil**, it should be a package or name to be the superpackage of the new package. The new package will inherit from the superpackage and from all

the other packages from which the superpackage inherits. The superpackage itself is marked as autoexporting; see the section "External and Internal Symbols" for more information. Superpackages are implemented for compatibility only; they are not recommended for use in any new package definitions.

:prefix-name
: This specifies the name to use for printing package prefixes that refer to this package. It must be either the name or one of the nicknames. The default is to use the *name*.

:shadow
: A list of strings that are names for symbols that should be shadowed in the newly created package. This argument is passed directly to the function **shadow** (see page 68).

:export
: A list of symbols or names to export in the new package. This is handled by the function **export** (see page 64).

:nicknames and **:use** are the only arguments allowed in transportable COMMON LISP. All of keyword arguments are for ZETALISP only.

:invisible
: If non-**nil**, means that this package should not be put on the list ***all-packages*** (see page 59). As a result, **find-package** will not find this package, either by its name or by its nicknames. You can make normal use of the package in all other respects (passing it to **intern**, passing it to **use-package** to make other packages inherit from it or it from others, and so on).

:import
: If non-**nil**, is a symbol or a list of symbols to be imported into this package. You could accomplish as much by calling **import** after you have created the package.

:shadowing-import
: If non-**nil**, is a symbol or a list of symbols to be imported into this package with shadowing. You could accomplish as much by calling **shadowing-import** after you have created the package.

:import-from
: If non-**nil**, is a list containing a package (or package name) followed by names of symbols to import from that package. Specifying *import-from* as **(chaos "open" "close")** is nearly the same as specifying *import* as **(chaos:open chaos:close)**, the difference being that with *import-from* the symbols *open* and *close* are not looked up in the **chaos** package until it is time to import them.

:relative-names
: An alist specifying the local nicknames to have in this package for other packages. Each element looks like (*localname* . *package*), where *package* is a package or a name for one, and *localname* is the desired local nickname.

:relative-names-for-me
: An alist specifying local nicknames by which this package can be referred to from other packages. Each element looks like (*package localname*), where *package* is a package name and *localname* is the name to refer to this package by from package. You will note that the elements of this list are not dotted while those of **:relative-names** are.

pkg-create-package *name* &optional (*super* ***package***) (*size* **200**) Function

(Not in COMMON LISP) Creates a new package named name of size size with superpackage super. This is for compatibility only.

defpackage &quote *name keywords-and-values...* Special form

(Not in COMMON LISP) This is the preferred way to create a package in ZETALISP. (It is compatible with the **defpackage** introduced in Symbolics Release 5.) All the arguments are simply passed to **make-package** (see page 60). The differences between this function (actually, macro) and **make-package** are:

- **defpackage** does not evaluate any arguments.
- Re-evaluating a **defpackage** for an existing package is allowed; it modifies the existing package in accordance with changes in the definition.
- The editor notices **defpackage** and records it as the "definition" of the package.

IMPORTANT: The latest edition of the *Lisp Machine Manual* documented this function to take arguments in **&key** (property-list) style. However, the keywords and values are actually supposed to be passed in an association list form. *Ignore the version in the manual.* For example, here is the correct version of the example given the bottom of page 653 in the *Lisp Machine Manual*:

```
(defpackage "EH"
  (:size 1200)
  (:use "GLOBAL" "SYS")
  (:nicknames "DBG" "DEBUGGER")
  (:shadow "ARG"))
```

Package attributes in a file's `-*-` line can now have this format

`Package: (`*name keyword value keyword value...*`);`

which means that the package to be used is name and, if that package does not exist, it should be created by passing name and the keywords and values to **make-package**.

sys:package-not-found Condition

This error condition is signalled whenever you do **pkg-find-package** with second argument **:error**, **nil** or omitted, and the package you were looking for does not exist.

The condition instance supports the operations **:name** and **:relative-to**; these return whatever was passed as the first and third arguments to **pkg-find-package** (the package name, and the package whose local nicknames should be searched).

The proceed types **:retry**, **:no-action**, **:new-name** and **:create-package** may be available.

:retry Says to search again for the specified name in case it has become defined; if it is still undefined, the error occurs again.

:create-package
Says to search again for the specified name, and create a package with that name if none exists yet.

:new-name Is accompanied by a name (a string) as an argument. That name is used instead, ignoring any local nicknames. If that name too is not found, another error occurs.

:no-action (Available on errors from within **read**.) Says to continue with the entire **read** as well as is possible without having a valid package.

package-declare &quote *name super size unused body...* Special form

(Not in COMMON LISP) Is one old-fashioned equivalent of **defpackage** (see page 61). It is no longer recommended for use. It creates a package named *name* with superpackage *super* (another name) and initial size *size*. The unused argument must be **nil**. *body* is now allowed to contain only these types of elements:

shadow *names*
Passes the names to the function **shadow** (see page 68).

external *names*
Does nothing. This controlled an old feature that no longer exists.

intern *names*
Converts each name to a string and interns it in the package.

refname *refname packagename*
Makes *refname* a local nickname in this package for the package named *packagename.*

myrefname *packagename refname*
Makes *refname* a local nickname in the package named *packagename* for this package. If *packagename* is **global**, makes *refname* a global nickname for this package.

pkg-add-relative-name *in-pkg name for-pkg* Function
(Not in COMMON LISP) Defines *name* as a local nickname in *in-pkg* for *for-pkg. in-pkg* and *for-pkg* may be packages, symbols or strings.

pkg-delete-relative-name *in-pkg name* Function
(Not in COMMON LISP) Eliminates *name* as a local nickname in *in-pkg.*

kill-package *name-or-package* Function
(Not in COMMON LISP) Kills the package specified or named. The name **pkg-kill** is also allowed for compatibility.

5.5 Package Inheritance

You now may completely control which packages are inherited by which other packages. Inheritance no longer has to be transitive. **x** can inherit from **y** and **y** from **z** but without **x** inheriting from **z** also. Inheritance can also be multiple. **x** can inherit from two unrelated packages **y** and **w**. However, in any case, only external symbols are inherited. More information on internal vs external symbols is in the following section.

In the past, a package would inherit only from its superior, its superior's superior, and so on. Thus, if **bar** and **quux** were two subpackages of **global**, a new package **foo** could inherit from **bar** and **global**, or from **quux** and **global**, or just from **global**; but **foo** could not inherit from **bar** alone, or **quux** alone, or from **bar** and **quux**, or from **bar** and **quux** and **global**. Now any of these possibilities is possible.

The functions **use-package** and **unuse-package** are used to control the inheritance possibilities of an existing package. The **:use** argument to **make-package** can be used to specify them when a package is created. If **foo** inherits from **bar**, we also say that **foo uses bar.**

use-package *packages* &optional (*in-package* ***package***) Function
Makes *in-package* inherit symbols from *packages*, which should be either a single package or name for a package, or a list of packages and/or names for *packages.*

unuse-package *packages* &optional (*in-package* ***package***) Function
Makes in-package cease to inherit symbols from packages.

package-use-list *package* Function
Returns the list of packages **used** by *package.*

package-used-by-list *package* Function
Returns the list of packages that **use** *package*.

You can add or remove **used** packages at any time.

If one package **uses** several others, the **used** packages are not supposed to have any two distinct symbols with the same pname among them all. An attempt to create such a situation causes an error, which you can override by shadowing. (See below.)

5.6 External and Internal Symbols

Each symbol in a package is marked as external or internal in that package. Symbols created in the package by **intern** are initially internal. You must mark symbols as external if you want them to be so.

The plan is that all the symbols in a package that are intended to be used from other packages will be marked as external.

The internal versus external distinction makes a difference at two times:

Only external symbols are inherited from other packages

In COMMON LISP, only external symbols can be referred to with ~~ordinary~~ colon prefixes. :: prefixes must be used for internals.

intern (see page 65) works by first checking the current or specified package for any symbol, whether external or not, and then checking all the inherited packages for external symbols only. All the symbols in **global** and **system** are external to start with, so that they can still be inherited, and all new symbols made in them are made external. All symbols in the **keyword** package are also automatically external.

Some other packages also automatically export all symbols put in them. This happens, for compatibility, in any package that has been specified as the "superpackage" in the old-fashioned **package-declare** and **pkg-create-package** functions. You are not allowed to **pkg-goto** one of these packages, and **read** makes a special check to prevent you from creating symbols in them with package prefixes.

Relevant functions:

export *symbols* &optional (*package* ***package***) Function
Makes *symbols* external in *package*. *symbols* should be a symbol or string or a list of symbols and/or strings. The specified symbols or strings are interned in package, and the symbols found are marked external in package.
If one of the specified symbols is found by inheritance from a **used** package, it is interned locally in *package* and then marked external.

unexport *symbols* &optional (*package* ***package***) Function
Makes *symbols* not be external in *package*. It is an error if any of the symbols to be marked not external are not directly present in package.

globalize *name-or-symbol* &optional (*into-package* **"GLOBAL"**) Function
If *name-or-symbol* is a name (a string), interns the name in into-package and then forwards together all symbols with the same name in all the packages that **use** into-package as well as in **into-package** itself.
If *name-or-symbol* is a symbol, interns that symbol in into-package, and then forwards together all symbols with the same name.
The symbol ultimately present in *into-package* is also exported.

pkg-external-symbols *package* Function

Returns a list of all the external symbols of package. *package* can be a package or a package name.

5.7 Looking Up Symbols

The four old functions for looking up symbols work with minor changes. There are also two new ones.

intern *symbol-or-string* &optional *package* Function

Looks up the specified name in the specified package and inherited packages. If *package* is omitted or **nil**, the current package is used.

If a string is specified, a symbol of that name is looked for first in the specified package and then in each of the packages it inherits from. If a symbol is found, it is returned. Otherwise, a new symbol with that name is created and inserted in the specified package, and returned.

If a symbol is specified, lookup proceeds using the symbol's pname as the string to look for. But if no existing symbol is found, the specified symbol itself is inserted in the package. No new symbol is made. Use of a symbol as argument is not defined in COMMON LISP.

intern actually returns three values. The first is the symbol found or created. The second is a flag that says whether an existing symbol was found, and how. The third is the package in which the symbol was actually found or inserted. It will be the specified package or a package from which the specified package inherits.

The possible second values are:

nil	Nothing was found. The symbol returned was just inserted.
:internal	The symbol was found as an internal symbol in the specified package.
:external	The symbol was found as an external symbol in the specified package.
:inherited	The symbol was inherited from some other package (where it was necessarily an external symbol).

intern-soft *symbol-or-string* &optional *package* Function
find-symbol *symbol-or-string* &optional *package* Function

(**find-symbol** is the COMMON LISP name.) Looks for an existing symbol like **intern**, but never creates a symbol or inserts one into package. If no existing symbol is found, all three values are **nil**.

package defaults to the current package if it is omitted or given as **nil**.

intern-local *symbol-or-string* &optional *package* Function

(Not a COMMON LISP function) Like **intern** but looks only in *package*, ignoring the packages *package* normally inherits from. If no existing symbol is found in *package* itself, the specified symbol or a newly created symbol is inserted in *package*, where it permanently shadows any symbol that previously would have been inherited from another package.

The third value is always *package*, and the second one is never **:inherited**.

package defaults to the current package if it is omitted or given as **nil**.

intern-local-soft *symbol-or-string* &optional *package* Function

(Not a COMMON LISP function) Like **intern-soft** but looks only in *package*, ignoring the packages it normally inherits from. If no symbol with the specified name is found in *package*, all three values are **nil**.

package defaults to the current package if it is omitted or given as **nil**.

remob *symbol* &optional *package* Function
unintern *symbol* &optional (*package* ***package***) Function

(**unintern** is the COMMON LISP name) Removes *symbol* from being present in *package*. In **remob**, *package* defaults to *symbol*'s package. In **unintern**, it defaults to the current package.

If a shadowing symbol is removed, a previously-hidden name conflict between distinct symbols with the same name in two USEd packages can suddenly be exposed, like a discovered check in chess. This signals an error.

import *symbols* &optional (*package* ***package***) Function

Is the standard COMMON LISP way to insert a specific symbol or symbols into a package. *symbols* is a symbol or a list of symbols. Each of the specified symbols will be inserted into package, just as **intern** (see page 65) would do.

If a symbol with the same name is already present (directly or by inheritance) in package, an error is signaled. On proceeding, you can say whether to leave the old symbol there or replace it with the one specified in **import**.

5.8 Looping Over Symbols

Several new macros are available for writing loops that run over all the symbols in a package.

do-symbols (*var package result-form*) *body...* Macro

Executes *body* once for each symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once if it is present directly in two or more of package and the inherited packages.

do-local-symbols (*var package result-form*) *body...* Macro

(Not a COMMON LISP form) Executes *body* once for each symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally *result-form* is executed and its values are returned.

do-external-symbols (*var package result-form*) *body...* Macro

Executes *body* once for each external symbol findable in package either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once if it is present directly in two or more of package and the inherited packages.

do-local-external-symbols (*var package result-form*) *body...* Macro

(Not a COMMON LISP form.) Executes *body* once for each external symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-all-symbols (*var result-form*) *body...* Macro

Executes *body* once for each symbol present in any package. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once.

These old functions still work:

mapatoms *function* &optional (*package* **"GLOBAL"**) (*inherited-p* **t**) Function
Calls *function* successively on each of the symbols in *package*. Symbols inherited from other packages are included if *inherited-p* is non-**nil**.

mapatoms-all *function* &optional (*package* **"GLOBAL"**) Function
Calls *function* successively on each of the symbols in *package* and all the packages that inherit from package. When *package* has its default value, this will include just about all packages.

5.9 The USER Package

In Release 2, the **user** package is an ordinary package that inherits from **global**.

The **user** package used to be the same as the **keyword** package, so in files read into **user** it was not necessary to put a colon on any keyword. This is no longer the case. You must use colons in the **user** package just as in any other package.

5.10 Package Prefixes

In COMMON LISP, a package prefix is used before a symbol to refer to a symbol that is not present or inherited in the current package. (In ZETALISP and NIL, one can also put prefixes in front of any form, and the package prefix will be pervasive during the reading of that form.) **tv:tem** is an example; it refers to the symbol with the print-name **tem** that is visible in the package named **tv**. (**tv** can be the primary name or a nickname.)

Internal symbols that print with package prefixes will print with :: prefixes, as in **tv::tem**, rather than as **tv:tem**. This is because in COMMON LISP a simple colon prefix can be used only for external symbols; a :: prefix must be used if the symbol is internal.

This restriction has *not* been implemented for ZETALISP programs. The colon prefixes in your programs will still work ! But :: prefixes are being printed for informational purposes, and will be accepted by the reader.

A prefix consisting of just **#**: indicates an uninterned symbol. Uninterned symbols are printed with such prefixes, and **#**: can also be used in input to create an uninterned symbol.

Package prefixes are normally decoded when read by checking the local nicknames, if any, of the current package and its superpackages before looking at the actual names and nicknames of packages. You can use a **#** before the colon in the prefix to prevent the use of the local nicknames. Suppose that the current package has **tv** as a local nickname for the **xtv** package. Then **tv:sheet will** get the **sheet** in the **xtv** package, but **tv#:sheet** will get the one in the **tv** package. That symbol will print out as **tv#:sheet** as well, if the printer sees that **tv:sheet** would be misinterpreted by the reader.

The package name in a package prefix is read just like a symbol name. This means that slash and vertical bars can be used to include special characters in the package name. Thus, **foo/:bar:test** refers to the symbol **test** in the **foo:bar** package, and so does **|foo:bar|:test**. Also, letters are converted to upper case unless they are quoted with a slash or vertical bar. For this reason, package names should normally be all upper case.

5.11 Shadowing and Name Conflicts

If multiple symbols with the same name are available in a single package, counting both symbols interned in that package and external symbols inherited from other packages, we say that a name conflict exists.

Name conflicts are not permitted to exist unless a resolution for the conflict has been stated in advance by specifying explicitly which symbol is actually to be seen in package. This is done by shadowing. If no resolution has been specified, any command that would create a name conflict signals an error instead.

For example, a name conflict can be created by **use-package** if it adds a new inherited package with its own symbol **foo** to a package which already has or inherits a different symbol with the same name **foo**. **export** can cause a name conflict if the symbol becoming external is now supposed to be inherited by another package that already has a conflicting symbol. On either occasion, if shadowing has not already been used to control the outcome, an error is signaled and the **use** or exportation does not occur.

Shadowing means marking the symbol actually interned in a package as a shadowing symbol, which means that any conflicting symbols are to be ignored.

package-shadowing-symbols *package* Function

Returns the list of shadowing symbols of *package*. Each of these is a symbol interned in package. When a symbol is interned in more than one package, it can be a shadowing symbol in one and not in another.

Once a package has a shadowing symbol named FOO in it, any other potentially conflicting external symbols with name FOO can come and go in the inherited packages with no effect.

There are two ways to request shadowing: **shadow** and **shadow-import**.

shadow *names* &optional (*package* ***package***) Function

Makes sure that shadowing symbols with the specified *names* exist in *package*. *names* is either a string or symbol or a list of such; any symbols present in *names* are coerced into their print-name strings. Each name specified is handled independently as follows:

- If there is a symbol of that name interned in *package*, it is marked as a shadowing symbol.
- Otherwise, a new symbol of that name is created and interned in *package*, and marked as a shadowing symbol.

In any case, *package* will have a symbol with the specified name interned directly in it and marked as a shadowing symbol.

The primary application of **shadow** is for causing certain symbols not to be inherited from any of the **used** packages. To avoid problems, the **shadow** should be done right after the package is created. The **:shadow** keyword to **make-package** (see page 60) or **defpackage** (see page 61) lets you specify names to be shadowed in this way when you create a package.

shadowing-import *symbols* &optional (*package* ***package***) Function

Interns the specified symbols in *package* and marks them as shadowing symbols. *symbols* must be a list of symbols or a single symbol; strings are not allowed.

Each symbol specified is placed directly into *package*, after first removing any symbol with the same name already interned in package. This is rather drastic, so it is best to use **shadowing-import** right after creating a package.

shadowing-import is useful primarily for choosing one of several conflicting external symbols present in packages to be **used**.

6. Window System Changes

6.1 The FONTS Package No Longer Uses Global

This means that any fonts created in earlier systems will have to be redumped in order to work with Release 2. This has been done for all the system's fonts appearing in the **SYS: FONTS:** directory. There are two ways to do update the fonts to run in Release 2. The first is to write out (using **fed**) a **kst** format file of the font, load that into a Release 2 world and then write out a **qfasl** font file. The other technique is to do the following (in Release 2):

```
(use-package "GLOBAL" "FONTS")
(load file-containing-font)
(unuse-package "GLOBAL" "FONTS")
(compiler:fasd-symbol-value file-to-contain-font 'fonts:name-of-font)
```

6.2 New way of initializing process of TV:PROCESS-MIXIN

Normally, if the *process* keyword argument to **make-instance** of some window flavor incorporating **tv:process-mixin** is a symbol, it is used as the top level function and **make-process** is called with no keyword arguments. But, as an exception, if **process** is **t**, the top level function is to send the window a **:process-top-level** message with no arguments. So, for example, one could write:

```
(defflavor crock-window ()
                        (tv:process-mixin tv:window)
  (:default-init-plist :process t)
  (:documentation "A window which displays a crock."))

(defmethod (crock-window :process-top-level) ()
  (draw-crock self)
  (do-forever
    (update-hands)
    (process-sleep 60.)))
```

6.3 TV:SHEET-FORCE-ACCESS Does Not Prepare the Sheet

The macro **tv:sheet-force-access** (documented in the *Window System Manual*) used to put a **tv:prepare-sheet** into its expansion unless an optional argument was supplied to inhibit doing so.

It turned out that most uses of the macro had no need to prepare the sheet but were neglecting to supply the optional argument. Since combining the two facilities is unmodular, the **prepare-sheet** has simply been flushed from **tv:sheet-force-access**. If you really want to do one, simply write a **tv:prepare-sheet** explicitly in the body of the **tv:sheet-force-access**.

The old optional *dont-prepare-flag* argument is still accepted but has no effect now.

6.4 TV:MAKE-WINDOW Now Identical to MAKE-INSTANCE

Windows can now be created with **make-instance** just like any other flavor instances. The function **tv:make-window** will be supported indefinitely since it is so widely used.

6.5 TV:MOUSE-WAKEUP and TV:MOUSE-RECONSIDER

The window manual says that you should call the function **tv:mouse-wakeup** to report a change in screen configuration. This is not exactly true.

The function **tv:mouse-wakeup** causes the mouse process to look again at the position of the mouse. It is called by the function **tv:mouse-warp**, so that the mouse will be tracked to its specified new position. It is also the thing to use if you redisplay a menu-like window with a new set of menu items, for example, so that the mouse process will notice whether the mouse position is now inside a different menu item.

However, actual changes in the window configuration may make it necessary to force recomputation of which window owns the mouse. This is done by setting the variable **tv:mouse-reconsider** non-**nil**. Calling **tv:mouse-wakeup** may not be enough, since the current mouse position may still be inside the old screen area of a no-longer-eligible window.

6.6 Mouse Clicks Are Blips By Default

If a window has an input buffer and does not define a handler for mouse clicks, they are handled by putting **:mouse-click** blips into the input buffer. It used to be necessary to mix in **tv:list-mouse-buttons-mixin** to get this behavior. Now that flavor is a no-op.

Refer to section 10.1 of the *Window System Manual* for more information.

6.7 :PREEMPTABLE-READ for TV:STREAM-MIXIN

Now all windows that handle **:rubout-handler** also handle the **:preemptable-read** operation. It used to be necessary to mix in **tv:preemptable-read-any-tyi-mixin** to have this operation available. That flavor is now a no-op.

Refer to page 55 of the *Window System Manual* for information on using this operation.

You can also do preemptable input using the **:rubout-handler** operation with the **:preemptable** option. This is a new feature documented in this file.

6.8 Menu Item Types

The value of a **:menu** menu item can now be any form that evaluates to a suitable menu. A menu itself is a special case of such a form, now that menus and other unusual objects evaluate to themselves.

:funcall-with-self is a new type of menu item. The value associated with it is a function of one argument. If the menu item is executed, the function will be called, with the menu (that is the value **self**, in the menu's **:execute** method) as its argument. The value that the function returns is the value of executing the menu item.

6.9 TV:MOUSE-WAIT Takes Who-state as Argument

tv:mouse-wait takes an additional optional argument that, if specified, is displayed as the run state in the who line while the function waits for mouse input.

6.10 Mouse Characters

window

You should no longer use the byte pointer **%%kbd-mouse** in making mouse characters or testing whether a character is a mouse character. It still works at the moment, but may stop working in the future. To avoid problems, convert code as soon as you have switched over to Release 2.

To test, use **tv:char-mouse-p**. To construct, use **tv:make-mouse-char**.

tv:char-mouse-p *char* Function

t if *char* is a mouse character. This function was incorrectly documented as **tv:kbd-mouse-p** in the *Lisp Machine Manual*.

tv:make-mouse-char *button n-clicks* Function

Returns the mouse character for clicking on button *button*, *n-clicks* times. Both *button* and *n-clicks* range from 0 to 2; *n-clicks* is actually one less than the number of clicks. The left button is button 0; the right one is 2.

Continue to use **%%kbd-mouse-button** and **%%kbd-mouse-n-clicks** as byte pointers to extract from a mouse character which button was clicked and how many times.

6.11 TV:MARGIN-SPACE-MIXIN

The mixin **tv:margin-space-mixin** defines a blank margin item. You can leave blank space next to any of the window's edges. The blank space can go between two margin items at that edge, or between the inside of the window and the margin items. For example, it can be used to separate the scroll bar from the inside of the window, or separate the scroll bar from the border, depending on where in the ordering you mix the mixin in.

The mixin defines an init keyword called **:space** whose value specifies how much blank space to leave at each edge. The values you can use are:

t Leaves one pixel of space at all four edges.

nil Leaves no blank space. This turns off the effect of the mixin.

n Leaves *n* pixels of space at each edge.

left top right bottom
Leaves top pixels of space at the top edge, left pixels at the left edge, etc.

Two operations are also defined by the mixin: **:space** and **:set-space**. **:set-space** takes an argument just like the **:space** init keyword and alters the amount of space the mixin is generating. **:space** as an operation returns a list of four values (left top right bottom) describing how much space is currently being taken up by the mixin.

6.12 TV:ADD-SYSTEM-KEY Improvement

If a system key is already defined and you use **tv:add-system-key** to redefine it, the previous definition is restored when you do **tv:remove-system-key** to remove the new definition.

6.13 New String Drawing Primitive

The following function has been added to help speed up string drawing. It is compatible with the Symbolics function of the same name.

tv:%draw-string *sheet alu xpos ypos string font start stop xlim* Function

Draw *string* on *sheet* starting with the character at index *start* and stopping after drawing the character at index *stop*, presuming it all fits. Output starts at *xpos*, *ypos* on the sheet and continues until all appropriate characters are drawn, or until the next character to be drawn would extend past *xlim*. The index of the next character to be drawn, and the xpos where it would go are returned. If a **newline** is encountered, **tv:%draw-string** returns its index and xpos immediately. The sheet's cursor position is ignored and left unchanged.

This function also handles fonted (**art-fat-string** or 16-bit) strings. Therefore, the function **tv:sheet-fat-string-out** is now obsolete; use **tv:sheet-string-out**. The message **:fat-string-out** is also obsolete; use the message **:string-out**.

7. User Interface Changes

The section on the Yank system in the Editor chapter (section 8.2, page 79) is also very relevant to the user interface.

7.1 New function COMMON-LISP

When this function is called in a Lisp Listener, it changes whether COMMON LISP or traditional ZETALISP (actually, their syntax and incompatible functions) are to be used for reading and printing lisp objects. It works by **setqing** ***readtable***.

It takes one argument, which should be either **t** or **nil**.

See chapter 1, page 3 for basic information on COMMON LISP support. See section 8.11.1, page 82 for COMMON LISP support in ZMacs.

7.2 New Run Bar

A new run bar can occasionally be seen at the bottom of the screen, to the left of the older run bars. This bar goes on whenever the machine would take a sequence break (see the *Lisp Machine Manual* chapter on processes), but cannot because **inhibit-scheduling-flag** is non-**nil**.

7.3 Arguments to APROPOS and WHERE-IS changed

apropos *substring* &optional (*package* ***all-packages***) &key (*inheritors* **nil**) (*inherited* **t**) *dont-print predicate boundp fboundp* — Function

The *package* argument is now the always the second argument (it used to be a keyword argument) The value of this argument may be **nil**, meaning to search all packages, a single package or package name, or a list of packages and/or package names.

where-is now accepts a package or a package name or a list of packages and/or package names as it second argument.

7.4 Beep Types

The system now supplies a non-**nil** beep-type to the function **beep** on certain occasions. These are the types defined so far:

zwei:converse-problem
Used for the beep that is done when Converse is unable to send a message.

zwei:converse-message-received
Used for the beeps done when a Converse message is received.

zwei:no-completion
Used when you ask for completion in the editor and the string does not complete.

tv:notify
Used for the beep done when you get a notification that cannot be printed on the selected window.

supdup:terminal-bell
Used when the remote host sends a "bell" character over while using SUPDUP.

fquery
Used when the **fquery** function beeps for attention.

Those of you who redefine **beep** can use the beep type (the first argument) to produce different sounds for different occasions. More standard beep types will be defined in the future, if users suggest occasions that deserve beep types.

7.5 *VALUES* for Evaluator Loops

LISP Listeners, **break** loops, and the debugger now record all the values of each evaluated form in the variable ***values***. Each process has its own ***values***. The value of ***values*** is a list, and each element is a list of the values of one evaluated form. The most recent forms' values come first.

If a form is aborted for any reason, **nil** is pushed on ***values*** for it.

(caar *values*) is therefore equivalent to the value of the variable ***** if and only if the last form was not aborted.

7.6 Variable Ratio Mouse Motion

The ratio of mouse motion on the table to mouse cursor motion on the screen now depends on the speed of motion. If you move the mouse slowly, the cursor moves only a little as the mouse moves. As you move the mouse faster, the same amount of mouse motion moves the cursor a long distance.

To control this feature, use this function:

tv:mouse-speed-hack &rest *specs* Function

specs consists of an odd number of elements: alternating scale factors and speeds, followed by one more scale factor. Each scale factor applies up to the speed that follows it. The last scale factor applies to all higher speeds. The standard settings are made with specs of (.6 120 1 200 1.5 400 2.2 700 3.3) so you can see that a speed of 120 is fairly slow, while 700 is moderately fast. A scale factor of 1 corresponds to the mouse motion ratio previously in use. So, **(tv:mouse-speed-hack 1)** would restore the old fixed-ratio behavior.

7.7 Evaluating/Compiling Multi-Font Files.

It now works to evaluate or compile files that contain multiple fonts as specified with the **Fonts** attribute in the -*- line. The old kludge that some users used for doing this should no longer be used.

To make this work in all cases, user-defined readmacro characters should do all input using the function **si:xr-xrtyi** (see its on-line documentation). You may wish to specify arguments of *stream* **nil t**.

Note that if a reader macro detects a syntax error and wants to report this by signaling an Lisp error, it should always make **sys:read-error** one of the condition names and provide the proceed-type **:no-action**, which should be handled by skipping over the invalid data and returning *something* (**nil** is a reasonable thing to return).

7.8 Debugging Changes

7.8.1 Evaluation in the Debugger

When you evaluate an expression in the debugger, it is evaluated in the binding environment of the frame that is current in the debugger.

Initially, the debugger starts out with its current frame being the one in which the error happened. Therefore, your expressions are evaluated in the environment of the error. However, you now have the option of evaluating them in other environments instead.

The debugger command **Meta-S** is no longer necessary in most cases, since simply evaluating the special variable will get the same result. But it is still useful with a few variables such as ***standard-input*** and **eh:condition-handlers** which are rebound by the debugger for your protection when you evaluate anything.

7.8.2 UNADVISE

The function **unadvise** has been generalized in that all arguments now act independently to restrict which pieces of advice should be removed. Thus, if all three arguments are **nil**, all advice is removed. If the first argument is non-**nil**, it is a function spec, and only advice on that function spec is removed. If the second argument is non-**nil**, it is an advice class (**:before**, **:after** or **:around**), and only advice of that class is removed. If the third argument is non-**nil**, it is a position (if it is a number) or a name (if it is a symbol), and only advice with that position or that number is removed.

unadvise-within has been improved in a similar fashion.

7.8.3 :STEPCOND Argument to TRACE

The **:stepcond** argument to TRACE generalizes the :STEP argument. It allows you to specify that STEP should be invoked on the execution of the traced function only if a certain condition is met. The value you provide for the **:stepcond** argument should be a form to be evaluated when the traced function is called; if the form evaluates non-**nil**, the function will be stepped.

7.8.4 MONITOR-VARIABLE No Longer Exists

One consequence of the fact that boxed data words no longer have a flag bit is that **monitor-variable** is no longer possible to implement. This function has been removed.

7.8.5 Describing Condition Handlers

The debugger command **Control-Meta-H** prints a description of the condition handlers established by the stack frame you are looking at.

7.8.6 Overriding *DEBUG-IO*

eh:*debug-io-override* Variable

If **eh:*debug-io-override*** is non-**nil**, the debugger will now use it for its input and output, rather than using the value of ***debug-io***.

7.9 Choose Variable Values Windows

Clicking the right mouse button on a variable's value now puts you in the rubout handler with the old value of the variable there for you to edit. You can use parts of the text of the old value to make up the text of the new value.

Clicking left still puts you in the rubout handler with a blank slate; then you must type the new value from scratch.

7.10 Output of Character Names

The **format** directive `~:C` and the function **format:ochar** now never output a character name for graphic characters other than **Space** and **Altmode**. All other graphic characters are output as themselves, whether or not they have names, since they appear on the keyboard as themselves.

7.11 Terminal T Change

Terminal T now controls just the deexposed Typeout action of the selected window. A new command **Terminal I** controls the deexposed type-In action. (Sadly, **Terminal O** is already in use).

Terminal 0 T
: Just wait for exposure on output when deexposed.

Terminal 1 T
: Notify user on attempt to do output when deexposed

Terminal 2 T
: Permit output when deexposed.

Terminal 0 I
: Just wait for exposure on input when deexposed.

Terminal 1 I
: Notify user on attempt to do input when deexposed

Terminal 2 I
: There is no **Terminal 2 I**. It doesn't make sense.

7.12 Terminal c-Clear-Input is now Terminal c-M-Clear-Input

This keyboard sequence is used to try to unhang some window-system problems. It has been changed so that **c-clear-input** is typeable (by having it quoted with **terminal**, which causes it to lose its special meaning of "flush keyboard typeahead" and be simply passed on to the program which is reading from the keyboard.)

7.13 DRIBBLE-START, DRIBBLE-END gone

Use **(dribble** *filename***)** or **(dribble-all** *filename***)** to start wallpapering output to a file, and **dribble** with no arguments to terminate output and close the file.

7.14 Compiler Behavior

The areas in which compiled code lives are now read-only. This is to catch bugs such as **nconc**ing onto a constant list. The print names of interned symbols are also read-~~only~~ now.

Compilation no longer uses fixed data structures that exist in only one copy. You will no longer get the message "Compiler in process FOO waiting for resources."

7.15 MAKE-SYSTEM Improvements

If **make-system** is done on a system that is not known, the file **SYS: SITE:** *system* **SYSTEM** is now loaded without any query if the file exists.

When **make-system** asks you about a list of files to be compiled or loaded, you now have the option of saying you would like to be asked again about each individual file. Do this by typing **S** instead of **Y** or **N**.

After you type **S**, you will be asked about each file in the bunch just as you would have been earlier if you had specified **:selective** as an argument to **make-system**. Finally you will be asked once again to approve of the entire bunch of files, before processing actually begins.

7.16 APROPOS and SUB-APROPOS Extended

In **apropos**, specifying a non-**nil** value for the keyword argument boundp restricts the search to symbols that have values. A non-**nil** **:fboundp** argument restricts it to symbols with function definitions. **sub-apropos** accepts the same new arguments.

7.17 LOAD Defaults Are the Default Defaults

COMMON LISP wants **load** to use the default defaults. It seems that if **load** should do so then everything else that used the **load** defaults should do likewise. So the two variables (**cli:load-pathname-defaults** and **fs:load-pathname-defaults**) have been forwarded together.

7.18 Hardcopy Options

Now, options to the system-defined hardcopy functions can be defaulted on a per-printer-type basis.

set-printer-default-option *printer-type option value* Function

This function allows the user to set a default option for a printer type, which the hardcopy functions look at. A common use at MIT may be **(set-printer-default-option :dover :spool t)**, which will cause Dover output to be spooled unless the **:spool** option to a hardcopy function is supplied. Currently defaultable options are **:font**, **:font-list**, **:heading-font**, **:page-headings**, **:vsp**, **:copies**, and **:spool**.

7.19 ZMail Changes

On the LAMBDA, Zmail is now a supported system. There will be a new manual, and introductory documentation as well.

7.19.1 Message-ID Fields.

If you want, ZMail can put a Message-ID field in your outgoing messages. Go into the Profile editor to get this behavior, because the default is not to generate Message-ID fields.

7.19.2 New Command M-X Undigestify Message

This command takes the current message and splits it into its submitted messages so that you can act on them individually. You can set aspects of what the command does by using the Profile editor:

1. Should the original message be deleted ? (Default: *Yes*)
2. Should everything but the header and "table of contents" be clipped out of the original message ? (Default: *No*)
3. Should the name of the digest be append to the subject field of all the new messages so that you can tell from which digest they came ? (Default: *Yes*)

7.19.3 Usual mail file directory option for ZMail

You can set this option in the Profile editor in ZMail. It simply informs ZMail to use a short name for a mail file in a menu, if that file is found in the directory. (The full name of the file is displayed if it has not been read into a buffer yet.)

8. Editor Changes

This chapter covers changes in command names, the yank system, and the rubout handler, among other things.

8.1 Selective Undo

You can now undo an editing change that is not the most recent change you made. If you give the Undo command **C-Shift-U** while there is a region, it undoes the most recent batch of changes that falls within the region. The region does not go away, so you can repeat the command to undo successive changes within the same region. For example, you can undo your changes to a specific Lisp function by using **C-M-H** to create a region around it and then using C-Shift-U.

8.2 Yank Command Improvements

What used to be called the *kill ring* is now called the *kill history* because it is no longer a ring buffer. It now records all the kills you have ever done, in strict chronological order.

Meta-Y still brings older kills into the region, and any particular sequence of **Meta-Y** commands works just as it used to. But the history is not permanently rotated; as soon as a new kill is done, it snaps back to chronological order. We say that **Meta-Y** rotates the history's yank pointer around the history list. **Control-Y** with no argument yanks what the yank pointer points at.

So far, the yank pointer corresponds entirely to what used to be the front of the kill ring, but here are the differences.

- Killing anything moves the yank pointer up to the front of the list.
- Numeric arguments to **Control-Y** count from the most recent kill, not from the yank pointer.

You can think of this as meaning that either killing or using **c-Y** with an argument "un-rotates" any rotation you have done, before it does its work.

Control-Y with an argument of zero prints a list of the first 20 elements of the kill history. Click on one of them to yank it. Click on the message saying that there are more elements, if you want to see the rest of them.

There are several other histories as well as the kill history. They all work just like the kill history, except that you use some other command instead of **Control-Y** to yank from them. **Meta-Y** is used for rotating the yank pointer no matter which history you are yanking from; it simply works on whatever history your last yank used.

For example, the previous inputs in the rubout handler are now stored in a history. The command **Control-C** yanks from it, much as before, except that it now takes arguments exactly like **Control-Y**. **Control-Meta-Y** is a new alias for **Control-C**; it has the advantage of not being a debugger command, so you can use it in the debugger with no extra complications.

All the pathnames you have typed in minibuffers now go in a history. The command **Meta-Shift-Y**, which used to yank the last pathname input, has been generalized to yanks from this history. It takes args just like **Control-Y**, now. Use **Meta-Y** immediately after a **Meta-Shift-Y** to rotate the yank pointer to other pathnames in the history.

All buffer names given as arguments in the minibuffer also have a history. (Actually, each ZMACS window has its own history of these.) **Meta-Shift-Y** is the command for this history as well.

All function specs and other definition names you give as arguments in the minibuffer also have their own history, which is accessible through **Meta-Shift-Y**.

There is no ambiguity in **Meta-Shift-Y**: when the minibuffer wants a pathname, **Meta-Shift-Y** uses the pathname ring. When the minibuffer wants a buffer name, **Meta-Shift-Y** uses the buffer name ring. When the minibuffer wants a definition name, **Meta-Shift-Y** uses that ring. Other rings of minibuffer arguments of particular kinds may be created in the future; **Meta-Shift-Y** will be the way to access all of them.

Note that the command **c-X Altmode**, which repeats previous minibuffer commands, takes arguments just like **c-Y**, and also stores its data in a history. However, this command does not really work by yanking text. There has been no change in the way **c-m-Y** is used to go back to previous minibuffer arguments or to a previous command.

To summarize, here are how the histories are accessed:

Control-Y Kill history; everywhere (including the rubout handler).

Control-Meta-Y
Input history; rubout handler.

Control-C
Control-Meta-Y
Input history; Editor and Ztop.

Meta-Shift-Y
Arg history; minibuffer.

Meta-Y Rotate yank pointer of any history.

The LISP (Edit) window or editor top level, and Ztop mode, now provide infinitely long input histories just like the one that the usual rubout handler provides. Formerly each batch of input read in a LISP (Edit) window or in Ztop mode was pushed on the kill history. Now it goes on the window's or Ztop buffer's input history instead. Use **c-m-Y** to yank the most recent element of the input history, just as you would in the rubout handler, and then use **Meta-Y** to rotate to earlier inputs if you wish.

8.3 More Rubout Handler Commands

The rubout handler now has a mark, and supports the commands **c-Space**, **c->**, **c-<**, **c-W** and **m-W**. They work about the same as the editor commands of the same names.

The rubout handler also now supports **Meta-T**.

Typing **Meta-Status** is now a way to print the rest of the input history beyond the part that Status shows you. A numeric argument specifies how many elements at the front of the input history to skip mentioning. **Control-Meta-Status** does a similar thing for the kill history, to complement the **Control-Status** command.

8.4 Sectionization Improvements

Now each form in a buffer gets its own section. This has several beneficial results.

m-X Compile Buffer Changed Sections will no longer recompile any random forms that are adjacent to functions you have edited. In fact, this command recompiles only sections containing **def**... forms.

Evaluating a random form in the buffer will no longer mark any definition as "already recompiled". Even evaluating a form that is part of a definition will no longer mark the entire definition as "already recompiled."

c-sh-C can now print the name of the function being compiled very quickly, based on the sectionization.

The section nodes for non-definition forms have names that are strings containing the file or buffer name, the function that the form invokes, and a numeric suffix to make the name unique: for example, **QFCTNS-DEFPROP-182**. You will see these section names mentioned in the output of **m-X List Sections** and other commands for listing or visiting sets of sections.

8.5 Buffer Selection History Now Per Window

Each **Zmacs** window now keeps its own history of all buffers. The **c-m-L** command, and defaulting when reading a buffer name argument, both use the selected window's history. (This is the same history that you can yank from using the **m-sh-Y** command when giving a buffer name argument.) The history's "most recent" elements are buffers that have been selected in this window, most recent first. The least recent elements are other **Zmacs** buffers that have not yet been selected by this window. The histories of different **Zmacs** windows all contain the same elements, but they may be in different orders. **c-X c-B** now displays the per-window history.

8.6 Per-Buffer Local Variables

Now you can make any special variable's value local in a specific editor or, in the case of ZMACS, in a specific buffer. For editor user option variables this can be done with a **Meta-X** comand.

zwei:make-local-variable *variable* &optional *value xvcell* Function

Makes *variable* local in the current editor, or the current buffer if this is an editor that can select various buffers (that is, ZMACS). If *value* is specified (whether **nil** or not) then *variable* is set to *value* after it is made local; otherwise it keeps its global value.

The argument *xvcell* is used in ZMACS buffer switching. If non-**nil**, it should be a closure value cell, which is used as the value cell for the local binding. value is ignored when xvcell is given.

zwei:kill-local-variable *variable* Function

Makes *variable* no longer be local in the current editor or buffer. It reverts to its global value.

The easy way to make a ZWEI user option variable (such as ***comment-column***) local is with the command **m-X Make Local Variable.** It reads a variable's pretty name (such as "Comment Column") with completion and makes that variable local. The complementary command **m-X Kill Local Variable** also exists.

m-X List Local Variables prints the names and values of all the local variables in the current editor or current buffer.

8.7 Shifted Mouse Clicks

If is now possible to use "shifted" mouse clicks to give ZMACS commands. ("shifted" means modified by one or more of the CTRL-, META-, SUPER- or HYPER- keys.) Thus it is now possible to give the **m-X Set Key** a "shifted" mouse click (like **control-Mouse-Left-1**) as the key, and to set "shifted" mouse keys in init files using **zwei:set-comtab**.

8.8 Close Parenthesis Displayed for Open Parentheses

When point is before an open parenthesis, the matching close parenthesis now blinks. If point is both before an open parenthesis and after a close parenthesis, the matching open of the preceding close parenthesis is the one that blinks.

8.9 Editor Aids for Common Lisp

There are now two new commands (available from Lisp mode) that allow easy modification of the current readtable for an editor buffer, which controls the particular syntax used for that buffer.

m-X Set Readtable

Changes the **Readtable** attribute of the current buffer, prompting for a readtable name (with completion available). A short description of the names of the standard readtables is available on 41.

To specify a readtable that doesn't already exist, you must exit with **Control-Return**, or type **Return** twice. Then you must confirm with "Yes."

You will also be asked whether to change the attribute list in the text. If you answer yes, the buffer's first line is modified to say that it should be read using the new readtable. This will affect all operations on the file, once you save the buffer.

m-X Set Common-Lisp

This commands changes whether the contents of this buffer are to be regarded as having COMMON LISP syntax, which is done by changing the readtable in effect for this buffer. The command then queries you for whether to change the attribute list in the text as well.

Besides binding the readtable for the editor buffer and the break loop, the readtable attribute also sets the quoting character (one of the two slash characters) as appropriate.

8.10 Lisp Case Changing Commands Renamed

The extended (m-X) commands for changing the alphabetic case of Lisp code have been renamed:

Old name	New name
Lisp Lowercase Region	Lowercase Lisp Code In Region
Lisp Uppercase Region	Uppercase Lisp Code In Region

As a result, typing **m-X lisp** now completes to **Lisp Mode**.

8.11 Font Handling Changes

8.11.1 Yanking and Fonts

When text is moved between buffers and files in which fonts are specified, the precise font of each character is now preserved. If you yank a character that was in font **medfnt** in the buffer where it used to be, it will be in **medfnt** after it is yanked. This may necessitate adding fonts to the font list of the buffer you are editing; if so, you will be asked whether to modify the attribute list (the -*- line) in the text as well.

This new feature applies to the commands **c-Y**, Insert File, Insert Buffer, and **c-X G**. But it only applies when fonts have been specified in the buffer you are editing (with Set Fonts or with a Fonts: attribute). Otherwise, all the yanked text gets put into the default font along with everything else in the buffer. Also, if fonts were not specified in the file or buffer that the text came from, it simply goes into font zero of the current buffer.

8.11.2 New Font Change Commands

The following **Zmacs** keys now have font change commands bound to them:

Control-Shift-J
: This is now Change Font Region, just like Control-X Control-J.

Meta-Shift-J
: This is the command Change One Font Region, which operates on all characters in the region that have a particular font, changing them to another font. It asks for the font to look for first, and then the font to change to. For example, you could specify to change all font A characters into font C.

8.12 New Meta-X Commands

Tags Search List Sections
: This command searches all the files in the currently selected tag table for a string that you specify. It does not itself move point or select a different buffer. Instead it records which sections the string is found in and then prints a list of the sections' names. You can then begin visiting the sections one by one with CONTROL-SHIFT-P.

Start Private Patch
: A private patch is one which is not installed in the system. It is not associated with any specific patchable system, and it does not get a patch version number. It is simply a file of redefinitions that you can load explicitly if you like. **load-patches** does not know about private patches.

m-X Start Private Patch starts editing a private patch. You are asked to specify the pathname of the patch file. Once you have done this, you can put text into the patch with **m-X Add Patch**, just as you can for installed patches. You finish with **m-X Finish Patch**, as usual. This saves and compiles the patch file.

You can use **m-X Start Private Patch** to resume editing a private patch you created previously. This works whether or not you had finished the patch earlier.

Add Patch Changed Sections
: The command **m-X Add Patch Changed Sections** finds all sections you have changed the text of, in all buffers, and asks you for each one whether to do **m-X Add Patch** of its text. But sections that have been **Add Patch**ed already since their last modification are excluded. If you answer the question **P**, then all the rest of the changed sections in the same buffer are patched without further question.

Add Patch Buffer Changed Sections
: This is similar but considers only the current buffer's sections.

8.13 CONTROL-X 4 J Jumps to Saved Point in Other Window

If you save a location in a register with **c-X S** *register*, you can jump to it again with **c-X J** *register*. Now you can also select the other window (in or entering two-window mode) and jump to the saved location in that window, by using **c-X 4 J** *register*.

8.14 Minor Command Changes

- The command **Control-Shift-D** now prints the full documentation of the function which point is inside a call to. **Control-Shift-D** is thus analogous to **Control-Shift-A**. **Meta-Shift-D** is still available if you wish to specify the function to be documented.
- The default version for the command **m-X Source Compare** is now **:newest** for both the first and the second input file.
- The **m-X View File** command now displays the file with its correct fonts if the file specifies fonts in its attribute list.
- The **Meta-X** commands Copy File, Rename File, Delete File, and Undelete File have been changed to do prompting and querying in a new way.

  If the pathname specified has no wildcards, no prompting or querying is done. The operation is just performed.

  If there is a wildcard, then a list of the files that match it is printed all at once. You are then asked to confirm with **Y** or **N**. If you say **Y**, then all the files are operated on forthwith.

8.15 Commenting a Region

This command puts a comment starter (the value of **zwei:*comment-begin***) in front of each line starting in the region, except for blank lines. With numeric argument, it removes precisely the value of **zwei:*comment-begin*** (a single semicolon in LISP mode) from each line in the region that starts with one.

You can use **c-X c-;** to comment out the lines of the region before recompiling a function. Later, use **c-U c-X c-;** to remove the commenting thus made. Only a single semicolon is removed, so any lines that were comments before commenting out the region remain comments after un-commenting the region.

8.16 Dired

Dired now displays files that are really deleted on disk with a lower case **d** in the first column. Files whose deletion has been requested but not done are displayed with a capital **D**. If you request undeletion of an actually deleted file, the file is displayed with a capital **U**, but such operations as printing, editing, or applying a function to the file are not allowed since the file is really still deleted.

When the current buffer is a **Dired** or **BDired** buffer and you issue a command that reads a filename, the default filename is now the file whose line you are pointing at.

A new command to edit the superior directory of the current buffer's directory can be found on the **<** key in **Dired**.

9. Site File Changes

9.1 Logical Host Definitions Kept in the SITE directory

The definition (and translations) for logical pathname hosts can now be kept in the site directory. Refer to page 31 for a discussion of this new feature of logical hosts. The **SYS** host, by convention, is now defined in the file **SYS: SITE; SYS TRANSLATIONS**. Use of **:sys-host-translation-alist** and variables to hold express the **SYS** host translations is considered obsolete.

9.2 Specification of File Servers

Due to a change in the internals of the pathname system, the name of the site option that lists file server hosts is now called **:file-server-hosts**. This is just like **:chaos-file-server-hosts**, except that it contains hosts that the machine knows about by default. If someone tries to reference a host that is a file server, but which did not appear on the list, he will still get the desired behavior, the host object will dynamically be added to the pathname host list, if need be. T~~hus,~~ **SYS: SITE; SITE LISP >** no longer needs to be changed merely to add new file servers – as long as they appear in the host table, that should be sufficient.

The hosts do not have to be Chaosnet hosts. Currently, Chaosnet access is the only kind of remote access. However, the name change is anticipation of access by other protocols, such as TCP FTP. Other access flavors are for Local File access and local LMFILE access.

It is now possible is specify the default device of a host by using the site option **:host-default-device-alist**, an alist of host names and device names (with the colon). This option is effective for Twenex and VMS hosts. An example of the use of the option:

```
(:host-default-device-alist '(("OZ" . "OZ")))
```

Here, we are overriding the default name **PS**. This option is especially useful for VMS hosts, since the "default" device is some logical name that can differ from system to system.

If, for example, a Twenex host is configured for a non-**PS** primary structure name, this option should be used, to eliminate some strange interactions that can happen when the truenames of files are compared against supplied names.

One user-level change has occured because of this. Suppose one supplies the file name

```
SRC:<L.IO.FILE>ACCESS.LISP
```

which is intended to name a file on the Twenex host OZ, to a program, and the default host is OZ. The pathname parsing system will try to see if SRC: is a file host, and that may entail going over the network to contact a host table server to see if SRC is a host. Still, when it determines that SRC is not a host, it will recognise it as a device instead, and the pathname will become **OZ:SRC:<L.IO.FILE>ACCESS.LISP**. All of this checking is a result of the unfortunate choice, made for historical reasons, of colon being the delimiter for both host and devices. The rule has now been changed so that the first colon delimits the device. Therefore, when supplying a pathname with an explicit device, but defaulting the host, a colon must also be supplied before the device, like

```
:SRC:<L.IO.FILE>ACCESS.LISP
```

9.3 New site option :STANDALONE

If the Lisp Machine is just by itself, the option should be supplied with value **t**. This will cause the Lisp Machine to not to try to use the Chaosnet for getting the time, for one thing. On the Lambda, the time will obtained from the SDU's clock. On the CADR, the time will be obtained from the user.

Concept Index

Lisp Index

Installation Packet
LMI Release 2.0–5/01/85

24-0100323-0002

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Table of Contents

2. Backing Up an Existing UNIX System

2.1 The Root Image

Installing the UNIX root image will write over all UNIX files in "/" and "/sdu". (UNIX files in /usr will not be changed by the root change.) Therefore, any valuable files here should be backed up before the update and restored after it. There are limitations on this, however, and a system operator at your site will have to be involved with this procedure.

For instance, the /etc/rc file has been changed to work with Release 2.0. Because of this, the system will work improperly if you merely restore your previous /etc/rc file. A UNIX system operator at your site should merge your changes with the new release file in order to maintain system release integrity.

In a standard UNIX environment, at least the following files will be customized: **/etc/passwd**, **/etc/ttytype**, **/etc/ttys**, **/etc/myhostname**, and **/etc/hostbin**. These files contain user ID, terminal, and site file information. Even if nothing else is customized at your site, you should back up these customized files and restore them after installing the new root.

The following procedure backs up the files **/etc/passwd**, **/etc/ttytype**, **/etc/ttys**, and **/etc/myhostname**; this procedure can be extended to other important, customized files.

Procedure:

Before installing the software update, mount a tape and type in UNIX

```
cd /etc
tar cvfb /dev/rmt0 20 passwd ttytype ttys myhostname hostbin
```

After the installation, you can restore these files to replace the standard files distributed with the release by typing in UNIX

```
cd /etc
tar xvpf /dev/rmt0
```

Note: Under no circumstances should you back up the "/dev" directory; attempting to do so will inappropriately open the "devices" needed to run the system (tape, disk, memory, terminal, etc.), and in doing so may crash the system.

2.2 Installation

The following is a command synopsis of the installation procedure.

> NOTE: If Monitor Version 8 is in use on the SDU, set up the CMOS RAM using the Release 2.0 diagnostic tape. Type at the SDU:
>
> ```
> /tar/setup eagle
> init
> ```

Then, for Monitor Version 7 or Monitor Version 8:

1. Back up customized files
2. Install the Release 2.0 Root Image at the SDU:

   ```
   copy tape $disk
   ```

   This will take about 20 minutes.

   Note that there are different versions of the root for systems with UNIX and systems without. The root with UNIX is labeled "Release 2.0 Root with UNIX." The non-UNIX version is labeled "Release 2.0 non-UNIX Root."
3. Use the SDU program **config** to configure the system. Refer to Chapter 7, "The config Program".
4. Install the LISP microcode and band using the SDU **load** program. Refer to Chapter 8, "The load Program".

   Be sure not to load new software over any valuable microcode or band.

   Loading will take about 30 minutes.
5. Use the load program to select the newly installed microcode and band, then boot the system using the SDU "superboot" program.
6. Once LISP is booted, install the Release 2.0 LISP Sources on the slot 0 Lambda processor by typing in LISP

   ```
   (fs:restore-magtape :query nil)
   ```

   This will take about one and one-half hours, depending on the amount of memory in the system.

Cautionary Preface

It is always a good idea to back up your files (LISP and UNIX) on the system before a software update.

Installing UNIX usr files will write over any files in the UNIX "/usr" directory that have the same name as files in the new usr file tape. This is because UNIX does not have file version numbers.

Be sure you back up any modified UNIX distribution files. This includes, as a minimum, your own user directories and the site information directory "/usr/lib/chaos/tables".

1. System Notes

This is a compilation of items that will affect booting and system operation. General LISP and UNIX changes are addressed separately throughout the release notes.

- The LISP Windowmaker user interface software and documentation has been incorporated into the standard release software.
- The LISP function **(si:with-open-device)** has been replaced by the function **(with-open-file)**. With a printer on SDU Port B, for example, the following LISP command now would be used to print the line **"This is a test"**:

```
(with-open-file
     (stream "sdu-serial-b:" :direction :output)
     (format stream "~% This is a test ~%"))
```

- The Microcompiler option is available for use under Release 2.0.
- The LM-PROLOG option is available for use under Release 2.0. This LM-PROLOG does have microcode support.
- The INTERLISP option is not yet available for use under Release 2.0, but will be available shortly.
- The SDU commands **uboot**, **lboot** and **qboot** are now obsolete. Only **superboot -a** and **superboot** should be used to boot a Release 2.0 system.
- The cache option is turned off by default by the config program. Enabling the cache has been seen to cause unreliable machine operations; we recommend that the cache not be enabled until this is corrected.
- Full and incremental garbage collection as implemented in Release 2.0 is an improvement over Release 1.2 but has been seen to act unreliably. A new garbage collection scheme is in test, and will be incorporated into Release 2.1.
- In most cases, the Release 2.0 root can support both the Release 2.0 and the Release 1.2 LISP bands. The following combination is the only one that does NOT work:

  If Release 2.0 is booted on both sides of a 2×2, and then the slot 4 processor is booted via LISP on Release 1.2, the machine crashes.

  If you need to boot the two processors on different bands, you can specify this from the SDU using the **lambda** *load* option in the config program rather than rebooting in LISP.
- If the slot 0 processor of a 2×2 crashes, the slot 4 processor will not be able to access the network. This is because the slot 0 processor "owns" the network.

  If you disable the slot 0 processor, the slot 4 processor assumes the name of the slot 0 processor (e.g., **LAMA**) and becomes the network owner. However, it still accesses the filesystem on which it had been booted. This is because the filesystem associated with each processor is set by the **lambda** *file* option in the config program.

7. If the system has UNIX, boot UNIX multiuser and install the Release 2.0 UNIX "usr" files tape by typing in multiuser UNIX

```
cd /
tar xvpf /dev/rmt0
```

   This will take about 20 minutes.

8. From here, you should update the site files. Refer to Chapter 10, "Site File Installation Notes".

Additional Information

If you prefer, you can install the Release 2.0 microcode, band and sources on the machine before you install the new root image. In this case, install the microcode and band in LISP instead of at the SDU, using the LISP command **(fs:restore-magtape)**. After that, while still booted on Release 1.2, install the Release 2.0 LISP Sources in LISP with

```
(fs:restore-magtape ':query nil)
```

After this LISP update is completed, proceed with steps 1-3 as described above, and with step 8 if the system has UNIX.

3. Diagnostic Tape

The *LMI Release 2.0 Diagnostic* tape differs from the Release 1.2 diag tape in the following ways:

- **setup** now supports both the new SDU Monitor Version 8 and the old SDU Monitor Version 7. Note the syntax change and description in Chapter 5, "The Setup Program".

- **setup.7** is the Release 1.2 version of **setup**, and is obsolete. It is included for Field Service use only.

- **new-2181** is a new version of the **2181** disk diagnostic program. It should be used by Field Service personnel only in cases where there are more than 20 bad tracks on the disk. It starts mapping tracks at cylinder 830, which is non-standard; hence its use should be recorded in the system site log.

- **2181** is the Release 1.2 version of the disk diagnostic program. It should be used for disk diagnostics and, by Field Service personnel, for reformatting when required.

- **lam** runs a diagnostic on the LISP processor. This program should be used by Field Service personnel only.

4. SDU Monitor Version 8

Beginning with LMI Release 2.0, the SDU monitor version has been changed. Where the SDU used to print out **Monitor Version 7**, it will now print **Monitor Version 8**.

The SDU monitor is the bootstrap program resident in PROM on the SDU. It controls the meanings of the rotary switch positions on the rear of the machine, and performs other functions that enable the SDU to communicate with a terminal, disk, and tape.

If the CMOS RAM loses its information while Monitor Version 8 is in use, the red SETUP light will come on in place of the green RUN light, as it did with Version 7. However, several monitor version differences will be observable by the user:

- The Z29 console (**ttya**) will remain at 9600 baud even if the CMOS RAM is corrupted; previously, it defaulted to 300 baud.
- Rotary switch position 1 defaults to SDU Port A as the 9600 baud SDU console.
- Rotary switch position 3 defaults to the high-resolution monitor as the SDU console.
- Rotary switch position 0 is hardwired to use SDU Port A as the 1200 baud SDU console.
- A corrupted CMOS RAM will cause the message **CMOS RAM Invalid** to appear on the SDU console in place of the **Monitor Version 8** message.

5. The Setup Program

The setup program is used to load the SDU's CMOS RAM with the necessary information about accessing the disk and the console. The setup comand line and setup time both have changed since Release 1.2.

If you have a Monitor Version 8 SDU and you do not have a terminal on SDU Port A, you must use the following procedure if the CMOS RAM becomes invalid (red SETUP light is on):

1. Turn rotary switch on rear panel to position 3.
2. Press INIT button on rear panel.
3. Proceed with **/tar/setup eagle** as given below.
4. Turn rotary switch on rear panel back to position 1.
5. Press INIT button on rear panel.

After this, the green RUN light will appear and the system will function normally.

To load the CMOS RAM if you have a terminal on SDU Port B or if the CMOS RAM is not invalid, mount the LMI Release 2.0 diagnostic tape and type at the SDU the command:

```
/tar/setup eagle
```

The line **Setup Version 19** will print on the screen, followed by an SDU prompt. This will take about 20 seconds. Type

```
init
```

and when the prompt reappears, the new setup will have taken effect.

If the SDU has Monitor Version 7 PROMs, this setup will enable the Z29 (SDU port A) as the system console. If the SDU has Monitor Version 8, this setup will default to using the high-resolution monitor as the system console.

If you have a Monitor Version 8 SDU but still want to use SDU Port A as the system console, setup the SDU from the tape using the command:

```
/tar/setup eagle ttya
```

NOTE: If the system has two high-resolution monitors, the one associated with the slot 8 video card will be the console.

6. Standard Configurations/Monitor Version 8

The user has more options for tailoring the console environment in this release than in Release 1.2. The config program allows several parameters to be varied according to processor configuration, terminal attachments, and personal preference.

In general, if UNIX is being used solely as an adjunct to LISP, **sharetty** is the console and no UNIX terminal is connected to SDU Port A. If used by a UNIX programmer working independently, UNIX is configured with **ttya** as the console. LISP is always configured with the default (high resolution) console, unless the system administrator wants to limit system console access.

Although other combinations are workable, the configurations below are those programmers so far have found the most convenient.

Lambda SDU console: High-resolution monitor

Lambda/Plus
SDU console: High-resolution monitor
UNIX console: ttya (Z29-type terminal on SDU Port A)

Lambda/2×2
SDU console: High-resolution monitor associated with slot 0

Lambda/2×2/Plus
SDU console: High-resolution monitor associated with slot 0
UNIX console: ttya

The UNIX console is set by the user the first time config is run, and later can be changed via config's UNIX command **console**.

7. The config Program

Config version 131 is the current version of config in Release 2.0.

As with config 89, you can type partition names in either upper or lower case, and they are forced to upper case.

Config 131 is compatible with the **load** program in assigning page and file partition names. Both programs now default to FILE and PAGE for the slot 0 Lambda processor, FIL1 and PAG1 for the slot 4 Lambda processor.

Config 131 allocates 1MB to UNIX by default. This fixes the config 89 bug that allocated too little memory to UNIX.

To configure your system, type at the SDU

config

The first time config is run after a new root image is installed in a machine that has UNIX, config will prompt for the UNIX boot console. Type

ttya

to use a Z29-type terminal as the boot console, or

sharetty

to use the high resolution monitor as the boot console. If **sharetty** is used as the UNIX console, you will have to halt the slot 0 LISP processor anytime you need to boot UNIX.

After this, config will print the configuration and ask if you want to change anything. In most cases, the default configuration will be appropriate. When config prompts,

Do you want to change anything? (y/n)

respond by typing "**n**" followed by a RETURN. The specified configuration will be saved onto the disk. This does not need to be done every time you power up the system; you do not need to run config again unless you need to change one of these basic configuration options.

Additional Information

The config program, run from the SDU, locates boards, allocates memory, sets the system console device and several other configuration options.

This config version differs from the config version 89 of Release 1.2, both cosmetically and internally.

The first time config is run after a new root image is installed, config will say what each slot has become before it prompts for changes. In subsequent runs, config will print out the current configuration.

Example

The following is a transcript of a typical config run. *Slanted text* indicates user input.

>> *config*

```
using 64K in slot 10
config version 131

Slot 0 has lambda
Slot 4 has lambda (disabled)
Slot 8 has vcmem
Slot 9 has vcmem
slot 10 has two-meg
slot 11 has has 68000 (disabled)
slot 12 has two-meg
slot 13 has half-meg
slot 14 has half-meg
slot 15 has sdu

Total memory = 5120K bytes

Lambda V4.0 in slot 0
     vcmem in slot 8
               (pool room)
     has processor switches 013600000000
        parity-enable byte 00 (all off)
        tram file /disk/lambda/c.tram-n-n, boot speed 1-1
        microcode band <default>, load <default>, page PAGE,
        file FILE
        scan-line-size 32.
```

```
          5030K bytes memory

     System parameters:
          user-defined shared area is 20K
          sdu code area is 64K
          reserved multibus space is 8K
                    (from 0xEE000 to 0xEFFFF)
          system-configuration shared area is 6K
     Do you want to change anything? (y/n) n

     Writing config file "/disk/lambda/shr-config.1"
     Initializing SDU ...
```

The following example illustrates how a change can be made after the question "Do you want to change anything?" The prompts used by config are **"cmd:"** at top level, **"lambda cmd:"** in "lambda" mode, and **"unix cmd:"** in "unix" mode.

```
     Do you want to change anything? (y/n) y
     Type "?" for instructions.

     cmd: ?
     The usual procedure is to disable any boards that you don't
     want to use, then change any of the LAMBDA or unix options,
     and then write the file.  When a number is asked for,
     <return> always defaults or aborts. Control-C aborts and
     exits without changing the file.

     Commands are:
          dflmem    -    reset memory allocation to default
          disable   -    turn off a board
          enable    -    turn on a board
          lambda    -    change lambda options
          loc       -    edit console location strings
          mem       -    change memory allocation
          print     -    print current config info
          reset     -    reset config file to defaults
          system    -    changes system-wide and sdu options
          unix      -    change unix options
          write     -    write new config file and exit
          x         -    exit
          ?         -    instructions

     cmd: enable
     Enable board in which slot? 11
     cmd: unix
     unix cmd: console
```

micro option for microcode band — load option to boot processors from separate bands

```
The console devices are: ttya sharetty
Enter console device: ttya
unix cmd: x
cmd: loc
         SDU port A
          pool room
Type new string, or <return> to leave unchanged.
pool room terminal 3
cmd: write

Writing config file "/disk/lambda/shr-config.1"
Initializing SDU ...
```

Typing "?" at a sub-level command prompt ("lambda cmd:" or "unix cmd:", for example) will give you help on the commands for that level.

8. The load Program

The load program can load microcode and band tapes, copy from one disk partition to another, edit disk label comments, and write out microcode and bands from the disk to tape. It reads the type of disk out of the CMOS RAM. It can deal with 1/2" tape, 1/4" tape, 474MB disk and 169MB disk.

8.1 How to Invoke the load Program

Invoke the load program from the SDU by typing

```
load
```

It will print a message that it is "load version 107", and will return a **disk loader>** prompt. To get a list of all the possible commands, type

disk loader> *help*

The first time you try to access the disk in the load program, it will look in the CMOS RAM to find out what type of drive you are using. Then this information will be printed on your screen. For example, if the first thing you do is print the disk label on a Lambda/PLUS, it will display

```
842 cylinders, 20 heads, 25 sectors per track
disk drive is Fujitsu Eagle
```

8.2 Functions of the load Program

The following is a description of the functions of the program.

printlabel
This prints the disk label. Note that this disk label differs from the old one. "FILE" is the name of the first file partition and "FIL1" is the name of the second (formerly "FIL1" and "FIL2"). "PAGE" is the name of the first page partition and "PAG1" is the name of the second (formerly "PAG1" and "PAG2").

initlabel This initializes the disk label. Note that it writes a different disk label than the old load program did. When you type *initlabel* at the **disk loader>** prompt, you will be asked to specify whether you have a 2X2 configuration. Appropriately, a 2X2 disk label is written if you respond **y**, and a single Lambda label is written if you respond **n**.

setmload This sets the current microcode to whatever you specify. You may type the partition name in upper or lower case. Be sure to type **lmc***n* instead of just *n*, or the machine will not boot.

Example
(User responses are given in *slanted text*, for clarity:)

```
disk loader> setmload lmc2
Setting current microcode to LMC2
```

setband This sets the current band to whatever you specify. You may type the partition name in upper or lower case. Be sure to type **lod***n* instead of just *n*, or the machine will not boot. **setband** does not check to see whether you have selected the microcode that corresponds to the band you specify; if you are unsure of which microcode to select for the band you want, use the **prefmic** command and then the **setmload** command.

prefmic When you specify a band partition, this tells you which microcode it goes with. This will make sure you can change bands and still have a compatible microcode. Previously, this could be done only in LISP.

Example

```
disk loader> prefmic
select a partition lod1
microcode version 152
disk loader>
```

ttod This command (short for "tape to disk") is the equivalent of the old **load** command. Only the name has changed. Known, minor bug: when it gets to the end of the tape, it will print the message **Read error in header, probably end of tape**.

Example

```
disk loader> ttod
What kind of tape drive are you using?
(1 = 1/2", 2 = 1/4") 1
Partition ULAMBDA 768; size 500
Type partition name to load, or return to skip this file
or type 'exit' to quit loading lmc1
Copying xxx bytes
........................
copy done
0 errors
disk loader>
```

dtot This command ("disk to tape") is used to write out a partition to tape (for making a backup, for example). When prompted, you supply the tape drive type and partition name for which partition you want written onto the tape, and later specify whether you want to copy another partition to the tape. Previously, this could be done only by the LISP **copy-disk-partition** command.

Example

```
disk loader> dtot
What kind of tape drive are you using?
(1 = 1/2", 2 = 1/4") 1
select a partition lod1
Copying xxx bytes
.........................
copy done
do you wish to copy another partition? n
disk loader>
```

dtod This ("disk to disk") can be used to copy one disk partition to another or to compare two disk partitions. When prompted, you specify whether to do a copy or a compare, and which partitions to use. . Previously, this could be done only by the LISP **copy-disk-partition** command.

Known, non-fatal bug: although the first time you specify a nonexistent partition name (e.g. "lmc9"), it will gracefully tell you that it couldn't find the partition, the second time it may break and give you a series of "unknown command" messages. In this case, you will have to exit the program by typing **CTRL-C**, then typing **init**.

Example

```
disk loader> dtod
copy or compare (1 to copy, 2 to compare) 1
select a partition lod1
select a partition lod2
copying xxx bytes
......
copy done
disk loader> dtod
copy or compare (1 to copy, 2 to compare) 2
select a partition lod1
select a partition lod2
comparing "102.117" to "102.117"
.......
compare done
0 errors
```

```
disk loader> dtod
copy or compare (1 to copy, 2 to compare) 2
select a partition lod1
select a partition lod3
comparing "102.117" to "102.117"
...............
..........compare error at location xx
...............
compare done
1 errors
disk loader>
```

change-comment
Use this to edit the disk label comment fields, to comment out a bad copy or to customize. You must specify the partition, and then type in the new comment when prompted. Remember that the comment cannot exceed 16 characters or it will be truncated. Previously, this could be done only with the LISP `edit-disk-label` command.

Example
This example shows how to name LOD3 "Experimental".

```
disk loader> change-comment
select a partition: lod3
change comment for partition LOD3 to:
Experimental
disk loader>
```

size When you specify a partition, this tells the actual measured size of the band in that partition, rather than the allocated length as written in the disk label.

Example

```
disk loader> size
select a partition lod4
partition LOD4 has physical size 30000
measured size 18211
```

tapetype Allows you to specify more than once whether you are using 1/2" tape or 1/4" tape. This is useful only if you are making a backup and want to make both 1/2" tape copies and 1/4" tape copies during the same load session.

exit This is the correct way to leave the load program when you are done. (Previously, you had to type **CTRL-C**, then type **init**, before doing anything else.) When you type *exit* at the **disk loader>** prompt, the SETUP and ATTN lights on the front panel will come on because the machine is automatically doing an "init." In the usual amount of time, the two red lights will go off and the green RUN light will appear.

At this time, you can run any program.

9. Boot Procedure

The Release 2.0 boot software changes the boot procedure considerably from that used in Release 1.2. This procedure holds for both SDU Monitor Version 7 and the new monitor version, SDU Monitor Version 8.

SDU Monitor Version 8, with **config** and **superboot**, allows the high-resolution monitor to be used as the system console for the SDU, LISP and UNIX. This completely eliminates the need for a Z29-type terminal on SDU Port A. If SDU Monitor Version 7 is used, SDU Port A is still needed as the system console.

The instructions below outline the procedures common to both Monitor Versions 7 and 8. Where procedures for the two versions diverge, the differences are noted. In either monitor version, the phrases *type at the SDU* and *type on the system console* are used interchangeably to refer to commands typed at the system boot console.

To boot a configured machine, type on the system console

```
superboot -a
```

This program will print its version number, clear the screen, and boot all processors in the system without further prompting. Since this is self-explanatory, the rest of the discussion here deals with **superboot** behavior without the **-a** option, and with returning to the boot menu after booting.

If you type at the SDU console the command

```
superboot
```

(*without* the "**-a**"), a version number will print, the screen will clear, and a boot menu will appear. One such menu will print on each processor's console, and the **boot** command on each refers to that processor only.

The high-resolution monitor can be used to return to the boot menu at any time by typing **ctrl-meta-ctrl-meta-⟨LINE⟩**. This will halt the LISP processor associated with that console, and will exit to the boot menu.

The specific behavior of this superboot differs according to the processor configuration and the system and also according to console options specified when using **config**. Because of this, behavior will be treated here in specific modules, before any integrated documentation is attempted.

9.1 LISP Console Boot Menu

Figure 1 shows the menu as it appears on the LISP console. It includes processor and console information, commands, and a prompt.

Typing commands at the command prompt has the following results.

```
This processor:
     LMI LAMBDA V4.0 in slot 0
     (system console)
          console is vcmem in slot 8
          location:  pool room terminal 3

Commands are:
          boot                cold-boot lambda
          warm                warm-boot lambda
          why                 diagnose reason for halt
          unix                connect to unix console
          reset-68000         reset 68000
          herald              print sign-on message
          INIT                reset SDU and all processors

Command:
```

Figure 1. Typical LISP Console Boot Menu

boot Prints **"Cold booting Lambda"** and boots the LISP processor associated with that console. This is equivalent to **ctrl-meta-ctrl-meta-**⟨RUB OUT⟩ on that LISP processor.

warm Prints the console location and then tries to reboot that LISP processor, saying **"Warm-booting lambda: If this works, save your files and cold boot"**. This command is equivalent to **ctrl-meta-ctrl-meta-**⟨RETURN⟩ on that LISP processor.

why Fills the window to the bottom with debug information, and then wraps to the top of the screen and continues printing. The information takes less than a screenful to display, so it will not overwrite itself.

unix Connects to the UNIX console and boot menu on the high-resolution monitor if **sharetty** was set in **config** as the UNIX console. Otherwise, it says **"You can't connect to unix from here"**. If **sharetty** is used as the UNIX console, and **superboot** is used instead of **superboot-a** you should boot UNIX before booting LISP or else you will have to halt the slot 0 LISP processor using **ctrl-meta-ctrl-meta-**⟨LINE⟩ in order to boot UNIX.

reset-68000
Says nothing on the console, but kills and then reboots UNIX if UNIX has

been running during this session. If not, it is a no-op. This is to be used in case UNIX wedges in some way that previously would have required pressing ⟨RESET⟩ to recover. If you just want to bring down a working UNIX, it is better to exit UNIX "gracefully" using the usual UNIX commands at the 68000 CPU.

herald Redisplays the "**This processor**" message.

INIT (Note that you must type this in capital letters.) Says "**Initializing SDU ...**" and does so. This really does reset all processors, no matter who is doing what on them; it should, therefore, be treated with respect. All users should be prepared for the machine to be halted before this command is used. **INIT** may create inconsistencies in the filesystems of the various processors if it is used during file operations.

? Redisplays the command list. (This is not on the menu.)

9.2 UNIX Console Boot Menu

Figure 2 shows the menu as it appears on the UNIX console. It includes UNIX processor, LISP processor, and console information, commands, and a prompt.

```
This processor:
     68000 in slot 11
          console is ttya

Other processors:
     LMI Lambda V4.0 in slot 0
     (system console)
          console is vcmem in slot 8
          location:  pool room terminal 3

Commands are:
          boot                     boot unix
          herald                   print sign-on message
          INIT                     reset sdu and ALL processors

Command:
```

Figure 2. Typical UNIX Console Boot Menu

Figure 2 shows a typical UNIX console boot menu. It is quite similar to the LISP menu. As with the LISP console boot menu, **INIT** resets all processors and **herald** prints the "**This processor**" information.

Typing **boot** at the UNIX menu prints the messages

```
Booting 68000 in slot 11 with /disk/unix.new
reading file '/disk/unix.new'
starting 68000
```

and then prints the singleuser UNIX herald and prompt.

If **ttya** was specified as the UNIX console during the config program, each LISP and UNIX processor will have its own boot console and menu and therefore booting is completely independent. If **ttya** is the UNIX console, **superboot -a** is the preferred boot method.

If **sharetty** was specified as the UNIX console during the config program, UNIX will share a boot console with the slot 0 LISP processor. When using this configuration, you might prefer using **superboot** instead of **superboot -a** and then booting UNIX multiuser before booting LISP. If you use **superboot -a**, LISP will come up while UNIX is still up only singleuser; to finish bringing up UNIX, you will have to type ⟨SYSTEM⟩ **u** on the **sharetty** console after LISP is running, then type **CTRL-D** in the UNIX screen.

9.3 Boot Sequence on Lambda/PLUS with superboot -a

Here is the boot sequence on a Lambda/PLUS using **superboot -a** and **ttya** as the UNIX console.

Type at the SDU

```
superboot -a
```

When you get the singleuser UNIX prompt on the Z29, type **CTRL-D**. At the end of this process, all processors will be booted completely.

Here is the boot sequence on a Lambda Plus using **superboot -a** and **sharetty** as the UNIX console. Type at the SDU

```
superboot -a
```

The machine will print some information. Type at the booted slot 0 Lambda console the following. (Your input is indicated by *slanted text*. This is an edited session; most of the machine response is indicated by ellipsis dots. Note the change of prompts, **>>** to **#**.)

```
        ⟨SYSTEM⟩ u
        .
        .
#       CTRL-D
        .
        .
        :login:
        .
        .
        ⟨SYSTEM⟩ l
```

9.4 Boot Sequence on Lambda/PLUS—superboot with sharetty

The boot sequence on a Lambda Plus using **superboot** with **sharetty** as the UNIX console is:

```
>>      superboot          ;;;Type this at the SDU
        .
        .
        Command: unix     ;;;Type this at the slot 0 Lambda console
        .
        .
        Command: boot
        .
        .
#       CTRL-D
        .
        .
        :login:
        CTRL-META-CTRL-META-⟨LINE⟩
        .
        .
        Command: boot
```

At the end of this procedure, both LISP and UNIX will be booted.

A second LISP processor does not affect this sequence; since it is not sharing a console, it can simply be booted via **boot** at its boot menu command prompt at any time.

As always, since booting requires heavy disk activity, multiple processors boot more slowly than a single processor.

10. Site File Installation Notes

10.1 Introduction

There are a number of differences between site files in Release 1.2 and Release 2.0. Some of the syntax has changed (particularly in **SITE.LISP**). A new file, **SYS.TRANSLATIONS**, has been added.

To simplify the process of updating and editing site files, we have included in this release a dedicated site file editor, invoked in LISP by

```
(sited)
```

Updating site files for Release 2.0 is a straightforward procedure.

- First the old site information is read into the "sited" editor. The site information is checked for consistency and completeness, then is written into the Release 2.0 site directory "**QL.CUSTOMER-SITE;**".
- The new site files are read into a Release 2.0 load band, and the band is saved.
- This band can then be copied to the other Lambdas on the network, so that the site information on all the systems is consistent.

The following procedure assumes that you have loaded the Release 2.0 load and microcode bands onto the SYS HOST (i.e., the machine with the source and site files for Release 1.2). You will need to know the "name" of the SYS HOST and the directory where the Release 1.2 site files can be found (usually "**RELEASE-1.CUSTOMER-SITE;**"). You will also need to find a spare load band which can be used for saving the updated Release 2.0 load band.

10.2 Read Release 1.2 Site Files:

If your SYS HOST is not currently running the Release 2.0 software, proceed with steps 1-3:

1. Log in to the SYS HOST:

   ```
   (login 'lispm t)
   ```

2. Set the current load and microcode bands to the Release 2.0 software. If this software has been loaded into "LOD3", for example, you should enter:

```
(set-current-band "LOD3")
```

(If you are prompted to change the microcode band as well, respond with "y".)

3. Cold boot your machine by pressing **CTRL-META-CTRL-META-**⟨RUB OUT⟩.

   At this point your SYS HOST should be running Release 2.0.

4. Once your SYS HOST is running the Release 2.0 software, log in:

```
(login 'lispm t)
```

5. Enter the following, substituting the name of your SYS HOST for "**george**":

```
(site:set-sys-host-for-sited "george")
```

   (The name you supply here will be used in **SYS.TRANSLATIONS** as the value for ":physical-host".)

6. Enter the following:

```
(si:set-sys-host "lm" nil nil "ql.customer-site;")
```

   The logical pathnames

   SYS:SITE;SITE and
   SYS:CHAOS;HOSTS TEXT

   will be translated into physical pathnames

   LM:QL.CUSTOMER-SITE;SITE.LISP and
   LM:QL.CUSTOMER-SITE;HOSTS.TEXT

   respectively.

7. If the Release 2.0 source files have not been loaded, you will need to create a directory for the new site files. (If the source files already exist, this command will have no effect.) Type the command

```
(fs:create-directory "lm:ql.customer-site;")
```

8. Start the sited editor:

```
(sited)
```

9. Read the Release 1.2 site information into the editor:

   SITED command> *readfiles*

   You will be prompted for the name of the site file directory:

   Directory: *lm:release-1.customer-site;*

   Respond with "y" when asked whether you wish to proceed. If you have renamed the site directory, substitute the name you supplied in place of "**customer-site;**".

   Disregard any complaints about the absence of SYS.TRANSLATIONS among the Release 1.2 site files.

10.3 Verify, Save New Site Information

1. Use the site editor to verify the Release 1.2 site information.

   Follow this procedure. (Your input is given in *slanted text.*)

   SITED command> *checkinfo*

   It is quite likely that the site editor will discover omissions in your Release 1.2 site files. For example:

   ```
   The PRETTY-NAME property for OURSITE-
   LAMBDA-C doesn't have any value
   specified for it.

   Should it be set to the default value
   of "Oursite Lambda C" ? (y or n)
   ```

   You should always respond with "y" at this stage. If you wish to use non-default values, you may specify them later. (See the site editor documentation for further details.)

   After the site information has been determined to be complete and correct, the following will be displayed:

   ```
   No errors found in the information.
   To generate new site files, use the
   WRITEFILES command.
   ```

2. Copy the site file information into the Release 2.0 site directory by entering:

   SITED command> *writefiles*

   You will be prompted for the name of the directory where you wish to write the new site information:

   Directory: *sys:site;*

   The site editor will generate the four site files (**SYS.TRANSLATIONS**, **SITE.LISP**, **LMLOCS.LISP**, and **HOSTS.TEXT**) and write them into the specified directory.
3. Quit from the site editor:

   SITED command> *quit*

10.4 Compile Site Files; Save in Load Band

1. Compile the new site files:

```
(make-system 'site :compile :noload
  :no-reload-system-declaration)
```

2. Cold boot the machine by pressing CTRL-META-CTRL-META-RUB OUT.
3. Log in.

```
(login 'lispm t)
```

4. Enter the following:

```
(si:set-sys-host "lm" nil nil "ql.customer-site;")
```

   This tells **(update-site-configuration-info)**, below, where to find the site files.
5. Bring site information into the current Lisp environment:

```
(update-site-configuration-info)
```

6. Save the current world into a spare band. If you do not mind clobbering "LOD4", for example, you enter:

```
(disk-save "LOD4")
```

You will be prompted for comments to be used by the **print-herald** and **print-disk-label** commands. The current Lisp world will be saved in the specified band, then the machine will be re-booted on the newly saved band.

7. Set the current band to this band by entering, for example:

```
(set-current-band "LOD4")
```

10.5 Copy New Band to Other Machines on the Chaosnet

1. Log onto the machine that you want to receive the newly saved band:

```
(login 'lispm t)
```

2. Locate a spare band on the machine:

```
(print-disk-label)
```

3. Receive the new band over the Chaosnet. (In this example the SYS HOST is "**george**", the new band was originally saved into "LOD4", and the local band into which the new band will be copied is "LOD2"):

```
(si:receive-band "george" "LOD4" "LOD2")
```

4. Repeat this process for all machines at your site.

11. UNIX

Once UNIX comes up singleuser via the superboot UNIX **boot** command, it can be brought up multiuser using **CTRL-D** as usual. If **sharetty** was specified in config as the UNIX console, this work will be done on the high-resolution monitor instead of on the SDU Port A terminal.

When you get the singleuser prompt (**#**), type **CTRL-D** to bring up multiuser UNIX. The UNIX file system consistency check **(/etc/fsck)** program will run, automatically correcting inconsistencies. This differs from the previous release; in Release 1.2, the user had to answer each consistency question when prompted.

If anything is inconsistent in the root filesystem **(/dev/dk0a)**, UNIX corrects it, halts, and reboots automatically. This removes the problems in Release 1.2 when an inconsistent UNIX root caused the user to reset the entire system (including LISP).

If UNIX reboots itself, type **CTRL-D** at the prompt again to bring up multiuser UNIX. **/etc/fsck** will run again, but this time the root will be consistent and the program will continue.

A login prompt will appear on every enabled UNIX terminal when the initialization sequence begun by typing **CTRL-D** is complete.

The following improvements have been made to the system:

- **Spell** now works.
- Various options to **nroff** (e.g. **-ms**) are implemented.
- **man -k** works.
- "?" in mail is implemented.

12. UNIX Usr File Installation

In this distribution, files are written onto the tape with relative pathnames (e.g. **../usr/bin/mail**) to simplify installation.

Remember that UNIX does not have file version numbers. The files on the new distribution tape will overwrite the file on your disk with the same name; be sure to back up any files you have customized.

NOTE: Be sure UNIX is booted multiuser before installing usr files!

To install the Release 2.0 UNIX usr files, mount the distribution tape and type in multiuser UNIX

```
cd /
tar xvpf /dev/rmt0
```

This will extract the 1159 files from the tape in about 20 minutes.

13. Freeing Port B and Tape Drive

Once LISP has used SDU Port B (by accessing a printer on port B, for example), the following commands must be issued in UNIX before UNIX can access that port:

```
/etc/ck -t ttyb
```

This "takes" control of the port from LISP. Since LISP is the default owner of Port B, no additional command has to be issued for LISP to regain use of the port.

Similarly, once LISP has used the half-inch tape drive, UNIX cannot access it. Type the following command to free the drive for UNIX use:

```
/etc/ck -t half-inch-tape
```

Again, no additional command has to be issued for LISP to regain use of the tape drive.

14. Updating UNIX Machines over the Network

This procedure can be used at sites with multiple UNIX systems on which it is impractical to load each system from tape.

14.1 First Machine

On a freshly updated release system (intact **root** and **usr** files), type the command:

```
update
```

This will create a file **/usr/update** that is a **tar** of all files in the **root**, **usr** and **sdu** filesystems (except for **/dev** and **/tmp**, since these should not be transferred).

14.2 To Update Another UNIX Machine over The Network

This assumes that both machines in question are booted in multiuser UNIX and can access each other over the network.

For the duration of this example, **unix-a** is the name of the fileserver (the machine that has the **/usr/update** file), and **unix-b** is the machine on which you want to install the update now.

Type on **unix-a**:

```
% cftp unix-b
cftp> raw
cftp> send /usr/update

Login: root
Password: <RETURN>
To foreign file: /usr/update
```

The machine will print **OPEN:** and a few other things, then give another prompt a few minutes later when the transfer is complete. Type **CTRL-D** to exit when the prompt is displayed:

```
cftp> CTRL-D
```

The file **/usr/update** is now on **unix-b**. Use **tar** to extract the files, thereby updating **unix-b**. Type on **unix-b**:

```
cd /
tar xvpf /usr/update
```

At this point, **unix-b** is updated. The entire procedure will take about two hours.

Appendix A. UNIX Files Size

Below are the number and size information of all files in the UNIX root, sdu, and usr file systems. This can be used to verify system integrity as a cross-check using the **/etc/fsck** file system consistency check program and the **df** disk space status program.

Release 2.0 UNIX system with UNIX usr files installed:

/etc/fsck:

```
/dev/dk0a: 303  files 3911 blocks 2752 free
/dev/dk0e: 1159 files 7450 blocks 30948 free
/dev/dk0g: 57   files 905 blocks 53 free
```

df:

```
/dev/dk0a: 59% full
/dev/dk0e: 19% full
/dev/dk0g: 94% full
```

Appendix B. Ascertaining Software Version

You can easily determine which LISP microcode and band you are using by printing the LISP disk label. This can be done either using the load program from the SDU or via the **(print-disk-label)** function in LISP.

To determine which version of the root is installed, you can type at the SDU the command

```
cat /uroot/version
```

This reads a file (the "**version**" file) that has existed and been updated appropriately since Release 1.0. The number of the release is on the first line of the file.

To read this version file in UNIX instead of at the SDU, type

```
cat /version
```

For the LMI Release 2.0 Root with UNIX, the version file says

```
LMI Release 2.0 -- 4/24/85

superboot linked to newboot 111
config 131
load 107
unix 3.286
```

For the LMI Release 2.0 Non-UNIX Root, the version file says

```
LMI Release 2.0 -- 4/24/85

superboot linked to newboot 111
config 131
load 107
```

If you are dealing with a system that contains UNIX, the files **/root.distr** and **/usr/usr.distr** are "cat"-able files containing a listing of the names, creation dates, permissions and owners of every file on the root and the usr file systems, respectively. These files were made at the time of the release with the **ls -lR** command in UNIX, and are on the system for baseline referencing.

Appendix C. Disk Labels for the Lambda Family

These are current as of Release 1.2 and hold for 2.0.

For a single Lambda or Lambda/PLUS:

Name	*Start*	*Length*
LMC1	25	500
LMC2	525	500
LMC3	1025	500
LMC4	1525	500
PAGE	2025	120000
FILE	122025	100000
LOD1	222025	35000
LOD2	257025	35000
LOD3	292025	35000
LOD4	327025	35000
METR	362025	8000

For a Lambda/2x2 or Lambda/2x2/PLUS:

Name	*Start*	*Length*
LMC1	25	500
LMC2	525	500
LMC3	1025	500
LMC4	1525	500
PAGE	2025	80000
PAG1	82025	80000
FILE	162025	75000
FIL1	237025	5000
LOD1	242025	30000
LOD2	272025	30000
LOD3	302025	30000
LOD4	337025	30000
METR	362025	8000

Release 2 Conversion Guide

October 1984

1000 Mass. Ave., Cambridge, Massachusetts 02139
Formatted with LaTeX on November 6, 1984.

Chapter 1.

Introduction

This documentation covers the incompatibilities for programmers between Release 1 and Release 2. The first section discusses in general what one can expect when converting code that ran in Release 1 for Release 2. The second section covers more specific incompatibilities between the releases. Note that two system facilities, packages and **defstruct**, have changed enough to warrant their own chapters in the Release Notes. The specifics of the changes and new features are discussed in those chapters, but the general guidelines for those facilities are discussed here in brief as well.

First, note that Release 1 binaries (QFASL files) cannot be successfully loaded in a Release 2 system. Because of new instructions, Release 2 binaries cannot be loaded into a Release 1 system. If you need to conditionalize code that depends on Common Lisp between Release 1 and Release 2, you may be advised that the symbol **common** appears on the features list in Release 2 but not in Release 1. (All implementations of Common Lisp have **common** on the features list.) You can use the read-time conditionalization feature of the reader to write code that will run in both Common and non-Common Lisp systems. Here is an example of its use: the Common Lisp form will be used in LMI Release 2, the newer versions of NIL, and other Common Lisp implementations, while the second form will be used in LMI Release 1, older versions of NIL, and Maclisp.

```
(defun print-with-radix (n the-base stream)
  #+common
  (write n :stream stream :radix t :base the-base :escape t)
  #-common
  (let ((*nopoint t))
    (format stream "#~D~VR" the-base the-base n)))
```

Here, the interpreter and the compiler never even get to see the forms meant for the "other" implementation of Lisp.

Chapter 2.

Common Lisp Issues

2.1. File Attribute Lists

Because of the adoption of Common Lisp, the default base in the Lisp Machine is now ten. If you have files with no base attribute, you should add one to the file while you are editing it in Zmacs. Another attribute that should be added to file attribute lists is a new one, the Readtable attribute. Readtables implement ability to run Common Lisp in the Lisp Machine without having to reference the new incompatible Common Lisp functions with the CLI: package prefix. Therefore, the readtable attribute indicates the syntax (for the editor) as well as what functions are to be used. No readtable attribute means to use the default (initially, the value of the readtable); a value of T means ZetaLisp; CL or Common-Lisp selects the Common Lisp mode. [1] Therefore, Release 1 files should be have an attribute to ensure that the correct mode is selected.

For changing the parts of the attribute list in an editor buffer, the extended (m-X) commands Set Readtable, Set Common Lisp, and Set Base will change the attribute list, as well as the values of any variables that may need to be changed in the editor.

2.2. Lexical Scoping

In Release 2, with the adoption of Common Lisp, lexical scoping becomes the rule in the interpreter. The compiler also supports full lexical scoping, including upward lexical closures. Lexical scoping is actually compatible with Lisp code that would compile without any warnings in Release 1, but because of some internal changes, the following must be noted:

[1] This scheme is also compatible with NIL.

1. The scope of local special declarations (usually heralded by **local-declare** or **declare** within a block-type special form like **defun**) has changed (but the old way still works for now – see the specific section for details). However, many of the uses of local special declarations can now go away because of lexical scoping. One of the most common instances of using a local **declare** in Release 1 is to make a variable special so that a **lambda**-expression being passed to a function (such as **mem**) can make a free reference to that variable.

2. By default, an error is signalled when a free reference is made to an undeclared variable. Therefore, any forgotten **defvars** or special declarations quickly manifest themselves in Release 2.

Here is an example of a function that will work in both releases, but one that can also take advantage of lexical scoping. Here is a function that is like **apropos**, except that only variables of a given type are printed:

```
(defun variable-type-apropos (string pkg type)
  (declare (special type))
  (apropos string pkg
                 ':predicate
                 #'(lambda (s) (and (boundp s)
                                    (typep (symeval s) type))))))
```

The local **declare** is needed here, in Release 1, because the lambda-expression that is being passed as the :**predicate** makes a reference to the variable **type**, which does not appear in the argument list of the lambda-expression. So, **type** variable is made special so that the passed function can refer to it.

This can be modernized to the following, after taking out the local declaration and using the new Common Lisp name for **symeval**:

```
(defun variable-type-apropos (string pkg type)
  (apropos string pkg :predicate
                   #'(lambda (s) (and (boundp s)
                                      (typep (symbol-value s) type)))))
```

This is more intuitive; we do not have to make **type** special because the predicate function is textually within the scope where **type** appears. Some other advantages to avoiding special variables are:

- Using the faster method of lexical variable lookup. (It is a simple extension of ordinary, old-style local variable lookup.)
- Avoiding name conflicts. If **type** is declared special by some other program, it may be possible for something wrong to happen if **type** were declared special in this function.

2.3. All Keyword Arguments Are Optional

In Release 2, **&optional** makes no difference after **&key** in a definition's lambda-list. Keyword arguments that are not supplied will not cause an error in Release 2, but will simply default to **nil**.

Chapter 3.

Other Changes

This part of the documentations covers some areas affected by incompatible changes in Release 2 that are more specific in nature – some, but not all of these changes are related to Common Lisp.

3.1. Array Order Has Changed in Release 2

The order of storage for array elements has been switched to be compatible with Maclisp and Common Lisp. Arrays are now stored in *row-major* order, which means that the last subscript varies the fastest through memory. The Lisp Machine used to store array elements in *column-major* order. The value of the constant **sys:array-index-order** is now **t**.

The change in order should be transparent for most programs, but there are some situations where code may have to be changed:

- Loops that go through large, multidimensional arrays (arrays where (**array-number-dimensions** *array*) > 1) will have to be rewritten if they took advantage of the older order to decrease paging.
- Programs that deal with pixel arrays should have been using **ar-2-reverse** and **as-2-reverse** to correctly reference the correct dimensions when given horizontal and vertical "coordinate" arguments.
- Use of multidimensional indirect arrays. If you have been exploiting the order in which elements appear when the displaced and source arrays are of different rank, you will have to rewrite the code. See section 8.2.1, especially the second paragraph on page 167, which discusses indirect arrays.

This is all explained in more detail in section 8.11 (pages 182-3) of the *Lisp Machine Manual.*

3.2. The Package System

The package system has been completely reimplemented. It is now a superset of the Common Lisp specification. Programs that used the hierarchical nature of the package system will have to be changed carefully; the structure of the package system is now an inheritance graph. The user and keyword packages are separate now; while this will not affect users who have always followed documentation about keywords, it will cause problems when programs running in Release 2 try to read data from Release 1, where some of that data was meant to be in the keyword package. Here is a quick rundown of the implications; read the chapter on packages for the details.

- **package-declare** is obsolete. Use **defpackage**, which is easier to use.
- The hierarchical structure that made symbol references like **foo:bar:baz** possible is gone.
- **user** and **keyword** are different packages keyword symbols always print out with colons in Release 2, but not in Release 1. This creates a problem with interchange of printed Lisp forms between releases, if keywords are included. (This could happen with data in files, or with a network protocol.)
- Many of the old package functions are still around, but they might not do exactly what was documented under Release 1. This is due to packages being arranged by inheritance, rather than by a strict hierarchy.
- Packages are now named like symbols.[1] In a clean Release 2 system, **(pkg-find-package "tv")** will get an error. A correct specification for the **TV** package would be `'TV` or `"TV"`. This kind of package specification (using lower case letters in a string) appears to be relatively common in Release 1 programs.
- In addition to changes in the package system, there has been symbol movement since System 94 (Release 1). Some symbols that were local to a package have now been globalized. Many of these are names of Common Lisp functions that were previously in the **FS** package. Also, note that since the keyword package is really a different package from the user package, keywords always print out with a colon even when they have the same name as a symbol in the global package.

3.3. Incompatible Calling Sequences

The following is a list of incompatible functions and macros that take a different set of arguments than they did in Release 1. Constructs marked with an asterisk (*) will still accept the old calling sequence, but are planned to change incompatibly in the future.

[1]More precisely, they are named like the *print names* of symbols.

```
break *
y-or-n-p
yes-or-no-p
si:print-object
si:print-list
select-match
```

Other classes of functions also have new incompatible calling sequences:

- **Evalhook** functions.
- **Applyhook** functions.
- **Macroexpand-hook** ????
- Macro expanders. [These can still take just argument for now.]

3.4. Function Warnings

You will get warnings, when compiling code with calls to obsolete functions, to use the preferred Common Lisp functions that are upward compatible. For example, **array-dimension-n** should be replaced by **array-dimension**. Such obsolete functions still work, but will go away in the future. Relatively few of these warnings should occur in well-written Release 1 programs, since the green (fifth) edition of the manual had also documented many of these functions as obsolete. But now that the newer functions have become part of the Common Lisp specification, the obsolete functions should not be used anymore.

3.5. Input and Output

3.5.1. Character Objects

Character objects are here, but it is not necessary for Zetalisp programs to always use character-oriented functions if they handle characters; most character functions will work on integers, and *vice versa*. The normal **aref** on strings (arrays of type **art-string** and **art-fat-string**) still returns integers; there is a Common Lisp version, **cli:aref**, which is exactly the same as **aref** except it returns character objects out of strings. (Using **vref** on strings will also return character objects, because it is a synonym for **cli:aref**.) Integer and character comparison and coercion actually work on both types of data, though this should not be relied upon in future releases. In particular, if you are using **aref** (in a character-oriented user-interface) to reference a command in a command table, for instance, the "character" should be coerced to an integer with **char-int**, even if the character is not actually a character object in your program. Besides, it is better to use the functions that operate on characters to make the code easier to understand, and to port to other Common Lisp systems.

When the standard Zetalisp readtable is in effect, **#\character** reads in as an integer; in Common Lisp syntax, such input will be read in as a true character object. In Zetalisp syntax, character objects print out as **#≠/character**; **#\character** currently is equivalent to **#/character**, which reads in as an integer.

3.5.2. Variable Issues

Though not predominantly an incompatible programming change, Common Lisp has given new names to many of the standard Lisp input and output "control variables" that control slashification (now called *escaping*), input and output radices, and the verbosity of the printout. The old names are still accepted, but programmers are encouraged to change the variable names in their code at their convenience.

Common Lisp has also introduced new variables that control other aspects of the reader and printer. Programmers should especially be interested in ***print-array***, which prints out arrays readably, and ***print-circle***, which prints out circular structure (which can be built out of all kinds of structure, including conses, arrays, and instances) readably. The variable ***print-gensym***, when **T**, prints out uninterned symbols so that they can be read back and still be uninterned. A complete description of the variables that affect printing can be found in section 23.2 (page 514) of the Lisp Machine Manual.

There is a new Common Lisp variable called ***print-radix***; if **T**, numbers will be printed out with some indication of what radix they should be read in with. If ***nopoint** is **T**, then ***print-radix*** really does take effect; otherwise, the old behavior with ***nopoint** takes effect.

3.6. FORMAT

The **format** function has changed; there are some new and incompatible control-string ("tilde") directives. An incompatible directive that is quite common in typical **format** control strings is `~X`, which used to perform a tabbing function – it now prints out its argument in hexadecimal. Occurrences of `~X` should be replaced to `~@T`. The old `~G` has been moved to `~@`; `~E` and `~F` have been extended to take more arguments but they have not become incompatible. For more information, read the documentation of the **format** function in the Lisp Machine Manual. (The Common Lisp release notes also briefly describe the new extensions to **format**). Remember that the error signalling functions and some other utilities pass their control-string arguments to **format**.

Since there is no "edit callers" command in Zmacs that can help you to track down obsolete **format** control strings, if you want to track them down, you can now get acquainted with the extended string search capabilities of Zmacs if you have not used them before. The functionality, which includes pattern matching, is available through the Zmacs extended commands **String Search** and **Tags Search**. [2] When typing a search string, use ⟨c-H⟩ ⟨Help⟩ to describe the various patterns that can be matched. For example, typing in the search string ~⟨c-H⟩⟨c-X⟩ X will search for an occurrence of the string ~, any character, X.

3.7. Syntax changes

Because of the incompatibilites of Release 1 and Release 2 syntax, the exchange of objects using printed representation will not always work correctly between releases. (The situation arises when **reading** in a data file produced by the **print** function, or using Lisp forms in a network protocol: one must be careful if one release is doing the **reading** and the other is doing the **printing**.) In general, Release 2 can read anything that can be printed readably in Release 1, but not the other way around. Besides the problem with syntax, there are certain situations that would cause what was meant to be a keyword (at that time, a symbol in the **user** package) to be printed out without a colon prefix in Release 1. Names of flavors of most hosts and pathnames have changed in Release 2, so hosts and pathnames printed out by Release 1 will not read back in, either. (If this is a serious problem in your application, notify us and LMI will supply a fix.) Most of these problems can be avoided by simply switching all machines at a site to Release 2 at the same time.

3.8. Logical Pathnames and Hosts

Logical hosts have been changed in an upward compatible manner; now, the translations are not directories, but mappings from one (usually wildcarded) specification to target file name. Also, the directories of a logical pathname can be structured: IO: FILE: can be considered a "subdirectory" of IO:. The newer features of logical pathnames (and the definition of logical pathname hosts) is documented in more detail in the Release 2 notes. It is now possible to specify, for example, on a typical Twenex host's SYS host translations, that all directory names translate to directories under the <L> directory: one translation for one-level directories, one translation for two-level directories (like IO: FILE:), and one translation that will specially translate CHAOS: HOSTS TEXT in PS:<SYSTEM>HOSTS2.TXT, which is where host table usually resides on Twenex host. Here is the *translation* needed: it would get passed to **fs:add-logical-pathname-host**:

[2] Try "Apropos" in Zmacs on the string "**tags table**" to learn how to select various kinds of groups of files (loaded buffers, systems defined with defsystem) as tags tables.

```
(("CHAOS; HOSTS TEXT" "PS:<SYSTEM>HOSTS2.TXT")
 ("*.*;" "PS:<L.*.*>")
 ("*;" "PS:<L.*>"))
```

Site translations can now be updated automatically: for example, SYS translations (the definition of the SYS host) are actually kept in the site directory. The function **fs:make-logical-pathname-host** requests that a logical host get its translations from the SITE; directory. For Release 2, at least the SYS translations file must be present. If you have defined your own logical hosts for your own systems, you can use this feature, but it is not necessary. (In that case, you can still use **fs:add-logical-pathname-host**.)

3.9. Host Device Specification In Filenames

Because of changes in the internals of the file access system, a change has been made in the rules for determining the device of a partially-specified pathname.[3] If you wish to specify a different device in a pathname that is still going to have the same host as the default pathname, you must put a colon in front of the device name.[4] This can be considered mostly a user-interface change, but users are advised to keep this in mind if they suspect that they have incompletely specified physical pathnames wired into their programs. The change is really important only for users of VMS, Twenex, and ITS file servers.

[3] This restriction is compatible with the Symbolics system, in which the same change had taken place a while ago.

[4] Actually, omitting the colon may still work, but if there is a file server host on the network with the same name as the device, something will probably go wrong. Even if there is no such host, your machine may also try to contact host table servers to check to see if the name is valid. The time to do this can be noticeable.

Editing Lambda Site Files
The SITED Editor

For LMI Release 2.0 1446-0000

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Introduction

Editing the Lambda's Site Information

This document describes methods for editing the Lambda's site information to customize it for different sites. Included are detailed procedures for:

A Customizing the first Lambda(s) at a new site.

B Customizing the new Lambda(s) at a site that already has other Lambdas on the Chaosnet.

C Making minor changes in the Lambda's site information.

D Customizing the site information of the UNIX side of a Lambda/Plus.

If you are editing site files for the installation of LMI Software Release 2.0, see Chapter 10, "Site File Installation Notes" in the *Release 2.0 Installation Packet.*

No prior experience with the Lambda is required in order to use these new procedures. One should, however, be familiar with some basic computer and network terminology, such as *file system*, *directory*, *network address*, *disk partition*, *database*, and *compilation*. If assistance is needed, please call:

LMI Customer Service

at our toll-free number:

1-800-872-LISP (except Massachusetts)
1-800-325-6115 (inside Massachusetts)

Overview of the Lambda Site Information

The LISP environment has a small database of *site info* which defines how the machines and peripherals are configured at the site. The database includes the *host table*. Each entry in the host table describes one machine at the site: the name of the machine, its network address, what kind of operating system it runs, etc. The database also defines various *site options*, such as the timezone, the name of the site, and what kinds of printers are used at the site.

The site information is recorded in a set of files called *site files*, which are kept in a special directory on one machine.

To change a Lambda's site information, first edit the site files. Then compile the new site files and load them into LISP. This changes the site information in the LISP environment, but only temporarily; the next time the Lambda is booted, it will revert to the old LISP environment. To make the new site information permanent, you must save the new LISP environment into a disk partition and set the disk label so that the Lambda will boot from that disk partition.

Brief Description of the SITED Program

The procedures below use a new program called SITED through which you can read existing site files, edit them, then write them back to the disk. Its user interface is command-oriented. For example, if you type the command **readfiles**, it will read the site files into SITED. If you type the command **writefiles**, it will write a new set of site files into a directory. There are many other commands for editing the site files in various ways.

One need not be familiar with all the features and commands of SITED to use the procedures in this document. The SITED program is self-documenting. For more information, enter SITED and press ⟨HELP⟩.

SITED runs only under Release 2 software.

If you ever accidentally abort out of SITED, type:

```
(sited 1)
```

This re-enters SITED in a way such that the contents of the database will still be intact.

Chaosnet Names, Addresses, and Nicknames

Each machine on the Chaosnet at a site has one official name. The name can't have any spaces or other unusual characters in it, and character case doesn't matter. To make the machine names unique, a short name for the site is typically used as a prefix in the official names. (Some examples: **LMI-LAMBDA-ONE, MIT-OZ, UTEXAS-CSLAB-VAX.**)

It is usually convenient for each machine to have at least one short nickname. This makes it easy to distinguish one machine from another, and allows the names of both processors of a 2×2 to be contained within the 32 characters allowed for this.

> NOTE: The procedures contained in this document assume that the Lambda's official name will be no longer than 32 characters.

A machine's Chaosnet address can be almost any number between 1 and 57777 octal. It is a 16-bit number, the high 8 bits indicating the subnet number. In most cases, all the machines at a site should have addresses with the same subnet number. Therefore, the addresses will all be within a range of 400 octal.

At LMI, all the machines use subnet number 7, and the high byte of all the machines' Chaosnet addresses is 7. Thus, the LMI machines' Chaosnet addresses lie between 3400 and 3777 octal.

The only addresses that you must never assign to an actual machine are 0 and 3412 octal. 0 is the Chaos broadcast address. 3412 is an address reserved by LMI for an extra host called *LMI Amnesia*. This host will be included on the hostat list and should not be deleted. This is the address that a Lambda uses when it can't figure out its own name, which usually happens only when the Lambda's site information or pack-name is incorrect.

Many customer sites have kept to the convention of having all addresses on subnet 7. Some sites use a different subnet. A few sites use addresses on two or more subnets, although this is not recommended practice unless the site has many machines.

In rare circumstances, a machine will have more than one Chaosnet address, each address being for a different subnet. For example, a CADR bridge has two addresses, one for the Ethernet subnet, and one for the Chaos hardware subnet. The front end PDP-11 for a DEC-20 (running the MINITS operating system) may also have two Chaosnet addresses. In the procedures described in this document, you can indicate the multiple Chaosnet addresses of a machine in the following way. Whenever SITED askes the question *What is the machine's Chaosnet address?*, you can answer by typing more than one octal addresses on the same input line, separated by ⟨SPACE⟩ and followed by ⟨RETURN⟩.

Chaosnet Host Tables

For step **a1** of **Procedure A**, you need to obtain a list of the names, addresses, and machine types of all the machines at the site on the Chaosnet. All this information should already be contained in a file on some of the non-LMI machines which are already on the Chaosnet. This file is called the host table.

Chaos host table format is fairly straightforward. Each line in the host table describes one Chaosnet machine at the site. The format of each line is:

HOST *nnnnnn*, **CHAOS***xxxxxx*, **USER**, *ssssss*, *hhhhhh*, [*nicknames*]

where:

nnnnnn is the official name of the machine,
xxxxxx is the octal Chaosnet address for that machine,
ssssss is a keyword describing that machine's operating system,
hhhhhh is a keyword describing that machine's hardware,
nicknames can be any number of nicknames for that machine, separated by commas.

Sometimes the word *USER* will be replaced by the word *SERVER*, but in either case this field is ignored by the Lambda.

On rare occasions, a machine will have multiple Chaosnet addresses. Such a machine's host table entry will look like:

HOST *nnnnnn*, [**CHAOS** *xxxxxx*, **CHAOS***yyyyyy*], **USER**, *ssssss*, *hhhhhh*, [*nicknames*]

where *xxxxxx* and *yyyyyy* are the machine's two Chaosnet addresses. The other fields are the same as before.

1. Procedure A

CUSTOMIZING FOR THE FIRST LAMBDA AT A NEW SITE

This procedure customizes the site information for installation of the first Lambda at a new site. When it is complete, the Lambda will be able to communicate with all the other machines on the Chaosnet.

If the site already has Lambdas communicating on the Chaosnet, use **Procedure B.**

If the first new Lambda is a 2×2, do this procedure from the monitor associated with the processor 0 side of the 2×2.

If more than one new Lambda has arrived at the new site, you need not repeat this entire procedure for each one. Install the correct site information on one Lambda, then copy its band to the other Lambdas via the Chaosnet. See Section 1.8.

1.1 Preparation

Step #

Step a1 Obtain a list of all the machines already communicating on the Chaosnet at the site. For each machine, list (on paper) the following information:

- the official name of the machine,
- its Chaosnet address,
- what kind of machine it is (e.g. LISP Machine, VAX-VMS, VAX-UNIX, DEC-20).

One of the non-LMI machines at the site may have all this information contained in a file called the *host table*. See the Introductory section, *Chaosnet Host Tables*.

Add to this list the official names and Chaosnet addresses of the new Lambda(s). If the machines have not been given names, read the guidelines in the section *Chaosnet Names and Addresses* before assigning names.

Step a2 The site files should reside in a special directory on a Lambda. If there is more than one Lambda at your site, choose one to maintain the site file directory. Use that Lambda for the rest of this procedure.

Create the site file directory by typing:

```
(fs:create-directory "lm:ql.customer-site;")
```

1.2 Site Definition

Step a3 Start SITED by typing:

```
(sited)
```

The SITED program should print a welcome message and then the prompt, **SITED command>**. *Slanted text* indicates user input.

Type:

SITED command> *interactive* ⟨RETURN⟩

SITED then prints the following message:

```
Type one of the following characters:
's' -- for questions about the site options.
'm' -- for questions about a particular machine.
'x' -- to return to SITED command level.
```

Type the letter **s**.

SITED then asks the following questions about the site. Each question will indicate the form of the answer it needs.

```
What is the name of the site?
```

```
What is the timezone there?
```

```
What kind of printer (if any) is used at the site to print
normal text?  And to which machine is it connected?
```

```
What kind of printer (if any) is used at the site to print
bit-arrays (screen dumps)?  And to which machine is it connected?
```

1.3 Machine Definition

Step a4 After you answer the last question, you will be asked to choose either **s**, **m**, or **x** again. Type **m**.

SITED then asks you to type a machine name. Type the official name of first machine on the list of machines you made in step **a1**.

After typing the official name, type **yes** to answer the question "Should I create a new host table entry?".

> NOTE: If you have made an error and do *not* wish to create a new entry, type **n** for "no". SITED will return to the beginning of Step **a4**. Type **m** again, re-enter the name correctly, and continue with Step **a4**.

SITED then asks the following questions about that machine. Each question will indicate how it wants you to type the answer.

`What kind of machine is it?`

`What is its Chaosnet address?`

`What nickname(s) will the machine have?`

`Does this machine have the main set of LMI source files?`

> NOTE: SITED allows only one machine to be designated as having the main set of LMI source files. That machine should be used for this procedure. (See Step **a2**.)
>
> It is possible to specify another machine; call LMI Customer Service for assistance.

Step a5 After you answer the last question, you will be asked to choose either **s**, **m**, or **x** again.

- If you are installing more than one Lambda on Chaosnet, type **m**, and repeat step **a4** answering the questions for the next machine on your list.

  Repeat Step **a4** for each of the machines on your list. If you are installing a Lambda/Plus or Lambda/2×2/Plus, be sure that the name and the Chaosnet address of the UNIX processor is added at this time.

- After you have answered the questions for every machine on your list, choose the letter **x** instead of **m**.

Step a6 At the SITED prompt, type: (User input is in *slanted text.*)

`SITED command>` *checkinfo* ⟨RETURN⟩

The site editor may discover omissions in your site files. For example:

```
The PRETTY-NAME property for OURSITE-
LAMBDA-A doesn't have any value
specified for it.

Should it be set to the default value
of "Oursite Lambda A" ? (y or n)
```

If you type **n** for "no", you will need to type `interactive` at the `SITED command>` prompt. Continue as you did in Step **a3**, making the necessary changes.

After the site information has been determined to be complete and correct, the following will be displayed:

```
No errors found in the information.
To generate new site files, use the
WRITEFILES command.
```

1.4 Create Site Files

Step a7 You can now create a new set of site files. Type:

`SITED command>` *writefiles* ⟨RETURN⟩

You will be asked to specify the directory for writing the new site files. Type (don't forget the semi-colon):

`Directory name>` *lm:ql.customer-site;* ⟨RETURN⟩

Four new site files, SITE.LISP, LMLOCS.LISP, HOSTS.TEXT, and SYS.TRANSLATIONS, will be generated and written into that directory.

Step a8 Exit SITED by typing:

`SITED commmand>` *quit* ⟨RETURN⟩

and answer **yes** to the question "Do you really want to quit SITED?"

1.5 Compile Site Files

Step a9 Compile the new site files by typing:

```
(si:set-sys-host "lm" nil nil "ql.customer-site;")
```

Type:

```
(make-system 'site :compile
                    :no-reload-system-declaration :noload)
```

and answer **yes** to all questions.

Step a10 Check the files after they have been compiled. Type:

```
(setq x si:disk-pack-name)
(si:set-pack-name "nnnnn")
```

where *nnnn* is the name for the Lambda according to the new site files.

Type:

```
(setq si:host-alist nil)
(update-site-configuration-info)
(si:set-pack-name x)
```

1.6 Check Files

The new site files are now loaded into the LISP environment, and should be checked before they are loaded into a new band.

The LISP function **print-herald** will list information about your system, its bands and memory. The last line will have the format:

site-pretty-name machine-pretty-name **with associated machine** *associated-machine-name*

A typical example would be:

```
LMI Lambda One with associated machine LAM3
```

Step a11 Type:

(print-herald)

Some possible problems that may be seen at this point are:

- Site name incorrect:
  Check the listing in the file **LM:QL.CUSTOMER-SITE;SITE.LISP.**
  Especially note the value for the **:SITE-PRETTY-NAME** option.
- Machine listed as **HUH? [No Chaos Address]**:
  The name *nnnn* listed in step **a10** was not defined in the site file
  LM:QL.CUSTOMER-SITE;HOSTS.TEXT.
- Machine name listed as **Unknown**:
  The name is lacking an entry in the site file
  LM:QL.CUSTOMER-SITE;LMLOCS.LISP.

  Each machine's pretty name should be specified in the file **LMLOCS** with the format:

  (*machine-official-name machine-pretty-name console-location*
  (*building-name floor-number*) *associated-machine*)

Step a12 After the site file information is listed correctly, test communication on the network. Type:

(hostat)

to print a list of all the machines on the network that are responding to the Lambda.

NOTE: **hostat** will list an extra host called *LMI Amnesia* with the Chaosnet address 3412 octal. Do not remove this entry; see the section *Chaosnet Names, Addresses, and Nicknames* for more information.

If there are machines that do not seem to be responding, try **hostat** again, this time giving the name of the unresponsive machine. Type:

(hostat *machine-name***)**

If a machine still does not respond:

- Check Chaos addresses in **LM:QL.CUSTOMER-SITE;HOSTS.TEXT.**
- Check the other machine's own site files.
- Check **hostat** from the other machine's console.

- Check the Ethernet cables and transceivers.
- If there seem to be actual network problems, call LMI for assistance.

1.7 Install Site Files into a New Band

After the site files are listed correctly and are communicating on the network, they can be permanently installed into a new band.

Step a13 Cold-boot the machine. To do this, press both Control keys, both Meta keys, and the Rubout key, all at the same time. Cold-booting should take about five minutes.

Type:

```
CTRL-META-CTRL-META-(RUB OUT)
```

CAUTION: From this point until Step **a18**, do not enter SITED, ZMACS, or any window other than LISP Listener 1, and do not type anything that is not listed as an instruction. If this should happen, or if anything else goes wrong, reboot the machine before you proceed and continue from this point.

Step a14 Log in by typing:

```
(login 'lispm t)
```

Step a15 Print a list of the Lambda's disk partitions. Type:

```
(print-disk-label)
```

Each line printed describes a disk partition: its name, starting block, size, and a comment describing its contents. An empty string ("") indicates that the partition is empty.

Identify the partition known as the *current band*. This partition is marked with an asterisk (*) and starts with the letters **LOD**. Write the name of this band on paper so it will not be forgotten.

Choose a disk partition for saving the LISP environment that contains the new site information. This partition:

- Should be an empty partition whose name starts with the letters **LOD** (e.g. **LOD1**, **LOD2**, **LOD3**, etc.).
- Should NOT be the current band.

Step a16 In this step, the name of the new Lambda is made permanent.

- For a Lambda or Lambda/Plus, type:

  (si:set-pack-name "*nnnnnn*")

  where *nnnnnn* is the name for the new Lambda. This can be either the official name or the nickname.
- For a Lambda/2×2 or 2×2/Plus, type:

  (si:set-pack-name "*nnnnnn1 nnnnnnn2*")

  where *nnnnnn1* is the name for the slot 0 processor and *nnnnnn2* is the name for the slot 4 processor. These can be either the official names or the nicknames, but the total number of letters cannot exceed 32 characters.

Step a17 Type:

```
(setq si:host-alist nil)
(update-site-configuration-info)
(disk-save "pppp")
```

where *pppp* is the four-letter disk partition name that you chose in step **a15**.

The system will ask "Do you really want to clobber partition *pppp*?" Answer **Yes**.

You will be asked to type a new comment for the herald. Type:

vvv.vv mm-dd-yy ⟨RETURN⟩

where *vvv.vv* is the system version number (e.g. 102.117) and *mm-dd-yy* is the date.

You will be prompted for a new comment for the disk label. Delete the current comment by positioning the cursor at the beginning of the comment and typing **CTRL-K**. Enter the disk label by typing the version number and date just as you did for the herald.

It will take about 10 minutes to save the new LISP environment into the disk partition.

Step a18 Make the Lambda use the new band by typing:

(si:.set-current-band "*pppp*")

where *pppp* is the four-letter name of the disk partition you typed in step **a17**.

Step a19 The new LISP environment is now permanently installed. Reboot the Lambda; if it is a 2×2, reboot both sides.

Type:

```
CTRL-META-CTRL-META-<RUB OUT>
```

1.8 Copy New Band to a Second Lambda

After one Lambda has a new LISP environment, it can be copied to a second Lambda by the following procedure.

NOTE: This procedure is NOT necessary for a Lambda/2×2 since the slot 0 processor and the slot 4 processor boot from the same band. Use this procedure only for two separate Lambdas.

Step a20 First obtain the following information:

- The octal Chaosnet address of the Lambda that has the new band. To get this, type on its console:

  ```
  (format nil "~O" chaos:my-address)
  ```

  This returns the Lambda's octal Chaosnet address in a character string.
- The name of the current band on the first Lambda. (See Step **a15**.)

Write these two things on a piece of paper.

Step a21 Look at the disk partitions of the second Lambda. Type:

```
(print-disk-label)
```

Choose a disk partition on the second Lambda to receive the new LISP environment. This partition:

- Should be an empty partition whose name starts with the letters **LOD** (e.g. **LOD1**, **LOD2**, **LOD3**, etc.)
- Must NOT be the current band. (See Step **a15**.)

Step a22 Type on the second Lambda:

```
(array-initialize chaos:routing-table-type :ethernet)
(si:receive-band "chaos|xxxx" "pppp" "qqqq")
```

where:

- *xxxx* is the octal Chaos address of the first Lambda (which you wrote down in Step **a20**).
- *pppp* is the name of the current band for the first machine (set in Step **a19**).
- *qqqq* is the name of the empty disk partition of the second machine (chosen in Step **a21**).

This should take about half an hour. Do not use either Lambda during this time.

Step a23 Type:

(si:set-current-band "*qqqq***")**

where *qqqq* is the same as in Step **a22**.

Step a24 Now permanently change the name of the second Lambda.

- For a Lambda or Lambda/Plus, type:

  (si:set-pack-name "*nnnnnn***")**

  where *nnnnnn* is the new name for the second Lambda. This can be either the official name or the nickname.
- For a Lambda/2×2 or 2×2/Plus, type:

  (si:set-pack-name "*nnnnnn1 nnnnnnn2***")**

  where *nnnnnn1* is the name for the slot 0 processor and *nnnnnn2* is the name for the slot 4 processor. These can be either the official names or the nicknames, but the total number of letters cannot exceed 32 characters.

Step a25 Reboot the second Lambda; if the second Lambda is a 2×2, reboot both of its processors. (Use **CTRL-META-CTRL-META-**⟨RUB OUT⟩ to boot.)

Step a26 The new band is now permanently installed on the second Lambda. Use the tests given in Steps **a11** and **a12** to check the new environment.

1.9 Copy New Band to Additional Lambdas

If there are other new Lambdas, repeat Section 1.8 for each machine. Copy the new band to one Lambda at a time, not in parallel.

2. Procedure B

CUSTOMIZING THE SITE INFORMATION OF A NEW LAMBDA AT A SITE WITH LAMBDA(S) ALREADY ON THE CHAOSNET

Use this procedure to customize the site information for a newly arrived Lambda at a site that already has Lambda(s) on the Chaosnet. When this procedure is finished, the new Lambda will be able to communicate with all other machines on the Chaosnet.

If the new Lambda is a 2×2, use this procedure for the slot 0 processor of the 2×2.

If more than one new Lambda has arrived at the site, you need not repeat this entire procedure for each. Install the correct site information on one Lambda, then copy its band to all the other Lambdas via the Chaosnet. See Section 2.5.

2.1 Preparation

Step #

Step b1 Obtain the following information:

- The official name of each new Lambda to be installed.
- Its Chaosnet address

If you are choosing the names and addresses yourself, read the guidelines in the section *Chaosnet Names and Addresses.*

Make a list of all the new Lambdas' official names and addresses.

Step b2

NOTE: This procedure must be done from the console of an ''old'' Lambda; one already installed on the Chaosnet. If there is more than one Lambda, use the one that has the main source files.

Start SITED by typing:

```
(sited)
```

The SITED program should print a welcome message and then the prompt `SITED command>`. *Slanted text* indicates user input.

Type:

```
SITED command> readfiles ⟨RETURN⟩
```

and answer **yes** to the question:

```
Do you really want to proceed? (y or n) y
```

Step b3 SITED will then ask you to specify a directory.

Type: (Don't forget the semi-colon.)

```
sys:site; ⟨RETURN⟩
```

Step b4 Type:

```
SITED command> interactive ⟨RETURN⟩
```

SITED then prints the following message:

```
Type one of the following characters:
's' -- for questions about the site options.
'm' -- for questions about a particular machine.
'x' -- to return to SITED command level.
```

Type the letter **m**.

2.2 Create New Site Information

Step b5 SITED then prompts you for a machine name. Type the official name of the new Lambda and answer **yes** to the question "Should I create a new host table entry?".

> NOTE: If you have made an error and do *not* wish to create a new entry, type **n** for "no". SITED will return to the beginning of Step **b5**. Type **m** again, re-enter the name correctly, and continue with Step **b5**.

Step b6 SITED then asks the following questions about the new Lambda. Each question will indicate the form of the answer it needs.

```
What kind of machine is it?
```

```
What is its Chaosnet address?
```

```
What nicknames will the machine have?

Does this machine have the main set of source files?
```

The answer to the last question should be "no"; the files should be on a Lambda already installed at this site.

Step b7 After the last question, you will be asked to choose either **s**, **m**, or **x** again.

- If you are installing more than one Lambda on Chaosnet, type **m**, and repeat step **a4** answering the questions for the next machine on your list.

  Repeat Step **a4** for each of the machines on your list. If you are installing a Lambda/Plus or Lambda/2×2/Plus, be sure that the name and the Chaosnet address of the UNIX processor is added at this time.

- After you have answered the questions for every machine on your list, choose the letter **x** instead of **m**.

Step b8 Then at the `SITED command>` prompt, type:

`SITED command>` *checkinfo* ⟨RETURN⟩

The site editor may discover omissions in your site files. For example:

```
The PRETTY-NAME property for OURSITE-
LAMBDA-A doesn't have any value
specified for it.

Should it be set to the default value
of "Oursite Lambda A" ? (y or n)
```

If you type **n** for "no", you will need to type `interactive` at the `SITED command>` prompt. Continue as you did in Step **b4**, making the necessary changes.

After the site information has been determined to be complete and correct, the following message will be displayed:

```
No errors found in the information.
To generate new site files, use the
WRITEFILES command.
```

Step b9 You can now create a new set of site files. Type:

`SITED command>` *writefiles* ⟨RETURN⟩

SITED then asks you to specify a directory for writing the new site files. Type:

sys:site; ⟨RETURN⟩

Four new files will be generated into the site file directory: **SITE.LISP**, **LMLOCS.LISP**, **HOSTS.TEXT**, and **SYS.TRANSLATIONS**.

Step b10 Exit SITED by typing:

SITED command> *quit* ⟨RETURN⟩

and answer **yes** to the question "Do you really want to quit SITED?"

2.3 Compile the New Site Files

Step b11 Compile the new site files by typing:

```
(make-system 'site :compile
              :no-reload-system-declaration :noload)
```

and answer **yes** to all questions.

Step b12 After the new site files are compiled, check by typing:

```
(update-site-configuration-info)

(hostat)
```

Be sure the new Lambda's names are included in the hostat list.

NOTE: The new Lambda will not respond to **hostat** until the new site information has been copied into its band.

2.4 Install New Site Information into Band

Step b13 From the console of the "old" Lambda, cold-boot the machine by pressing both Control keys, both Meta keys, and the Rubout key, all at the same time. Cold-booting should take about five minutes.

Type:

CTRL-META-CTRL-META-⟨RUB OUT⟩

CAUTION: From this point until Step **b17**, do not enter SITED, ZMACS, or any window other than LISP Listener 1, and do not type anything that is not listed as an instruction. If this should happen, or if anything else goes wrong, reboot the machine before you proceed.

Step b14 Log in by typing:

```
(login 'lispm t)
```

Step b15 Print a list of the Lambda's disk partitions. Type:

```
(print-disk-label)
```

Each line printed describes a disk partition: its name, starting block, size, and a comment describing its contents. An empty string (**""**) indicates that the partition is empty.

Identify the partition known as the *current band*. This partition is marked with an asterisk (*) and starts with the letters **LOD**. Write the name of this band on paper so it will not be forgotten.

Choose a disk partition for saving a new band. This partition:

- Should be an empty partition whose name starts with the letters **LOD** (e.g. **LOD1**, **LOD2**, **LOD3**, etc.).
- Should NOT be the current band.

Step b16 Type:

```
(update-site-configuration-info)
(disk-save "pppp")
```

where *pppp* is the four-letter name of the disk partition you chose in Step **b15**.

The system will ask, "Do you really want to clobber partition *pppp*?". Answer **Yes**.

You will be asked to type a new comment for the herald. Type:

vvv.vv mm-dd-yy ⟨RETURN⟩

where *vvv.vv* is the system version number (e.g. 102.92) and *mm-dd-yy* is the date.

You will be prompted for a new comment for the disk label. Delete the current comment by positioning the cursor at the beginning of the comment and typing **CTRL-K**. Enter the disk label by typing the version number and date just as you did for the herald.

It will take about 10 minutes to save the new band into the disk partition.

Step b17 Make the Lambda use the new band by typing:

```
(si:.set-current-band "pppp")
```

where *pppp* is the four-letter name of the disk partition you typed in step **b16**.

Step b18 The new configuration is now permanently installed. Reboot the Lambda; if it is a 2×2, reboot both sides.

Type:

```
CTRL-META-CTRL-META-RUB OUT
```

2.5 Copy New Band to Other Lambdas

The LISP band you have just set can be copied to all the other Lambdas at the site (both old and new). Follow the procedure given in Steps **b19** through **b25** for each machine in turn.

NOTE: The slot 0 and slot 4 processor of a Lambda/2×2 boot from the same band; use this procedure only for the slot 0 side of a 2×2.

Step b19 First obtain the following information:

- The octal Chaosnet address of the Lambda which has the new band. To get this, type on its console:

  ```
  (format nil "~O" chaos:my-address)
  ```

  This returns the Lambda's octal Chaosnet address in a character string.
- The name of the current band on that Lambda. (See Step **b15**.)

Write these two things on a piece of paper.

Step b20 Look at the disk partitions of the Lambda that is to receive the new band.

Type **on the Lambda that is to receive the new band**:

```
(print-disk-label)
```

Choose a disk partition on this Lambda to receive the new band. This partition:

- Must have a name starting with the letters **LOD** (e.g. **LOD1**, **LOD2**, **LOD3**, etc.)
- Must NOT be the current band. (See Step **b15**.)

Step b21 Type on the Lambda that is receiving the new band:

```
(array-initialize chaos:routing-table-type :ethernet)
(si:receive-band "chaos|xxxx" "pppp" "qqqq")
```

where:

- *xxxx* is the octal Chaos address (which you wrote down in Step **b19**) of the Lambda with the new band installed.
- *pppp* is the name of the current band for the first machine (set in Step **b16**).
- *qqqq* is the name of the empty disk partition of the receiving machine (chosen in Step **b20**).

This should take about half an hour. Do not use either Lambda during this time.

Step b22 Type:

```
(si:set-current-band "qqqq")
```

where *qqqq* is the same as in Step **b21**.

Step b23

NOTE: This step is necessary only for the new Lambda(s) being installed on the Chaosnet. When copying the new band to Lambda(s) previously communicating on the Chaosnet, omit this step and proceed to Step **b24**.

- For a Lambda or Lambda/Plus, type:

  ```
  (si:set-pack-name "nnnnnn")
  ```

  where *nnnnnn* is the new name for the second Lambda. This can be either the official name or the nickname.
- For a Lambda/2×2 or 2×2/Plus, type:

  ```
  (si:set-pack-name "nnnnnn1 nnnnnnn2")
  ```

  where *nnnnnn1* is the name for the slot 0 processor and *nnnnnn2* is the name for the slot 4 processor. These can be either the official names or the nicknames, but the total number of letters cannot exceed 32 characters.

Step b24 Reboot the Lambda that has received the new band; if this Lambda is a 2×2, reboot both of its processors. (Use **CTRL-META-CTRL-META-**⟨RUB OUT⟩ to boot.)

2.6 Check the New Site Files

After the new band has been installed on the Lambdas, use these steps to check their operation.

The LISP function **print-herald** will list information about your system, its bands and memory. The last line will have the format:

site-pretty-name machine-pretty-name **with associated machine** *associated-machine-name*

A typical example would be:

```
LMI Lambda One with associated machine LAM3
```

Step b25 Type:

```
(print-herald)
```

Some possible problems that may be seen at this point are:

- Site name incorrect:
  Check the listing in the file **LM:QL.CUSTOMER-SITE;SITE.LISP**.
  Especially note the value for the :**SITE-PRETTY-NAME** option.
- Machine listed as **HUH? [No Chaos Address]**:
  The name *nnnn* listed in step **b23** was not defined in the site file
  LM:QL.CUSTOMER-SITE;HOSTS.TEXT.
- Machine name listed as **Unknown**: The name is lacking an entry in the site file
  LM:QL.CUSTOMER-SITE;LMLOCS.LISP.

  Each machine's pretty name should be specified in the file **LMLOCS** with the format:

  (*machine-official-name machine-pretty-name console-location*
  (*building-name floor-number*) *associated-machine*)

Step b26 After the site file information is listed correctly, test communication on the network. Type:

```
(hostat)
```

to print a list of all the machines on the network that are responding to the Lambda.

> NOTE: **hostat** will list an extra host called *LMI Amnesia* with the Chaosnet address 3412 octal. Do not remove this entry; see the section *Chaosnet Names, Addresses, and Nicknames* for more information.

If there are machines that do not seem to be responding, try **hostat** again, this time giving the name of the unresponsive machine. Type:

(hostat *machine-name***)**

If a machine still does not respond:

- Check Chaos addresses in **LM:QL.CUSTOMER-SITE;HOSTS.TEXT.**
- Check the machine's own site files.
- Check **hostat** from the other machine's console.
- Check the Ethernet cables and transceivers.
- If there seem to be actual network problems, call LMI for assistance.

3. Procedure C

MAKING MINOR CHANGES IN THE LAMBDA'S SITE INFORMATION

Use this procedure to make small modifications in the Lambda's site information, such as changing printer definitions or adding new non-LMI machines to the host table.

It is assumed that all Lambdas are already customized and are able to communicate on the Chaosnet. If a new Lambda has just arrived at the site, use **Procedure A** or **Procedure B** for the new Lambda.

If the Lambda is a 2×2, use this procedure for the processor in slot 0. Both sides of the 2×2 boot from the same band, so it is only necessary to install site information one time.

If there is more than one Lambda at the site, you need not repeat this entire procedure for each machine. When one of the Lambdas has the correct site information, copy its band to the other Lambdas via the Chaosnet. See Section 3.4

The actual changes to the site files can be made by following Steps **c1** through **c4** of this procedure, or by using the ZMACS Editor. See the *LISP Machine Manual* for more information.

After the changes have been made, the files must be compiled and installed into the load band of each machine on the network. Follow Steps **c7** through **c19** of this procedure.

3.1 Using SITED to Edit Files

If you wish to use the SITED editor to alter your site files:

Step #

Step c1 Start SITED by typing:

```
(sited)
```

The SITED program should print a welcome message, then the `SITED command>` prompt. *Slanted text* indicates user input.

Step c2

Type:

`SITED command>` *readfiles* RETURN

and answer **yes** to the question "Do you really want to proceed?"

Step c3 SITED then asks you to specify a directory. Type (don't forget the semi-colon):

`sys:site;` ⟨RETURN⟩

Step c4 A number of SITED commands can be used to change the existing site information. For example:

- The **`timezone`** command will change the **`:timezone`** site option.
- The **`lmprettyname`** command will change the name of the Lambda as it is printed in the herald.
- The **`textprinter`** and **`bitprinter`** commands will change the default text and bit-array printers for the site.
- New non-LMI machines can be added to the host table. Use the **`interactive`** command, then type **m** to answer questions about the new machine.

See Appendix B for a complete listing of SITED commands.

For on-line help:

- Type **?** at the SITED command level for a complete listing of SITED commands, or
- Type ⟨HELP⟩ at the SITED command level for general documentation on using the SITED program.

Step c5 After the changes have been made, type:

`SITED command>` *checkinfo* ⟨RETURN⟩

The site editor may discover omissions in your site files. For example:

```
The PRETTY-NAME property for OURSITE-
LAMBDA-A doesn't have any value
specified for it.

Should it be set to the default value
of "Oursite Lambda A" ? (y or n)
```

If you type **n** for "no", go back to the SITED command level to make the appropriate changes, or edit the file using the ZMACS Editor.

After the site information has been determined to be complete and correct, the following will be displayed:

```
No errors found in the information.
To generate new site files, use the
WRITEFILES command.
```

Step c6 Create a new set of site files. Type:

SITED command> *writefiles* ⟨RETURN⟩

SITED then asks you to specify a directory for writing the new site files. Type:

sys:site; ⟨RETURN⟩

and exit SITED by typing:

SITED command> *quit* ⟨RETURN⟩

Answer **yes** to the question "Do you really want to quit SITED?"

3.2 Compile Site Files

Step c7 After the changes to the site files have been completed, compile the new site files by typing:

```
(make-system 'site :compile
                   :no-reload-system-declaration :noload)
```

and answer **yes** to all questions.

Step c8 Check the new files by typing:

```
(update-site-configuration-info)
```

and use tests appropriate to the changes you have made.

3.3 Install Site Files into a New Band

Step c9 After you have determined that the new files are correct, they can be installed into a new band.

Cold-boot the machine. To do this, press both Control keys, both Meta keys, and the Rubout key, all at the same time. Cold-booting should take about five minutes.

Type:

CTRL-META-CTRL-META-⟨RUB OUT⟩

CAUTION From this point until Step **c13**, do not enter SITED, ZMACS, or any window other than LISP Listener 1, and do not type anything that is not listed as an instruction. If this should happen, or if anything else goes wrong, reboot the machine before you proceed.

Step c10 Log in by typing:

```
(login 'lispm t)
```

Step c11 Print a list of the Lambda's disk partitions. Type:

```
(print-disk-label)
```

Each line printed describes a disk partition: its name, starting block, size, and a comment describing its contents. An empty string **("")** indicates that the partition is empty.

Identify the partition known as the *current band*. This partition is marked with an asterisk (*) and starts with the letters **LOD**. Write the name of this band on paper so it will not be forgotten.

Choose a disk partition for saving the LISP environment that contains the new site information. This partition:

- Should be an empty partition whose name starts with the letters **LOD** (e.g. **LOD1**, **LOD2**, **LOD3**, etc.).
- Should NOT be the current band.

Step c12 Type:

(update-site-configuration-info)
(disk-save *pppp***)**

where *pppp* is the name of the disk partition that you chose in Step **c11**.

You will be asked, "Do you really want to clobber partition *pppp*?" Answer **yes**.

You will be asked to type a new comment for the herald. Type:

vvv.vv mm-dd-yy ⟨RETURN⟩

where *vvv.vv* is the system version number (e.g. 102.92) and *mm-dd-yy* is the date.

You will be prompted for a new comment for the disk label. Delete the current comment by positioning the cursor at the beginning of the comment and typing `CTRL-K`. Enter the disk label by typing the version number and date just as you did for the herald.

It will take about 10 minutes to save the new band into the disk partition.

Step c13 Now set the Lambda to use the new band when it is booted.

Type:

(si:.set-current-band "*pppp*")

where *pppp* is the four-letter name of the disk partition you typed in step **c12**.

Step c14 The new band is now permanently installed. Reboot the Lambda; if it is a 2×2, reboot both sides.

Type:

`CTRL-META-CTRL-META-`(RUB OUT)

3.4 Copy New Band to Other Lambdas

The LISP band you have just set can be copied to all the other Lambdas at the site. Follow the procedure given in Step **c15** through **c19** for each machine in turn.

> NOTE: The slot 0 and slot 4 processor of a Lambda/2×2 boot from the same band; use this procedure only for the slot 0 side of a 2×2.

Step c15 First obtain the following information:

- The octal Chaosnet address of the Lambda that has the new band. To get this, type on its console:

  (format nil "~O" chaos:my-address)

  This returns the Lambda's octal Chaosnet address in a character string.
- The name of the current band on that Lambda. (See Step **c11**.)

Write these two things on a piece of paper.

Step c16 Look at the disk partitions of the Lambda that is to receive the new band.

Type **on the Lambda that is to receive the new band:**

```
(print-disk-label)
```

Choose a disk partition on this Lambda to receive the new band. This partition:

- Must have a name starting with the letters **LOD** (e.g. **LOD1, LOD2, LOD3,** etc.)
- Must NOT be the current band. (See Step **c11**.)

Step c17 Type on the Lambda that is receiving the new band:

```
(array-initialize chaos:routing-table-type :ethernet)
(si:receive-band "chaos|xxxx" "pppp" "qqqq")
```

where:

- *xxxx* is the octal Chaos address (which you wrote down in Step **c15**) of the Lambda with the new band installed.
- *pppp* is the name of the current band for the first machine (set in Step **c13**).
- *qqqq* is the name of the empty disk partition of the receiving machine (chosen in Step **c16**).

This should take about half an hour. Do not use either Lambda during this time.

Step c18 Type:

```
(si:set-current-band "qqqq")
```

where *qqqq* is the same as in Step **c17**.

Step c19 Reboot the Lambda that has received the new band; if this Lambda is a 2×2, reboot both of its processors. (Use **CTRL-META-CTRL-META-⟨RUB OUT⟩** to boot.)

4. Procedure D

CUSTOMIZING UNIX SITE INFORMATION IN A LAMBDA/PLUS

Use this procedure to customize the site information of a UNIX system that is part of a Lambda/Plus or Lambda/2×2/Plus.

Before beginning this procedure:

- Choose names and Chaosnet addresses for each new UNIX to be installed. Follow the guidelines given in the section *Chaosnet Names, Addresses, and Nicknames.*
- Be sure that the LISP site files have all the correct site information, including the names and Chaosnet addresses of the UNIX systems to be installed. If the site files do not have this information, use **Procedure B** to enter it.

> NOTE: Check the site file **SYS:SITE;HOSTS.TEXT** to be sure the correct names and addresses have been entered into the file.

When this has been done, use the following procedure for each new UNIX to be installed. If the Lambda is a 2×2/Plus, work from the LISP processor in slot 0.

4.1 Preparation

Step #

Step d1 Obtain the following information:

- The official name of the LISP host. This can be obtained by typing to a LISP Listener:

  si:local-host-name RETURN

- The octal chaos address of the LISP host. This can be obtained by typing:

  (format nil "~8R" chaos:my-address)

  The octal address will be returned as a character string.

- The new official name and octal chaos address for the UNIX side of the Lambda/Plus. The file **SYS:SITE;HOSTS.TEXT** should have a line with the format:

  HOST *uuuuu*, **CHAOS** *xxxxx*, **USER, UNIX, NU,** [*nicknames*] where:

uuuuu is the UNIX's official name,

xxxxx is its octal Chaos address.

Step d2 Type:

⟨SYSTEM⟩ **U**

and log in to UNIX as **root**.

Then, as a safeguard, rename the old UNIX host table. Type:

```
cd /usr/lib/chaos/tables ⟨RETURN⟩
mv hst-customer hst-customer-old ⟨RETURN⟩
```

4.2 Create Site File

Step d3 UNIX is case-sensitive; everything in the following lines should be upper case except the words "echo" and "hst-customer".

Type:

```
echo "NET CHAOS, 7\⟨RETURN⟩
HOST UUUUU,CHAOS XXXXX,USER,UNIX,NU,[]\⟨RETURN⟩
HOST LLLLL,CHAOS YYYYY,USER, LISPM,LISPM,[]" > hst-customer ⟨RETURN⟩
```

where:

UUUUU is the official name for the UNIX,

XXXXX is the octal Chaos address of the UNIX,

LLLLL is the official name of the Lambda,

YYYYY is the Lambda's octal Chaos address.

Step d4 Copy the file (just in case). Type:

```
cp hst-customer hst-mini ⟨RETURN⟩
```

then compile and install the mini site file. Type:

```
make ⟨RETURN⟩
```

An error message at this point probably indicates a typing error somewhere in Step **d3**. Return to that point, enter the line correctly, and continue.

Step d5 When there are no errors, type:

echo "*UUUUU***" > /etc/sysnames** ⟨RETURN⟩

where *UUUUU* is the official name of the UNIX.

4.3 Copy File

Step d6 Bring down UNIX multiuser by typing:

```
su ⟨RETURN⟩
sync ⟨RETURN⟩
sync ⟨RETURN⟩
kill 1 ⟨RETURN⟩
sync ⟨RETURN⟩
sync ⟨RETURN⟩
```

Reboot the Lambda, both LISP and UNIX.

As UNIX is booting multiuser, it should print:

I am host *UUUUU* **with chaos address** *XXXXX*.

then display the **:login:** prompt.

Step d7 Type ⟨SYSTEM⟩ **L** to bring up a LISP Listener, then log in to LISP and type:

(hostat "*UUUUU***")**

where *UUUUU* is the name of the UNIX.

If **hostat** lists the UNIX correctly, type:

(fs:copy-file "sys:site;hosts.text"
"*UUUUU***://usr//lib//chaos//tables//hst-customer")**

to copy the host table to UNIX.

4.4 Check and Compile

Step d8 Type:

```
⟨SYSTEM⟩ U
```

and log in to UNIX as **root**.

Type:

```
cd /usr/lib/chaos/tables ⟨RETURN⟩
more hst-customer ⟨RETURN⟩
```

to examine the host table just copied from LISP.

Step d9 Compile and check the new host table. Type (still to UNIX):

```
make ⟨RETURN⟩
```

Step d10 Bring down UNIX multiuser (See Step **d6**) and reboot the Lambda, both LISP and UNIX.

The UNIX site files are now completely installed.

Log in to UNIX as **root** and type:

```
hostat ⟨RETURN⟩
```

The other hosts at the site should respond to the new UNIX.

Step d11 Type:

```
/etc/setnettime ⟨RETURN⟩
```

and answer the question to set the time for your site.

lo "input patbo

Appendix A

The following is a table of Chaosnet subnet numbers, and the corresponding ranges of Chaosnet addresses. All numbers are octal.

Subnet	**Address Range**
0	0 (This is the Chaos broadcast address)
1	400 to 777
2	1000 to 1377
3	1400 to 1777
4	2000 to 2377
5	2400 to 2777
6	3000 to 3377
7	3400 to 3777 (note: 3412 is reserved)
10	4000 to 4377
11	4400 to 4777
12	5000 to 5377
13	5400 to 5777
14	6000 to 6377
15	6400 to 6777
16	7000 to 7377
17	7400 to 7777
20	10000 to 10377
21	10400 to 10777
22	11000 to 11377
23	11400 to 11777
24	12000 to 12377
25	12400 to 12777
26	13000 to 13377
27	13400 to 13777
30	14000 to 14377
31	14400 to 14777
32	15000 to 15377
33	15400 to 15777
34	16000 to 16377
35	16400 to 16777
36	17000 to 17377
37	17400 to 17777
40	20000 to 20377
41	20400 to 20777

42	21000 to 21377
43	21400 to 21777
44	22000 to 22377
45	22400 to 22777
46	23000 to 23377
47	23400 to 23777
50	24000 to 24377
51	24400 to 24777
52	25000 to 25377
53	25400 to 25777
54	26000 to 26377
55	26400 to 26777
56	27000 to 27377
57	27400 to 27777
60	30000 to 30377
61	30400 to 30777
62	31000 to 31377
63	31400 to 31777
64	32000 to 32377
65	32400 to 32777
66	33000 to 33377
67	33400 to 33777
70	34000 to 34377
71	34400 to 34777
72	35000 to 35377
73	35400 to 35777
74	36000 to 36377
75	36400 to 36777
76	37000 to 37377
77	37400 to 37777
100	40000 to 40377
101	40400 to 40777
102	41000 to 41377
103	41400 to 41777
104	42000 to 42377
105	42400 to 42777
106	43000 to 43377
107	43400 to 43777
110	44000 to 44377
111	44400 to 44777

112	45000 to 45377
113	45400 to 45777
114	46000 to 46377
115	46400 to 46777
116	47000 to 47377
117	47400 to 47777
120	50000 to 50377
121	50400 to 50777
122	51000 to 51377
123	51400 to 51777
124	52000 to 52377
125	52400 to 52777
126	53000 to 53377
127	53400 to 53777
130	54000 to 54377
131	54400 to 54777
132	55000 to 55377
133	55400 to 55777
134	56000 to 56377
135	56400 to 56777
136	57000 to 57377
137	57400 to 57777

APPENDIX B, A LIST OF SITED COMMANDS, WILL BE AVAILABLE SHORTLY.

IN THE MEANTIME, PLEASE CONTACT YOUR CUSTOMER SERVICE REPRESENTATIVE AT THE LMI LINE IF YOU HAVE ANY QUESTIONS ABOUT SITED COMMANDS.

Lambda Tape Software

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

This document corresponds to LMI Release 2.

Any customers having specific suggestions for the next magnetic tape software release, or comments on this documentation, should address them to:

Dr. Sarah Smith
LMI
1000 Massachusetts Avenue
Cambridge MA 02138
(617) 876-6819

Table of Contents

Introduction

This manual documents the tape software for the LMI Lambda series of LISP machines. This software is a workable magnetic tape system with all the needed functionality for dumping and retrieving files and disk partitions.

This document is intended for both users of the tape software and programmers who want to incorporate tape functions into their programs. To use the high-level facilities provided, you will need to know basic LISP syntax and how to use the LISP Listener. Documentation conventions are those of the *LISP Machine Manual.*

The first part of this document describes the tape software functions for copying files onto tape, getting them off tape, and generally providing the services that any operating system tape utility program would provide. (The major exception to this is a true file backup system, which is really a separate facility and which is still being implemented.)

The second part of this document describes the programmer's view of the tape software. Most of the interface to the tape software is at the stream level; in fact, no hardware command-oriented documentation is supplied. The `''LMFL''` header format is documented (in section 2.6, page 13) in case a need arises for a tape to be read or written by a non-LMI machine. (For example, there is the **lmtar** for LMI-supplied UNIX systems.) Except for obtaining streams to tapes, the programming interface follows LISP Machine stream conventions as closely as possible.

General Information

Lambda and CADR tape software can each read and write tapes for the other; however, since the formats are not completely compatible differences are noted in this document when appropriate.

All keyword arguments (those arguments that appear after the **&key** symbol in the argument list) are optional. *unit* arguments, which are for selecting one of multiple drives on a controller, need not be specified. The term "generic tape function" refers to functions prefixed by **mt-**; these are the only ones that should be called directly by a user, as they select the correct special operations according to the type of tape drive being used.

When a directory is said to be specified in "internal format," this means the same format as that found in pathname objects: single-level directories are strings, while multi-level directories are lists of strings:

```
RPK;  => "RPK"          RPK.DOC;  => ("RPK" "DOC")
```

This internal format is exactly the same as the directory element of a pathname object. This scheme is used for labeling the files on the tape so that they can be easily copied from or copied to (restored to) any of the supported file system types (LISP Machine, UNIX, Twenex, Tenex, ITS, and VMS) without having to actually translate the file name syntax of the system from which the file was copied.

In Release 2, keywords (such as the symbol **:byte-size**) do *not* have to be preceded by a quote, because keyword symbols are self-evaluating. Don't worry about changing old software, the quotes will not affect its ability to run under Release 2. The colons, however, are always needed. This document follows the more modern convention; any examples that do not quote keywords in a context in which they previously would have to have been quoted must be read more carefully by users of earlier releases.

1. User Tape Functions

This chapter documents functions that *can* be used from programs, but are usually used as commands to LISP through a LISP Listener window.

Functions for putting a tape drive offline, rewinding the tape, and positioning the tape to append more files to it are documented in section 2.2, page 11.

1.1 Selecting and Initializing the Tape Device

Before you use a tape drive you need to allocate it. This reserves use of the drive to the current process. To do this, use the same **open** function that you would use to open a file. For example:

```
(open "half-inch-tape:")       ;note colon

                               ;or...

(open "quarter-inch-tape:")       ;note colon
```

This will return a shared device object,

```
#⊂shared-device-style-object⊃#
```

that you will need to close when you are done.

Alternatively, you can allocate the device using **with-open-file**. This will allocate the tape drive, execute the *body* and deallocate the device automatically when it's done.

```
(with-open-file (s "half-inch-tape:")
...
            body...
)
```

Note that these functions merely reserve the particular device for your use; in order to do anything with the device you should use the generic tape functions described in the following section.

On the CADR, tape operations can be directed to only one controller. However, the Lambda can use both the quarter-inch cartridge drive (via the SDU) and an industry standard drive (usually a Cipher tape drive) through the TAPEMASTER controller connected to the Multibus. When the Lambda boots, the default tape device is whichever one is actually present; if both tape devices are available, the quarter-inch cartridge drive is selected by default. To change the default drive type, use one of the following functions:

fs:use-quarter-inch-tape Function

Changes the mode of the generic tape functions (the **mt-** functions) to use the Quarterback 1/4" tape drive.

fs:use-half-inch-tape Function

Changes the mode of generic tape operations to use the TAPEMASTER controller with a half-inch tape drive, either Cipher or Kennedy.

Sometimes the controller may end up in an inconsistent state, and thus appear to not be working. Before investigating a hardware problem, try calling this function:

fs:mt-reset Function

Resets the tape controller according to the value of **fs:quarter-inch-tape-mode**. Use this function only to unwedge the tape controller. In general, if you want to abort a tape operation, use **CTRL-⟨ABORT⟩** to ensure that the tape controller isn't left in a strange state.

The variable **fs:*quart-paranoia-margining*** (see page 12) may be set if the quarter-inch drive is giving poor results. However, you should do this only as a last resort.

1.2 Tape Utilities

This section documents higher-level functions for using the tape drives. Such functions are self-contained, report what they are doing, and may query the user. Mostly, they copy various kinds of data (files, disk partitions) back and forth between the tape and disk (usually, a file system). If you wish to use some facet of these functions for your own specialized application, please refer to the latter section of this document, which describes the lower-level access to the tape software and various calling and storage conventions used for storing files on tape.

fs:mt-write-partition *partition* &optional (*unit* **0**) Function

Writes *partition* on *unit* to tape; then writes an **eof** mark and spaces backwards to it. This ensures that the last partition is in fact followed by another **eof**. Note that the *unit* argument refers to the unit number of the *disk* on which the partition is stored; the partition is always written to tape unit 0. If the partition to write is not on your local machine, then *unit* must be specified as a string giving the name of the machine. For example:

```
(fs:mt-write-partition "lod5" "lam3")
```

The function **fs:mt-write-partition** is built on top of **si:copy-disk-partition**. You can specify **"mt"** to be used as a source or a destination with this function.

To restore a partition from disk, use **fs:restore-magtape** (see below).

fs:restore-magtape &key (*host* **si:local-host**) (*query* **t**) *transform directories* *copy-options tape-options* Function

Restores the files (possibly some disk partitions) from the tape onto the file system of *host* (by default, the local machine). If *query* is **t** (the default), then you are asked, file by file, whether to restore the file onto the disk. Answering `'P'` to the query turns off querying for the remainder of the files on tape, except for partitions, which are considered a special case. If a partition is encountered, you will be queried for a place to copy the partition (this may even be a band on another machine).

If *directories*, a list of directories in internal form, is supplied, only files on tape from those directories (and their subdirectories) are considered for restoring.

The *copy-options* argument, passed to **fs:fs-copy-file**, is usually not needed. The *tape-options* argument is supplied to **fs:make-mt-file-stream** (see page 9); this argument is usually not used, either.

If *transform* is supplied, it is either a function of five arguments or a symbol whose **fs:tape-restore-transform** property is a function, as described below.

The transform function's arguments are the *host* (which is a host object, not a string), the *directory* (in internal format), the *name*, the *type*, and the *version* of the file which is ready to be restored. The function returns either a pathname (which then becomes the name of the file to restore the file from tape onto disk) or **nil**, meaning to skip the file and go on to the next one. When a *transform* is supplied and *query* is **t**, the filename displayed in the prompt is the one that the transform returns.

There are currently two predefined transforms. One is called **:ask-and-default**, the other is **:standard-sys**. The first asks you what translation to use every time it encounters a new file directory on tape. When it asks you for a transform (note the question mark "?"prompt], type in a directory to use; the host will default to the one used in the prompt. You should always supply the directory component. For example:

```
Translation for the directory FRED: FOO; ? BAZ:
```

will translate the file `FRED: FOO; DOC.TEXT` to `FRED: BAZ; DOC.TEXT`, and

```
Translation for the directory FRED: L.PATCH; ? SRC: L.S98.PATCH:
```

`FRED: L.PATCH; SYSTEM-98-23.QFASL` to `SRC: L.S98.PATCH; SYSTEM-98-23.QFASL`

Notice that you can supply a different host from the default one.

When *query* is **t**, you should supply the correct directory for translation even when you are not planning to restore the first file from that directory. If you want **:ask-and-default** to use a new set of translations (i.e., you want it to forget the ones given it before), **setq** the variable

fs:*ask-per-directory-defaults* to **nil**.

The second predefined **:transform**, **:standard-sys**, should be used only when restoring a tape of system source files, when your site's **sys:** host translations are not of the form `SYS: FOO; => L.FOO;`. **:standard-sys** expects that all the files on the tape were dumped from directories under a **L:** or **NL:** hierarchy. (This transform should not usually be used by customers unless documentation that comes with a release source tape says to.)

Here is an example of a user defined tape transform function:

```
;;; Change X; to TAPE.X; on the mounted tape for all directories; also,
;;; change TXT to TEXT, and LSP to LISP, for those types. Finally, make,
;;; all versions :NEWEST and don't restore files from the SYSTEM
;;; directory.
(defun (:property :my-xform fs:tape-restore-transform)
       (host dir name type version)
       version ; this argument not used
   (if (equal dir "SYSTEM") () ; ignore this file
     (fs:make-pathname ':host host
       ':directory (if (stringp dir) (list "TAPE" dir)
                       (cons "TAPE" dir)) ; Multilevel directory here
       ':name name ':version ':newest
       ':type (cond ((string-equal type "TXT") "TEXT")
                    ((string-equal type "LSP") "LISP")
                    ((t type))))))
```

After evaluating the above form, **:my-xform** can be given as the **:transform** argument to **fs:restore-magtape**.

fs:fs-copy-file *from to ...* Function

Use this function to write one file to tape; specify **mt:** as the *to* argument. Currently, you may not specify a new filename for the file on tape. **mt:** cannot be specified as the *from* argument; use **fs:restore-magtape** to accomplish this (See page 5.) Ellipses indicate keywords that exist but are rarely needed or used.

This function is considered obsolete for all purposes except tape applications.

fs:copy-files *files to ...* Function

This function takes the same options as **fs:fs-copy-file**. *files* should be a list of filenames to copy. Ellipses indicate keywords that exist but are rarely needed or used.

This function is considered obsolete for all purposes except tape applications.

fs:copy-directory *from to* &key *copy-only selective since* (*copy-subdirectories* **t**) Function

(This function, exists only in the magnetic tape software. It is like the regular system function **copy-file**, except that you have more control over which files are copied.) The arguments *from* and *to* can be any pathnames that name files; *to* takes its defaults from *from.*

If *copy-only* is supplied, its value names what versions of files will actually get copied. The supplied value, then, can be **:wild** (the default), **:newest**, **:oldest**, or a number. If *selective* is **t**, the user is queried about each file to be copied. If *since* is a date (meaning a string that can be parsed as a universal time), then only files with a creation date after that are copied. *to* can be **"mt:"** for copying to tape.

If *copy-subdirectories* is non-**nil**, then files in subdirectories are copied as well; the target directory for those files is a subdirectory under the target directory with the same name. In older versions of the software, *copy-subdirectories* would not propagate the exact specifications of *from* when recursing; as of Release 2 (System 99), it will take name, type, and version defaults from the superior directory unless the value of *copy-subdirectories* is **:wild**, in which case the old behavior will occur, namely, to copy *all* files in the contained subdirectories.

fs:magtape-list-files &optional (*out-stream* ***standard-output***) (*unit* **0**) Function

Lists the files dumped on the currently mounted tape, printing out onto *out-stream* the byte size, creation date, and file name. Structured directories are printed out as lists.

In the following four functions, the rest argument, *options*, gets passed to **fs:make-mt-file-stream** (see page 9).

fs:print-magtape &rest *options* Function

Copies the contents of the tape, until the **eof** marker, to ***standard-output***. This is useful for debugging and trying to figure out foreign tape header formats.

fs:copy-magtape-file *fn* &rest *options* Function

Like **print-magtape**, but sends the output to the file **fn** (which will be a character file). Somewhat like the UNIX command **dd**. No character set translation is done.

The following two functions do not use header conventions. They deal with unlabeled tapes using the ASCII character set. They are intended as quick ways to exchange files between a Lambda and another computer which is not accessible via the Chaosnet, but which does have a tape drive.

fs:print-ascii-magtape &rest *options* Function

Copies the contents of the tape until the **eof** to ***standard-output***, translating from ASCII to the LISP Machine character set.

To direct the output to a file:

```
(with-open-file (s file :direction :output)
  (let ((standard-output s)) ; directs standard-output to
    (fs:print-ascii-magtape))) ; this file "s".
```

fs:write-ascii-magtape *fn* &rest *options* Function
Writes file *fn* to tape; translating to the ASCII character set. No header is written.

1.3 Other Tape Formats

The following functions allow you to read tapes from other computers onto the Lambda. Currently the ANSI, and TOPS-20 formats are supported.

tops20:restore-tape &optional &key *to-host query tape-block-size file-opener* Function
Restores a tape written using the **dumper** program under the TOPS-20 operating system. *tape-block-size* is in units of PDP-10 words (5 bytes) and should usually be a multiple of 1024. *file-opener* is a function to receive a TOPS-20 file specification in the form of a string and return either NIL or a file object.

tops20:*default-tape-block-size* 5120 Variable
The default for the *:tape-block-size* argument for the above function.

tops20:*unhandled-file-types* ("QFASL" "EXE" "BIN") Variable
Filenames of these types are skipped over.

ansi:restore-tape &optional &key *file-id-append skip-if-exists query verbose* Function
This restores an ANSI labeled tape written according to American National Standard X3.27-1978. For example, the **copy** command under VAX/VMS produces such a tape. Directory name information is not preserved; therefore the *file-id-append* argument is provided which defaults to "LM:TMP;". The value for *Verbose* is either **t**, **1**, **2**, or **3**. **t** and **1** provide a printout for each file restored; **2** provides information for each ANSI label and header processed; **3** gives information about each record processed. This last may be useful for debugging purposes.

ansi:*record-format-handlers* an a-list Variable
Presently the "D" (variable record) and "F" (fixed record) formats are handled.

2. Programming Information

This chapter describes more user functions that interface to the tape software; most of these functions are meant to be called from programs. There are functions for obtaining streams to the tape device, functions for controlling tape-specific operations, and condition names for handling tape-related errors.

The main interface for getting data to and from the tape is the use of *streams*, and the standard ZetaLISP stream functions. Special functions are used to obtain tape streams; the model of tape storage is not very compatible with a "file system" model of storage, so pathnames are not the recommended interface to tape. The user functions that deal with files (see the utilities that are documented starting on section 1.2, page 4) should provide most of the functionality needed for transfer between tape and file systems. If you really need to use the tape as a file system for an application, refer to section 2.1, page 9, which describes the conventions for storing files on tape.

2.1 Obtaining Streams

Streams made with the following two functions can be read or written with the standard buffered stream operations (see pages 476-77 of the *LMM*) for "record access". (The actual buffer objects returned are **RQB**s, a system structure normally used for disk I/O.) However, they are buffers all the same, and simple array functions such as **aref** can be applied to them. Of course, all standard stream operations work on these streams.

There are two functions for creating magtape streams. One, **fs:make-mt-stream**, is for using the tape in "raw" mode; the other, **fs:make-mt-file-stream**, creates streams that structure the data with headers so that **fs:restore-magtape** (see page 5) and other functions that use the tape as a file system can read the data back in. All the arguments to these functions are keyword arguments:

fs:make-mt-stream &key *direction characters byte-size* (*unit* **0**) *ibm-mode* (*record-size* ***default-record-size***) Function

Makes (and returns) a magtape stream that uses the tape without trying to use any file or operating system format – no header is written or parsed. The keyword arguments are the same as the function **fs:make-mt-file-stream** below.

fs:make-mt-file-stream &key *direction characters byte-size* (*unit* **0**) *ibm-mode* (*record-size* ***default-record-size***) *plist* (*format* **:mit**) Function

Makes (and returns) a magtape stream that acts like a file stream. It will have an associated pathname, author, creation-date, etc., if you pass that information along in the *plist* argument. If *characters* is **:default**, the value is determined from the plist passed in, and the **:byte-size** will default after checking the character argument, if it is not passed explicitly.

To really produce a file stream, callers should make sure the *plist* argument has **:directory**, **:name**, **:type**, **:version** (which should be a number), **:creation-date**, **:byte-size**, and **:characters** properties, though you may add more. (The **:author** property, for example, should usually be supplied as well.) If the **:byte-size** and **:characters** properties appear on the *plist*, then not passing those parameters as keyword arguments will do the right thing.

These are the keyword options used by the two functions discussed above.

:byte-size This can be a number (8 or 16 on the Lambda, also 1, 2, or 4 on the CADR) or **:default**, which is the default value.

The symbol **:default** means different things depending on the direction of data transfer. When *direction* is **:output**, it means 8 for a character stream and 16 for binary stream. When *direction* is **:input**, the actual byte size is the one stored in the property list in the header, if the function is **fs:make-mt-file-stream**. Otherwise, the default byte size will be 8 for a character stream, or 16 for a binary stream.

Currently, the decoding of Common LISP stream-element type arguments is not supported.

:direction This is either **:input** or **:output**. Probe opens, where the *direction* is **:probe** or **nil**, are not allowed and signal an error.

:characters This argument can be **t** or **nil** for **fs:make-mt-stream**. For that function, the default value is **t**.

An additional value accepted by **fs:make-mt-file-stream** is **:default**, which is the default value for that function. On input, using **:default** as the value causes the **:characters** property in the header plist on the tape to be used; on output, **:default** behaves the same as **t**.

:unit The tape unit number to use; usually 0.

:record-size

This variable controls how large the block size will be for the stream. It defaults to **fs:*default-record-size*** (see page 12).

:ibm-mode This keyword is used on the CADR only; it is ignored on the Lambda. By default, **nil**. An argument of **t** causes the "IBM mode" bit to get set on the Wesperco controller on write or read operations.

Both functions return the **fs:end-of-tape** error if the end of tape is reached. Although you are allowed to make streams in the **:output** direction on tapes with the write ring removed, the **fs:write-only-tape** error is signalled as soon as you try to write to the tape.

The actual flavor of the stream is determined by the *characters* and *direction* arguments. The device selected also is taken into consideration; on the CADR, this is always the same (the Unibus controller); on the Lambda, this is controlled by the selected tape device through the variable **fs:quarter-inch-tape-mode** (see page 12).

The current version of the software can read and write only file streams using **:mit** format, which is the type used by the LISP Machine. There are also some miscellaneous, unsupported functions to read other formats, but they do not present a stream interface.

The **:open** method for **mt-filehandle** (the equivalent of a tape pathname, but not as complex as a real file pathname) accepts the usual **open** keywords, as well as **:defaults-from-stream**, which, when supplied with a stream as the value, will put the relevant properties from that stream onto the *plist* argument of the function **fs:make-mt-file-stream**.

The **:close** operation on a magtape file input stream advances the tape to the next record after the **eof** mark. Thus, **fs:make-mt-file-stream** can be called again to get the next file.

2.2 Tape Movement

The following tape-drive controlling functions have default *unit* arguments of 0 and default *ntimes* arguments of 1 unless otherwise noted.

fs:mt-rewind &optional *unit* — Function

Rewinds the tape mounted on *unit*.

fs:mt-offline &optional *unit* — Function

Brings *unit* offline.

fs:mt-unload &optional *unit* — Function

A synonym for **fs:mt-offline**.

The following six functions are usually called by programs that are reading or writing the tape when viewed as a stream of records.

fs:mt-space &optional *unit* (*ntimes* **1**) — Function

Spaces forward *ntimes* record(s) on *unit*.

fs:mt-space-rev &optional *unit* (*ntimes* **1**) — Function

Spaces back *ntimes* record(s) on *unit*.

fs:mt-space-to-eof &optional *unit* (*ntimes* **1**) — Function

Spaces forward past the next filemark. If *ntimes* is more than 1, then that many files are skipped over.

fs:mt-space-rev-to-bof &optional *unit* (*ntimes* **0**) — Function

Spaces back to the beginning of this file, or to the *ntimes*th file if *ntimes* is greater than 0. This function returns prematurely if the beginning of the tape is encountered before all the files are skipped.

fs:mt-space-to-append &optional *unit* Function

Searches for two **eof** marks and places the tape header over the second one, so that writing a stream will add more files to the end of the tape.

fs:mt-write-eof &optional *unit* Function

Writes an **eof** mark. After the last file on tape, there should be two **eof** marks; one is written when the MT file stream is closed; and the other must be written by the user (or the application program).

2.3 CIPRICO TAPEMASTER Specific Functions

fs:tapemaster-initialize Function

Initializes the TAPEMASTER controller. If you don't initialize the controller, tape commands will time out, and you will be put into the error handler.

fs:tm-print-unit-status &optional *unit* Function

Prints out the status of the tape drive *unit* connected to the controller.

2.4 Variables

fs:*default-record-size* Variable

Initially 10000 (octal). Old (pre-release 1.2) Lambda tapes were written with this variable set to 2000 (octal), but you can still read these tapes without changing the value.

fs:quarter-inch-tape-mode Variable

Determines which tape device will be used by generic functions. **t** represents the quarter-inch drive, **nil** the TAPEMASTER 1/2" drive.

This variable should not be set directly. Use the **fs:use-*type*-tape** functions (see page 4) to change the tape drive to use.

fs:*quart-paranoia-margining* Variable

Setting this variable to **t** will reset the drive before some operations to ensure the correct state of the device and in other cases to allow a **process-sleep** period so that the device can fully react to a given command and be ready for the next command. It should not be used unless problems occur. Call LMI first.

2.5 Conditions

fs:tape-error (error) Condition flavor

This is the flavor of condition that is signalled when there is trouble with a tape operation.

fs:end-of-tape (tape-error) Condition

Signifies end of tape. This is an especially useful condition name in user programs when used with **condition-case**.

fs:write-only-tape (tape-error) Condition

Signalled by the output operations of tape streams when the write ring is not there. This error is *not* signalled by the tape output stream making functions.

Both tape error conditions handle the following operations, in addition to standard error condition operations. These keywords are also init options to the **fs:tape-error** condition flavor.

:unit Returns the offending tape unit.

:rqb Returns the **RQB**. This is usually not needed.

:ibm-mode **t** if writing in IBM mode; applicable to the CADR only.

:command A number; the meaning is hardware-dependent.

:byte-count

Intended byte count.

:density Usually **nil**. **t** means high density. The exact meaning is hardware dependent.

2.6 LISP Machine Tape Format (LMFL, :MIT)

The format used by the function **fs:make-mt-file-stream** is very simple: The first four characters are "**LMFL**". Then immediately following is a property list, printed and read with the package bound to **fs:** and the base (number radix) bound to ten. Then there is some padding with spaces; the next block starts the actual data. The length of the first block should be 1024 bytes, but it may be different if the tape was written with old software (Magtape 14 or before). The end of file is signalled, as usual, with an **eof** block.

A typical header might look like this. (The example has been changed for readability; the newlines are not actually there.)

```
(:characters t :byte-size 8 :creation-date 54583923
 :directory ("RPK" "LM") :name "HOSTAB-SERVER" :type "LISP"
 :version 17 :length 6002 :AUTHOR "RpK")
```

The properties may come in any order. **Note** that the data on the tape is stored in eight-bit bytes. If the file on the tape consists of characters (i.e. when a non-null **:characters** property appears), then the characters are in the LISP Machine character set. (Refer to the discussion of the character set in the *LISP Machine Manual.*) If the data is binary, the bytes are packed in "little-endian" order. For a file with a **:byte-size** of 16 the low eight bits will be encountered first. For a binary file tape made on the CADR whose byte size is less than eight, the first nibbles are the least significant ones.

Here is a list of the defined properties that the system functions understand. Sometimes **nil** may appear as `"()"` in the tape header. Programs written for non-LISP Machines should interpret both `"()"` and `"NIL"` as **nil**. You may add your own properties to tapes that you write yourself.

:directory The directory of the file, a string or list, given in the internal format.

:name The name of the file, a string.

:type The type of the file, usually a string, but sometimes **nil** if the file was dumped from a file system that does not require a type component (like LMFL), to be present.

:version The version. This is always a number, even though there are other allowable values for versions in pathnames.

:characters Whether the file contains characters or not.

:byte-size The file's byte size. This should be 8 for text files, or a power of two between 1 and 16 inclusive for binary files. However, the Lambda does *not* support byte sizes of 1, 2, or 4 (the CADR does).

:creation-date
The creation date of a file as a universal time, a number.

:author The file's author, given as a string.

:length The length of the file in bytes, a number.

The stream's file properties should not be accessed via the **:get** operations, but instead should send the message directly to the stream:

```
(send mt-stream :get :directory)        ; Wrong
(get mt-stream :directory)              ; Wrong
(send mt-stream :directory)             ; Right
```

The following properties are used when the "file" is actually a disk partition. (Unlike with the properties above, you can use **:get**, even on the **:name** attribute.)

:partition **t** if a partition, either **t** or **nil**.

:comment Description of the contents, a string.

:name Original partition band on disk, a string.

:size Size in blocks.

2.6.1 Lambda/CADR Format Discrepancies

Header Block Size

In old software, before the Lambda, the header block was as big as it needed to be: this was related to the length of the printed representation of the plist, plus a few extra characters for **"LMFL"** and spaces for ensuring that the **read** function did not cause the next tape block (which contains the beginning of the actual file data) to be read in. When the software was adapted for the Lambda, it was discovered that the (half-inch) controller did not handle irregular block sizes gracefully. So, now the header block is 1024 (decimal) characters, padded with spaces. A standard header block usually has about 200 characters of "real" data, so user-added properties can be added safely. Just don't go overboard. ZetaLISP software will read the header block correctly no matter when it was written; so this information is of interest to those who want to write programs to handle tapes made on the LISP Machine to run on other machines.

Package Problems

All symbols in the plist are supposed to be in the keyword package, but various changes in the package system have complicated getting certain properties off the plist. Therefore, some property symbols in tape headers written by older software may not be preceded by colons. This occurs with the following symbols: **:byte-size**, **:directory**, and **:length**. The **:byte-size**, **:directory**, and **:length** methods for **mt-file-streams** compensate for the problem.

Characters

Earlier versions of the software did not write the **:characters** property to the header. On input, the function **fs:make-mt-file-stream** (with direction **:input**) assumes for such tapes that if the **:byte-size** is 8, then it's a character file.

This means that if an eight-bit binary file (usually a press file) has been mistakenly restored as a text file, it will not work correctly if the machine to which it was copied is based on the PDP-10; as a text file, it will get stored in seven-bit bytes. If such a file is restored to a system that had to be accessed through a network (any other machine besides the LISP Machine with the tape drive), the file will be stored in eight-bit bytes, but unfortunately the LISP Machine character set translation will have been applied to the bytes in the file.

However, if the file was restored locally (onto the disk), then the easiest solution is to:

1. Invoke the ZMACS command **META-X Dired** on the containing directory.
2. Invoke the **.** (period) command (**Dired Change File Properties**) on the file. A menu will pop up with various file system properties of the file.
3. Click on the word **Yes** on the **''Characters''** line of the menu.
4. Click on **''Do It''** on the bottom of the menu.

LISP Index

Interprocessor Communication
The Extended STREAMS Interface

Published by LMI 1000 Massachusetts Avenue, Cambridge MA 02138 USA

Comments on this document and other documentation should be addressed to:
LMI
1000 Mass. Ave.
Cambridge MA 02139

Phone: (617) 876-6819

Principal writer of this document is Meryl R. Cohen.
Formatted with BoTEX.

Summary Table of Contents

Table of Contents

1. Introduction

The Extended STREAMS Interface facilitates interprocessor communication. This software allows you to share memory and program control between the Lambda LISP and UNIX (68010) processors and also between the two LISP processors on a Lambda 2x2. Communication between processors can be accomplished in several different ways:

- by sending information through the Chaosnet
- with the Share TTY feature
- using the shared memory area

Each of these three methods has its advantages and drawbacks. However, for applications involving a lot of data transfer, using the shared memory capabilities will be considerably faster.

This document covers all three methods of LISP/UNIX communication with emphasis on the shared memory techniques. The examples in the body of the text are for illustration only; there is an extended example included with your software in **SYS:EXAMPLES;STREAMS** that can be run online. You can find the text of this example in Appendix B.

Familiarity with the streams concept in general, Chaosnet, and UNIX and **C** programming will be necessary in understanding the following material. The *LISP Machine Manual*, as usual, is the best alternate source of information; it contains the definitive documentation on the Zeta-LISP implementation of Chaosnet and the machine subprimitives which are used in LISP/UNIX communication.

Changes in Release 2

The major changes to the Extended STREAMS software in Release 2 are additional UNIX functions that allow access to shared memory, and a new approach to devices in ZetaLISP. The UNIX material is covered in Section 4.2, "Shared Memory from UNIX"; the new device information is in Section 5.1, Devices from LISP.

2. Chaosnet

Chaosnet is the least direct connection between LISP and UNIX. Passing data and programs over Chaosnet involves the greatest amount of processor overhead, but allows you to communicate between processors that reside on separate busses. For instance, you can use Chaosnet to communicate between a Lambda LISP processor and another computer running UNIX. The various protocols that are available and user definable are useful tools for solving many problems and for initiating and coordinating high performance intra-bus transactions.

All functions are transparent with respect to bus configuration. Many symbolic Chaosnet contact names, such as **FILE** and **TIME**, are also transparent with respect to the operating system of the particular processor.

The following example shows how you would set up a call from the LISP processor over Chaosnet to the UNIX processor and request UNIX to run a program for you.

The **chaos:open-stream** function is useful for connecting to the UNIX **eval** server.

```
(defun simple-unix-eval (host command)
  (with-open-stream (s (chaos:open-stream host
                          (string-append "EVAL " command)))
    (format t "~&@c ~A~%" command)
    (do ((c))
        ((null (setq c (send s ':tyi))))
      (send standard-output ':tyo
            (selectq c
             ((#o12 #o15) #\return)   ;LISP to UNIX character
             (#o11 #\tab)             ; set translation.
             (t c))))))
```

The command to **eval** is given to the UNIX shell, for example:

```
(simple-unix-eval "unix-test-a" "ls //etc")  ;this produces the
                                             ; effect of ls /etc.

myhostname
passwd
hosts
...
```

Here is an example using both file-system protocol and **eval** protocol:

```
(with-open-file (s "unix-test-a://tmp//foo.c" ':out)
   (format s "#include <stdio.h>~
             ~%main()~
             ~%{ printf(/"Hello world.\n/");}~%"))

(simple-unix-eval "unix-test-a" "cc //tmp//foo.c")  ;this produces the
                                                    ; effect of cc /tmp/foo.c.
```

```
(simple-unix-eval "unix-test-a" "//tmp//a.out")    ;this produces the
                                                   ; effect of /tmp/a.out.
Hello world.
```

3. Share TTY

Share TTY actually uses the shared memory area set aside in Extended STREAMS, but you can take advantage of it using operations that look like regular stream operations, without worrying about any issues of "safe" versus "dangerous" memory addresses.

The simplest way to take advantage of this feature is with (SYSTEM) **U**. This will give you direct access to the UNIX system as another window reachable through the Lambda window system. You can use this window as you would any other UNIX connected terminal, yet still quickly switch back to your LISP and editor windows.

The UNIX window feature is implemented using a low-overhead character stream between LISP and UNIX. Under UNIX it is a device such as **/dev/ttyl0**, **/dev/ttyl1**, etc.; under LISP it is a stream of type **unix:unix-stream** which is built on **si:buffered-stream**. There are eight such devices.

The default configuration has two "login enabled" ports, and six disabled ones. This means that you can have two SYSTEM U-created UNIX windows at once. You can use the other lines for applications that want a unix-stream for communication between LISP and a UNIX process; for instance, PICON/RTIME uses this interface. You can change the relative number of enabled and disabled ports at any time by editing the UNIX file **/etc/ttys**. The file will contain entries that look like this:

```
...
  ...
17ttyl0
17ttyl1
07ttyl2
07ttyl3
...
  ...
```

Entries that begin with a **1** are login enabled; those that begin with a **0** are disabled. To login-enable an additional port, change the first character of the entry from zero to one. Conversely, to disable a port, change the first character from one to zero.

For example:

```
(defconst *s* (open "unix-stream-3:"))
#<UNIX-PORT 3>
```

The corresponding device under UNIX would be **/dev/ttyl3**, and the usual stream operations are supported. One caveat: since ZetaLISP does not use the traditional ASCII character codes, the integers returned by the **:tyi** message to a **unix-stream** need to be given the UNIX interpretation of the ASCII character set. Hence it is usually useful to build a stream around a **unix-stream** to do the LISP/UNIX character set translations.

There are two kinds of translation you typically want: one like the **FILE** protocol uses, to handle all characters normally stored in strings and files (8-bit representation), and another, to handle those characters and operations associated with terminals.

The **simple-unix-window-mixin**, as used by the **simple-unix-window** that implements the (SYSTEM) **U** command, provides both simple character set translation and display operation translation. Conventional terminal escape codes are defined to generate the **:clear-screen**, **:insert-line**, **:clear-eol**, and other operations including the handling of inverse video.

Here is the flavor used to implement the share TTY feature.

unix:unix-stream (si:buffered-stream) Flavor

Use this flavor stream for simple LISP/UNIX interprocessor communication.

Note that you should not use **make-instance** with this flavor; rather, **open** a device of type "**unix-stream-***n*" as in the example above. Alternatively, you can use the following function.

unix:find-unix-stream *with-login-p* Function

Returns the device pathname of the first free **unix-stream**. You can then operate on this. *with-login-p* determines whether or not to return a login-enabled stream.

See Section 5.1, "Devices from LISP", for more information on the device/pathname relationship, and Section 5.2, "Devices from UNIX", for more information and functions to use from the UNIX side.

4. Shared Memory

The shared memory area set aside by the Extended STREAMS software provides the most direct connection between different processors. You can use it to communicate between LISP and UNIX on a Lambda Plus, and between the two LISP processors on a Lambda 2x2. This method involves the least processor overhead, but requires the most care on the part of the programmer. The shared memory area is by default, 20K bytes; you can change this from the **config** program.

Facilities that relate directly to shared memory are covered below; the general Multibus and NuBus functions are described in Appendix A.

4.1 Shared Memory from LISP

These are the facilities for accessing shared memory from LISP.

si:*global-shared-memory-size* Variable

Number of bytes in the shared memory area. **Never set this variable**; if you want to change the size of the shared memory area, do so from **config.**

si:*global-shared-memory-8* Variable

An **art-8b** array indirected to the shared memory area.

si:*global-shared-memory-16* Variable

An **art-16b** array indirected to the shared memory area.

si:*global-shared-memory-32* Variable

An **art-32b** array indirected to the shared memory area. Currently, the high seven bits of a number get overwritten with a data type when you read from a location in this array. To get true 32-bit access, use adjacent locations in the **si:*global-shared-memory-16*** array or use the functions described below.

si:share-mem-read *address* Function

Reads a 32-bit value from a given *address.* Since this function always reads 32 bits of data aligned on the word boundary, the lower two bits of the address are ignored.

si:share-mem-write *address data* Function

Writes a 32-bit value to *address.* Again, it will write only along word boundaries, so the lower two bits of the address are ignored.

4.2 Shared Memory from UNIX

In order to use the UNIX shared memory functions you need to let **C** know about them. They are defined in **/usr/lib/libshare.a**. To do this, add the following line to the top of your **C** files.

```
#include <share.h>
```

You also need to specify the **-lshare** option to **cc**, the compile/link command.

```
cc vision.c -lshare
```

These are the UNIX functions and variables that will allow you to take advantage of Extended STREAMS from the UNIX processor. They fall into several catagories.

- Shared memory setup functions.
- Byte swap primitives.
- Multibus and NuBus read and write functions.
- Functions that access the system configuration structure.

4.2.1 Shared Memory Setup

char ***sharebase** Variable

Pointer to shared memory area.

sharesize Variable

The size in bytes of the shared physical memory area.

share_setup () Function

Sets the variables **sharebase** and **sharesize**. Returns -1 on failure.

4.2.2 Byte Swap Primitives

The 68010 and the Lambda LISP Processor use different conventions for how numbers are stored. The 68010 stores the low order byte on the left; this is the so-called "big-endian" convention, that IBM uses. The Lambda LISP Processor (and the SDU) store the low order byte on the right; this is the "little-endian" convention also used by DEC. To reconcile these notational differences, several functions swap these bytes and allow you to convert from one format to the other.

swapn (*destptr, srcptr, nwords*) Function

long *destptr, *srcptr;
int nwords;
Byte reverses and copies ***nwords*** words from ***srcptr*** to ***destptr***. The source and destination pointers can have the same value.

SWAB32 (*x*) Macro

Expands into an expression that byte reverses x. Since this is a macro it is fast because it expands into inline code. However, it evaluates its argument four times and so should not be used with large expressions or expressions with side effects; in such cases, use the function defined below.

long **swab32** (*x*) Function
long x;
Returns a byte-reversed copy of *x*.

4.2.3 Multibus and NuBus Functions

The file **/dev/nubus** is an extension of the UNIX **mem** device. The following functions access the NuBus at the address specified, and bypass the UNIX memory mapping used in **/dev/mem** and **/dev/kmem**. All reads and writes are done through the **ioctls** defined in **/usr/include/mem.h**. **Do not read from or write to /dev/nubus directly.**

This file is accessible only to the superuser. **CAUTION:** There is no protection from bus timeouts, and an access that causes a timeout will crash UNIX. This problem will be fixed in the release of System V UNIX.

mread8 (*addr*) Function
long addr;
Reads an 8-bit value from the Multibus at the 20-bit address *addr*.

mwrite8 (*addr, data*) Function
long addr, data;
Writes an 8-bit value, *data*, to the Multibus at the 20-bit address *addr*.

mread16 (*addr*) Function
long addr;
Reads a 16-bit value from the Multibus at the 20-bit address *addr*.

mwrite16 (*addr, data*) Function
long addr, data;
Writes a 16-bit value, *data*, to the Multibus at the 20-bit address *addr*.

In the following functions *slot* must be a number between 0 and 31, inclusive.

nread8 (*slot, addr*) Function
long slot, addr;
Reads an 8-bit value at the address specified by the 8-bit *slot* and the 24-bit *addr*.

nwrite8 (*slot, addr, data*) Function
long slot, addr, data;
Writes an 8-bit value, *data*, to the NuBus at the address specified by the 8-bit *slot* and the 24-bit *addr*.

nread16 (*slot, addr*) Function

long slot, addr;

Reads a 16-bit value at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nwrite16 (*slot, addr, data*) Function

long slot, addr, data;

Writes a 16-bit value, ***data***, to the NuBus at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nread32 (*slot, addr*) Function

long slot, addr;

Reads a 32-bit value at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nwrite32 (*slot, addr, data*) Function

long slot, addr, data;

Writes a 32-bit value, ***data***, to the NuBus at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nread (*addr*) Function

long addr;

Reads a 32-bit value from the 32-bit address, ***addr***.

nwrite (*addr, data*) Function

long addr, data;

Writes a 32-bit value, ***data***, to the 32-bit address, ***addr***.

4.2.4 System Configuration Structure Access

The file **/dev/sysconf** is a special extension to **/dev/nubus**. The file is readable by anyone, but writable only by the superuser. It allows access to just the **sysconf** and **procconf** structures. A file offset of 0 corresponds to the first byte of the **sysconf** structure. When accessed directly the **/dev/sysconf** the structures will be in Lambda Processor byte order. Access through **/dev/sysconf** is restricted to memory that is guaranteed to exist, so it is safe to read beyond the defined area.

getprocconf (*pp, pn, max*) Function

struct procconf *pp;

int pn, max;

Reads up to ***max*** bytes of the ***pn***th procconf structure into memory at ***pp***. Byte-reverses all the words in the structure.

getsysconf (*sp, max*) Function

struct sysconf *sp;

int max;

Reads up to ***max*** bytes of the sysconf structure into memory at ***sp***. Byte-reverses all the words in the structure.

5. Devices and Allocation

The following functions deal with shared devices: devices available from both LISP and UNIX. In order for the system to know about the devices, the SDU's (System Diagnostic Unit's) **config** program must be run to correctly configure the system. (For details see the *Release 2.0 Installation Packet.*)

5.1 Devices from LISP

ZetaLISP devices are coming to be thought of as analogous to files, just as UNIX devices are. This means that you can operate on devices through their pathnames, and macros like **with-open-file** will work appropriately.

```
(with-open-file (str "sdu-serial-b:")
  (format str "This is a test. ~C#o215 ~C#o212))
```

You can allocate and deallocate devices, and perform other operations by sending messages to the objects gotten by parsing the device pathname; however, since devices in ZetaLISP are not (unlike UNIX) really identical to files from a software point of view, you need to go down an extra level to find the device object. To do this, send the parsed pathname the **:host** message. You can then send this object the messages for operations that you want carried out.

For example:

```
(send (send (fs:parse-pathname "medium-resolution-color:") :host)
:allocate-if-easy)
```

To find out the available devices you can either look at the **Devices** option of **PEEK** or evaluate the variable **si:all-shared-devices**. Looking at **PEEK** is the method of choice, because the **si:all-shared-devices** only contains shared devices, and is not guaranteed to remain a stable part of the system. What **si:all-shared-devices** returns is a list of all the shared device *objects*; not the device *pathnames*. This means that you can send messages directly to these objects without first sending them a **:host** message.

5.1.1 Device Messages

These are some of the user callable methods for the flavor **si:shared-device**. They are actually inherited from **si:basic-shared-device**. For more information see the file **SYS:SYS;SHARED-DEVICE.LISP**.

:owner Operation on **basic-shared-device**

Returns either **nil** which means that the device is free; a number from 0 to 31, which is the slot number of the processor that currently owns the device; or **:not-on-bus**, which means that you don't physically own such a device.

:quad-slot Operation on **basic-shared-device**

For a NuBus device, this returns the quad-slot the device occupies; else **nil**.

:device-still-owned-by-me-p Operation on **basic-shared-device**

Returns **t** if the same processor still owns the device; **nil** if its either **:not-on-bus**, free, or owned by someone else.

:error-if-i-dont-own-device Operation on **basic-shared-device**

If **:device-still-owned-by-me** would return **nil** this signals the appropriate error.

:allocate-if-easy Operation on **basic-shared-device**

Allocates the device and returns **t** if it is currently free; returns **nil** if the device is not on the bus, or is owned by someone else.

:allocate Operation on **basic-shared-device**

Allocates the device and returns **t** if it is currently free; signals the appropriate error if not.

:deallocate Operation on **basic-shared-device**

Deallocates the device if owned by the current processor, else just returns.

5.1.2 Configuration Variables

These are some variables used by the shared device function.

si:*sys-conf* Variable

A representation of the system configuration structure. Use **describe** on the variable to get a "human readable" response.

si:*my-proc-conf* Variable

A representation of the processor configuration structure for this particular processor.

5.2 Devices from UNIX

The following standard **C** system calls allow you to access devices from the UNIX side. This is a brief review of these functions, in LISP-style documentation. For complete documentation see the *UNIX Reference Manual*.

open (*device, mode*) Function

char *device;

int mode;

The returned value is **int**, a *file descriptor*. (A file descriptor (*fd*), is an identification number that you use when referring to that device in functions.) A negative return value indicates failure. A *mode* of 0 indicates reading; *mode* of 1 is writing; *mode* of 2 is for both reading and writing.

read (*fd, buffer, length*) Function
char *buffer;
int fd, length;
The returned value is an **int**, the number of characters actually read. A negative value indicates failure; zero indicates logical end of file.

write (*fd, buffer, length*) Function
char *buffer;
int fd, length;
The returned value is an **int**, the number of characters actually written.

To find out what devices are defined for UNIX look at the **/dev** directory under UNIX. Currently many more devices are defined for LISP than for UNIX.

Here is an example that measures the cycle frequency of a LISP/UNIX/LISP communication channel.

```
main()
 {int f; char c[1];
  f = open("/dev/tty14",2);
  if (f<0) {printf("error"); exit(0);}
  while (1)
  { read(f,c,1);
    write(f,c,1);}}

;; from LISP

(defconst *p* (make-instance 'unix:unix-stream ':port-number 4))

(defun test (&optional (n 10000.) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j)))
      ((= j n))
   (send *p* ':tyo 5)
   (send *p* ':tyi))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
        "cycles per second"))
```

Appendix A. Memory Functions From LISP

There are two ways to view shared physical memory, other than at the device/stream level which uses shared physical memory in its device buffers. One way is through system calls accessing a particular memory location. In LISP this would be a call to a function such as **%nubus-read**. To accomplish this kind of memory access under UNIX, you need to use the **share** library functions discussed earlier, in Section 4.2, "Shared Memory From UNIX".

The second way to access shared physical memory is through virtual-memory/physical-memory mapping, as normal language-specific references to data structures which are previously arranged to have some fixed relationship with the virtual memory subsystems of the processors under consideration. This second way is more powerful but inherently more difficult, because it can bring to the forefront the problems of finite memory and disk resource allocation which were previously handled by the system.

The functions described below are both powerful and dangerous; work carefully to ensure that these functions are used in ways that don't hurt your programming environment. **CAUTION:** The Lambda does not handle non-existing memory exceptions. Reference to non-existing memory with any of the **%bus** functions will result in the machine halting with a bus timeout. The SDU can recognize this condition and restart the processor. In other words, if you use this function to read information from a memory location that does not exist you will crash the Lambda and may need to warm or cold boot it.

NuBus Functions

%nubus-read *slot byte-address* Function

Returns the contents of a word read from the NuBus. Addresses on the NuBus are divided into an 8-bit slot number which identifies the physical board being referenced and a 24-bit address within the slot. (Slot numbers on the bus go from F0 through FF.) The address is measured in bytes and therefore should be a multiple of four. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%nubus-write *slot byte-address word* Function

Writes the contents of a word to the NuBus. **Caution:** This function can crash the Lambda if you access nonexistent memory.

Multibus Functions

%multibus-read-8 *address* Function

Reads an 8-bit byte from the Multibus byte address. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-write-8 *address value* Function

Writes an 8-bit byte to the Multibus byte address. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-read-16 *address* Function

Reads a 16-bit halfword from the Multibus byte address. NOTE: To use this function you need to have the latest version of the SDU hardware, which is revision **K**. To find out whether you have this revision, look at the assembly number on the SDU board (slot 15). The last character of the assembly number is the revision letter. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-write-16 *address value* Function

Writes a 16-bit halfword to the Multibus byte address. NOTE: This function requires revision **K** of the SDU hardware. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-read-32 *address* Function

Reads a 32-bit word from the Multibus byte address. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-write-32 *address value* Function

Writes a 32-bit word to the Multibus byte address. NOTE: This function requires revision **K** of the SDU hardware. **Caution:** This function can crash the Lambda if you access nonexistent memory.

The **%multibus** functions can be effectively used for writing simple device drivers for heavily buffered Multibus devices which can be efficiently handled by a busy-wait.

The following example illustrates one way to write a function that writes to an array processor on the MultiBus.

```
(defconst *opcode-reg* #x0A00C)
(defconst *status-reg* #x0A00D)
(defconst *data-start* #x0A00E)
(defconst *op-clear* 0)
(defconst *op-fft* 1)

(defun fft-data-array (x)
  (%multibus-write-8 *opcode-reg* *op-clear*) ; reset machine clear
  (%multibus-write-8 *status-reg* 0)
  (dotimes (j (array-length x))
    (%multibus-write-8 (+ *data-start* j) (aref x j)))
  (%multibus-write-8 *opcode-reg* *op-fft*)
  (process-wait "Array Processor"
    #'(lambda (reg) (not (zerop (%multibus-read-8 reg))))
    *status-reg*))
```

Certain areas of virtual memory are by default mapped to Multibus and NuBus memory. Some functions for dealing with this are defined in the file **SYS:MULTIBUS;MAP**. This file must be loaded if you want to use the following function and variables.

si:describe-multibus-address-space Function

Provides a listing of what address space is free, what is used, and what is mapped to the NuBus.

Below are three arrays mapped to the Multibus. All three do a 32-bit access; then, for the **art-8b** and the **art-16b**, all but the relevant 8, or 16 bits are stripped off. Therefore, you can use them for accessing device buffers, but not in a controller situation, because they may confuse device registers.

si:*multibus-bytes* Variable

An **art-8b** array mapped to the Multibus.

si:*multibus-halfwords* Variable

An **art-16b** array mapped to the Multibus

si:*multibus-words* Variable

An **art-32b** array mapped to the Multibus.

Safe Address Space

Address space available for use by applications programmmers occasionally changes. The best way to make sure that address space can be used is to call LMI and ask. Outside of Massachusets call 1-800-872-LISP. Within the state call 1-800-325-6115.

Appendix B. Online Example

This file can be found online in **sys:examples;streams**.

```
;;; -*- Mode:LISP; Package:(STE global); Fonts:(cptfont); Base:8 -*-

;; Copyright LISP Machine, Inc. 1984
;;   See filename "Copyright" for
;; licensing and release information.

;; A self-contained example of streams software usage for
;; testing the performance of and documenting the LAMBDA<->UNIX interface.
;; This code runs in system version 1.120, unix-interface version 12.
;; 10/13/84 00:10:24 -George Carrette.
;; modified for release II beta-test 2/26/85 13:23:09 -George Carrette.

;; To run the tests:
;; (1) Create the C programs by running (CREATE-C-PROGRAMS)
;;     These functions illustrate some of the higher level protocals.
;; (2) Create a split-screen with two lisp listeners.
;;     Use RUN-C-PROGRAM in the top window, switch to the bottom and
;;     use the corresponding lisp function.

(defun attached-unix-host ()
  "Returns host object for attached unix-host if it exits otherwise NIL"
;; Relevant variable:
;; si:*other-processors* list of structures of type SI:OTHER-PROCESSOR
  (dolist (op si:*other-processors*)
    (let ((host (SI:GET-HOST-FROM-ADDRESS
  (si:%processor-conf-chaos-address (si:op-proc-conf op))
  ':CHAOS)))
      (if (typep host 'fs:unix-host)
  (return host)))))

(defun temp-unix-path (name type)
  (fs:make-pathname ':host (attached-unix-host)
    ':directory "TMP"
    ':name (string-append (string-upcase si:user-id)
  "-"
  name)
    ':type type))

;; a simple "null-device" for testing.

(defconst *p* (open "unix-stream-4:"))

(defun null-device (message &rest ignored)
  (selectq message
```

```
    (:tyi 0)
    (:tyipeek 0)
    (:which-operations '(:tyo :tyi :tyipeek :untyi :string-out))))

(defconst *null* (closure () #'null-device))

;; This uses the FILE protocal and EVAL protocal.

(defun share-compile-string (name string)
  "writes out the string as name.c and C compiles it to name"
  (with-open-file (stream (temp-unix-path name "C") ':out)
    (princ string stream))
  (simple-unix-eval (attached-unix-host)
    (format nil "cc ~A -o ~A -lshare"
    (send (temp-unix-path name "C") ':string-for-host)
    (send (temp-unix-path name ':unspecific)
  ':string-for-host))))

(defun simple-unix-eval (host command)
  (with-open-stream (s (chaos:open-stream host
  (format nil "EVAL ~a" command))
    (format t "~&% ~A~%" command)
    (do ((c (send s ':tyi) (send s ':tyi)))
((null c))
      (send standard-output ':tyo
    (selectq c
      ((12 15) #\return)
      (11 #\tab)
      (t c))))))


(defvar *c-programs* ())

(defun enter-c-program (name string)
  (setq *c-programs* (delq (ass #'string-equal name *c-programs*)
   *c-programs*))
  (push (list name string) *c-programs*)
  name)

(defun create-c-programs ()
  (dolist (p *c-programs*)
    (create-c-program (car p))))

(defun create-c-program (x)
  (let ((p (ass #'string-equal x *c-programs*)))
    (format t "~&;Writing and compiling ~A.C" (car p))
    (apply #'share-compile-string p)))

(defun run-c-program (name)
  (simple-unix-eval (attached-unix-host)
```

```
    (send (temp-unix-path name ':unspecific)
  ':string-for-host)))


;;; The tests

;;; open loop frequency

(defun test-olf (&optional (n 1000.) (stream *p*) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j)))
      ((= j n)
       (send stream ':tyo #/S))
    (send stream ':tyo #/?))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
"cycles per second"))


(enter-c-program "OLFT" '
|//*  program for open-loop sink response *//
#include <stdio.h>
main()
 {int f,n; char c[1];
  f = open("//dev//ttyl4",2);
  if (f < 0) {printf(/"open lost\n/"); exit(0);}
   while(1)
   { n = read(f,c,1);
     if (n == 0) {printf("got end of file\n"); exit(1);}
     if (n < 0) {printf("read lost\n"); exit(0);}
     if (*c == 'S') {printf("Been told to stop\n"); exit(1);}}}
|)

;; the closed loop frequencey is the basic "remote-function-call"
;; overhead time. With this implementation it is highly dependant on,
;; and usually limited by the lisp scheduler timing because
;; of the process-wait which encumbers the ':tyi to the unix share tty.
;; As things stand, without adding an interrupt driven process wakeup
;; feature to the lispmachine system, the unix processor can
;; affect a process on the lispmachine in no less than 1/60'th of
;; a second. realtime programming applications needing faster response
;; times should consider more low-level clock-break and scheduler
;; modifications. A faster speciallized remote function call mechanism
;; itself calls for a special microcoded function. However, the
;; following is more than reasonable for any job that takes more
;; than half a second in the unix processor.

(defun test-clf (&optional (n 100.) (stream *p*) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j)))
      ((= j n)
```

```
      (send stream ':tyo #/S)
      (print (if (eq (send stream ':tyi) #/O)
  "Unix process stopped ok"
"Unix process failed to reply to stop")))
    (send stream ':tyo #/?)
    (send stream ':tyi))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
"cycles per second"))

(enter-c-program "CLFT" '
|//* program freq.c for closed-loop. *//
#include <stdio.h>
main()
 {int f,n; char c[1];
  f = open("//dev//tty14",2);
  if (f < 0) {printf("open lost\n"); exit(0);}
  while(1)
  { n = read(f,c,1);
    if (n == 0) {printf("got end of file\n"); exit(1);}
    if (n < 0) {printf("read lost\n");exit(0);}
    if (*c == 'S') {printf("Been told to stop\n");
                    c[0] = 'O';
                    write(f,c,1);
                    exit(1);}
    n = write(f,c,1);
    if (n < 0) {printf("write lost\n"); exit(0);}}}
|)


(defsetf si:share-mem-read si:share-mem-write)

(defun share-mem-read-single-float (addr)
  (float-68000-32b (si:share-mem-read addr)))


(defmacro share-mem-read-bit (j)
  `(ldb (byte 1 (remainder ,j 32))
(si:share-mem-read (quotient ,j 32))))

(defun float-68000-32b (x)
  "Take 32bits, a 68000 float, and return a lisp float object"
  ;; note: This takes byte reversal into account. It doesnt try to be
  ;; efficient in its use of lispmachine arithmetic.
  (// (* (expt -1 (ldb #o3701 x))
 (expt 2.0 (- (ldb #o3007 x) #o100))
 (+ (ldb #o2010 x)
    (ash (+ (ldb #o1010 x)
    (ash (ldb #o0010 x)
 8.))
 8.)))
```

```
      #o100000000))


(defun 68000-32b-float (x &aux sign exp frac)
  "Take a lispmachine floating point number and return 32 bits suitable for
the 68000"
  (cond ((zerop x)
 0)
('else
 (cond ((small-floatp x)
(cond ((< x 0.0)
       (setq sign 1)
       (setq x (- x)))
      (t
       (setq sign 0)))
(setq exp (+ (- (si:%short-float-exponent x) #o101) #o100))
(setq frac (ash (- (si:%short-float-mantissa x) (expt 2 16))
(- 23 16))))
       ((floatp x)
(cond ((< x 0.0)
       (setq sign 1)
       (setq x (- x)))
      (t
       (setq sign 0)))
(setq exp (+ (- (si:%single-float-exponent x) #o2001) 127))
(setq frac (ash (- (si:%single-float-mantissa x) (expt 2 30))
(- 23 30))))
       (t
(ferror nil "Not a floating point number: ~S" x)))
 (ferror nil "foo, work on this tommorow"))))


(defun test-inc-loop  (&optional (n 100.) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j))(value))
      ((= j n)
       (send *p* ':tyo #/S)
       (print (if (eq (send *p* ':tyi) #/O)
  "Unix process stopped ok"
"Unix process failed to reply to stop")))
    (setq value (test-inc-1 j))
    (or (= value (1+ j))
(format t "~&;Error, expecting ~D, got ~D"
(1+ j) value)))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
"cycles per second"))

(defun test-inc-1 (integer)
  "When the program test-inc is compiled on a unix system with
cc test-inc.c -lshare
```

```
and then executed, you can call (test-inc n) and it will
write the integer n into the shared-array area, signal the
unix process to do the computation, wait for the computation
to complete, then return the result, which is N+1 in this case."
  (si:share-mem-write 0 integer)
  (send *p* ':tyo #/?)
  (send *p* ':tyi)
  (si:share-mem-read 0))


(enter-c-program "INCT" '

|//*  program test-inc *//

#include <stdio.h>
#include <share.h>

main()
 {int f,n,*p,val;
  char c[1];
  f = open("//dev//tty14",2);
  if (f < 0) {printf("open lost\n"); exit(0);}
  if (share_setup() < 0) {printf("share setup lost\n"); exit(0);}
  p = (int *) sharebase;
  while(1)
  { n = read(f,c,1);
    if (n == 0) {printf("got end of file\n"); exit(1);}
    if (n < 0) {printf("read lost\n");exit(0);}
    if (*c == 'S') {printf("Been told to stop\n");
                    c[0] = '0';
                    write(f,c,1);
                    exit(1);}
    val = p[0];
    val = SWAB32(val)+1;
    p[0] = SWAB32(val);
    n = write(f,c,1);
    if (n < 0) {printf("write lost\n"); exit(0);}}}
|)


;; these test values are probably wrong (i.e. TOO LOW) for Release II.
;; 2/26/85 13:33:07 -gjc

;; (test-inc-loop)  20 Hz.


;; results:
;; (test-clf 100. *p*)            52.2 Hz. using freq.c
;; (test-clf 10000. *null*)       6.5 KHz.
;; (test-olf 1000. *p*)  303.3 Hz. Using cat /dev/tty14 > /dev/null
```

```
;; (test-olf 1000. *p*)  220.0 Hz  Using freqc.c
;; (test-olf 10000. *null*)  9.8 Khz.
```

Lisp Index

Common LISP Release Notes

Richard M. Stallman
Richard Mlynarik

Published by LMI 6033 W. Century Blvd. Los Angeles CA 90045
USA

Table of Contents

Chapter 1

Data Types

Common LISP defines four standard names for floating point formats. In order of increasing precision, these are SHORT-FLOAT, SINGLE-FLOAT, DOUBLE-FLOAT, and LONG-FLOAT. However, they are not required to be all distinct. The LISP machine actually provides two distinct floating point formats, just as it used to. SHORT-FLOAT is a name for the smaller one, the small-flonum, and the other three names, SINGLE-FLOAT and so on, are names for the larger one.

In a complex number, Common LISP specifies that the real and imaginary part must either both be rational or both be floating point numbers of the same type. This is now so.

In addition, a complex number whose components are rational and whose imaginary part is zero is automatically converted to a real number whenever it is formed. So complex rational numbers have only one representation; 12+0i is the same as 12. But the same is not true for floating imaginary parts: 12.0+0.0i is different from 12.0.

1.1 Names for Data Types

Common LISP includes a very general way of naming data types: type specifiers. A type specifier describes a class of possible LISP objects; the function TYPEP asks whether a given object matches a given type specifier.

Some type specifiers are symbols: for example, NUMBER, CONS, SYMBOL, INTEGER, CHARACTER, COMPILED-FUNCTION, ARRAY, VECTOR. Their meanings are mostly obvious, but a table follows below.

Lists can also be type specifiers. They are either combinations or restrictions of other type specifiers. The car of the list is the key to understanding what it means. An example of a combination is (OR ARRAY SYMBOL), which will match any array and will match any symbol. An example of a restriction type is (INTEGER 0 6), which specifies an integer between 0 and 6 (inclusive).

Any given object matches many different type specifiers; for example, the number 1 matches the type specifiers NUMBER, INTEGER, REAL, RATIONAL, (INTEGER -5 2), (SIGNED-BYTE 3), (MOD 2), (MOD 3), (MOD 4), (MOD 5), and infinitely many others. The function TYPE-OF returns a type specifier that a given object matches, chosen to be one that clearly and usefully classifies the object, but you should not make assumptions about which one it would be for any particular object.

TYPEP object type-specifier

T if object matches type-specifier.

TYPE-OF object

Returns some type specifier that object matches. This replaces the old usage of TYPEP with one arg, though that usage is still supported.

SUBTYPEP type1 type2

Returns T if type1 is a subtype of type2; that is, if any object of type1 is certainly also of type2. For example, (SUBTYPEP 'CONS 'LIST) is T, but (SUBTYPEP 'LIST 'NUMBER) is NIL, because not all lists are numbers (in fact, no lists are number). (SUBTYPEP 'NUMBER 'RATIONAL) is also NIL.

In some cases the system cannot tell whether type1 is a subtype of type2. In the most general case, it is impossible to tell due to the existence of SATISFIES type specifiers. SUBTYPEP's value is NIL if the system cannot tell. Thus, NIL could mean that type1 is certainly not a subtype of type2, or it could mean that there is no way to tell whether it is a subtype. SUBTYPEP returns a second value to distinguish these two situations: the second value is T if SUBTYPEP's first value is definitive, NIL if the system does not know the answer.

TYPECASE object clauses...

Tests object against various type-specs and dispatches according to the result. Each clause looks like

(type-spec forms...)

The clauses are tested one by one by matching object against the clause's type-spec. If it matches, then the forms of the clause are executed and the last form's values are returned from the TYPECASE. If no clause matches, the value is NIL.

A clause can also have T or OTHERWISE instead of a type-spec. Then the clause always matches if the previous clauses have not.

COERCE object type-spec

Converts object to an "equivalent" object that matches type-spec. Common LISP specifies exactly which types can be converted to which other types. In general, anything which would lose information, such as turning a float into an integer, is not allowed as a coercion. Here is a complete list of types you can coerce to.

COMPLEX
(COMPLEX type)

Real numbers can be coerced to complex. If the real is a rational, however, then the result will actually be the same as the original number, and not complex at all! But if you coerce it to type (COMPLEX SINGLE-FLOAT), say, then the result really will be complex.

SHORT-FLOAT
SINGLE-FLOAT

Rational numbers can be coerced to floating point numbers, and any kind of floating point number can be coerced to any other floating point format.

FLOAT

Rational numbers are converted to SINGLE-FLOATs; floats of all kinds are left alone.

CHARACTER

Strings of length "one" can be coerced to characters. Symbols whose pnames have length "one" can also be. Integers can be coerced to characters.

LIST
VECTOR or ARRAY or any restricted array type

Any list or vector can be coerced to type LIST or to any type of array.

If you specify a type of array with restricted element type, you may actually get an array that can hold other kinds of things as well. For example, the Lambda does not provide anything of type (ARRAY SYMBOL), so if you specify that, you will get an ART-Q array (since that at least can hold symbols). Also, if the elements you have in the original sequence do not fit in the new array, you just get an error.

T

Any object can be coerced to type T, without change to the object.

The following sections are tables of type specifiers. Valid kinds of lists are described according to the symbol that would appear as the car of the list.

1.2 Basic Data Types

CONS	Non-NIL lists
SYMBOL	Symbols
ARRAY	All arrays, including strings
NUMBER	Numbers of all kinds
INSTANCE	All instances of any flavor
STRUCTURE	Named structures
LOCATIVE	Locatives
CLOSURE	Closures
ENTITY	Entities
STACK-GROUP	Stack groups
COMPILED-FUNCTION	Macrocode functions such as the compiler makes
MICROCODE-FUNCTION	Built-in functions implemented by the microcode
SELECT	Select-method functions (defined by DEFSELECT)
CHARACTER	Character objects

1.3 Other Useful Simple Types

T	All LISP objects belongs to this type
NIL	Nothing belongs to this type
STRING-CHAR	Characters that can go in strings
STANDARD-CHAR	Characters defined by Common LISP. These are the 95 Ascii printing characters (including Space) and Return.
LIST	Lists, including NIL
SEQUENCE	Lists and vectors. There are many new functions that accept either a list or a vector as a way of describing a sequence of elements.
NULL	NIL is the only object that belongs to type NULL
KEYWORD	Keywords (symbols whose home package is KEYWORD)
ATOM	Anything but CONSes
COMMON	Objects of all types defined by Common LISP. This is all LISP objects except closures, entities, stack groups, locatives, instances, select-methods, and compiled and microcode functions. (but a few kinds of instances, such as pathnames, are COMMON, because Common LISP does define how to manipulate pathnames, and it is considered irrelevant that the LISP machine happens to implement pathnames using instances).
STREAM	Anything that looks like it might be a valid I/O stream. It is impossible to tell for certain whether an object is a stream, since any function with proper behavior may be used as a stream. Therefore, use of this type specifier is discouraged.

1.4 More Obscure Types

PACKAGE	Packages, such as FIND-PACKAGE might return
READTABLE	Structures such as can be the value of *READTABLE*
RANDOM-STATE	Random-states. See RANDOM, below.
PATHNAME	Pathnames (instances of the flavor PATHNAME)
HASH-TABLE	Hash-tables (instances of the flavor HASH-TABLE)
flavor-name	Instances of that flavor, or of any flavor that contains it
defstruct-name	Structures of that type, or of any type that includes it

1.5 Simple Number Types

NUMBER	Any kind of number
INTEGER	Fixnums and bignums
RATIO	Explicit rational numbers, such as 1/2
RATIONAL	Integers and ratios
FIXNUM	Small integers, whose %DATA-TYPE is DTP-FIX and which occupy no storage
BIT	Very small integers--only 0 and 1 belong to this type
BIGNUM	Larger integers, which occupy storage
FLOAT	Any floating point number regardless of format
SHORT-FLOAT	Small-flonums
SINGLE-FLOAT	Regular-size flonums

DOUBLE-FLOAT, LONG-FLOAT
Synonymous with SINGLE-FLOAT, on the Lambda

REAL
Any number that is not explicitly stored as complex

COMPLEX
A number explicitly stored as complex. It is possible for such a number to have zero as an imaginary part but only if it is a floating point zero.

1.6 Restriction Types for Numbers

(COMPLEX type-spec)

A complex number whose components match type-spec. Thus, (COMPLEX RATIONAL) is the type of complex numbers with rational components. (COMPLEX T) is equivalent to COMPLEX.

(INTEGER low high)

An integer between low and high. low can be:

- An integer, which is then an inclusive lower limit.
- A list of one integer element. That integer is then an exclusive lower limit.
- *, which means that there is no lower limit.

high has the same sorts of possibilities. If high is omitted, it defaults to *. If both low and high are omitted, you have (INTEGER), which is the same as plain INTEGER. (INTEGER 0 *) therefore specifies a nonnegative integer. So does (INTEGER 0). (INTEGER -4 3) specifies an integer betwee -4 and 3, inclusive. FIXNUM is equivalent to (INTEGER l h), for appropriate values of l and h. BIT is equivalent to (INTEGER 0 1).

(RATIONAL low high)
(FLOAT low high)
(SHORT-FLOAT low high)
(SINGLE-FLOAT low high)
(DOUBLE-FLOAT low high)
(LONG-FLOAT low high)

These specify restrictive bounds for the types RATIONAL, FLOAT and so on. The bounds work on these types just the way they do on INTEGER.

(MOD high)

A nonnegative integer less than high. high should be an integer. (MOD), (MOD *) and plain MOD are allowed, but are the same as (INTEGER 0).

(SIGNED-BYTE size)

An integer that fits into a byte of size bits, of which one bit is the sign bit.

(SIGNED-BYTE 4) is equivalent to (INTEGER -8 7).

(SIGNED-BYTE *) and plain SIGNED-BYTE are the same as INTEGER.

(UNSIGNED-BYTE size)

a nonnegative integer that fits into a byte of size bits, with no sign bit. (UNSIGNED-BYTE 3) is equivalent to (INTEGER 0 7).

(UNSIGNED-BYTE *) and plain UNSIGNED-BYTE are the same as (INTEGER 0).

1.7 Simple Types for Arrays

ARRAY	All arrays
SIMPLE-ARRAY	Arrays that are not displaced and have no fill pointers
VECTOR	Arrays of rank one
BIT-VECTOR	ART-1B arrays of rank one
STRING	ART-STRING and ART-FAT-STRING arrays of rank one
SIMPLE-BIT-VECTOR	(AND BIT-VECTOR SIMPLE-ARRAY)
SIMPLE-STRING	(AND STRING SIMPLE-ARRAY)

SIMPLE-VECTOR — Simple-arrays of rank one, whose elements are unrestricted (ART-Q, ART-Q-LIST, etc). This is not the same as (AND VECTOR SIMPLE-ARRAY)!

1.8 Restriction Types for Arrays

(ARRAY element-type dimensions)

an array belongs to this type if:

1. It is capable of having as an element anything that matches element-type and nothing else. If element-type is (SIGNED-BYTE 4), the array must be an ART-4B array; an ART-8B or ART-Q array whose elements happen to be less than 16. does not belong to (ARRAY (SIGNED-BYTE 4)).

If element-type is T, it specifies arrays that are capable of having any LISP object as a component (such as ART-Q arrays).

If element-type is *, the array type is not restricted.

2. Its dimensions match dimensions. dimensions can be an integer or a list.

If it is an integer, it specifies the rank of the array. Then any array of that rank matches.

If dimensions is a list, its length specifies the rank, and each element of dimensions restricts one dimension. If the element is an integer, that dimension's length must equal it. If the element is *, that dimension's length is not restricted.

If dimensions is * or omitted, the array's dimensions are not restricted.

(SIMPLE-ARRAY element-type dimensions)

The restrictions work as in (ARRAY element-type dimensions), but in addition the array must be a simple array.

(VECTOR element-type size)

element-type works as above. The array must be a vector. size must be an integer or *; if it is an integer, the vector's length must equal size.

(BIT-VECTOR size)
(SIMPLE-VECTOR size)
(SIMPLE-BIT-VECTOR size)
(STRING size)
(SIMPLE-STRING size)

The type of elements is implicitly specified; therefore, there is no point in having an element-type. size works as in VECTOR.

1.9 Combination Type Specifiers

(SATISFIES predicate)

An object belongs to this type if the function predicate returns non-NIL with the object as its argument. Thus, (SATISFIES NUMBERP) is equivalent as a type specifier to NUMBER (though the system could not tell that this is so). predicate must be a symbol, not a LAMBDA-expression.

(AND type-specs...)

Includes an object if it matches all the type-specs. Thus, (AND INTEGER (SATISFIES ODDP)) is the type of odd integers.

(OR type-specs...)

Includes all objects that match any of type-specs. Thus, (OR NUMBER ARRAY) includes all numbers and all arrays.

(NOT type-spec)

Includes all objects that do not match type-spec.

1.10 Defining New Type Specifiers

DEFTYPE type-name lambda-list body...

Defines type-name as a type specifier by providing code to expand it into another type specifier -- a sort of type specifier macro.

When a list starting with type-name is encountered as a type specifier, the lambda-list is matched against the cdr of the type specifier just as the lambda-list of a DEFMACRO is matched against the cdr of a form. Then the body is executed and should return a new type specifier to be used instead of the original one.

If there are optional arguments in lambda-list for which no default value is specified, they get * as a default value.

If type-name by itself is encountered as a type specifier, it is treated as if it were (type-name); that is to say, lambda-list is matched against no arguments.

1.10.1 Type Predicates

INTEGERP object

T if object is an integer. The Common LISP name for FIXP.

CLI:LISTP object

T if object is a list, including NIL. Regular LISTP is unchanged, and still returns NIL if the arg is NIL.

CHARACTERP object

T if object is a character object.

VECTORP object

T if object is an array of rank 1.

BIT-VECTOR-P object

T if object is an ART-1B array of rank 1.

SIMPLE-VECTOR-P object

T if object is an array of rank 1, with no fill pointer and not displaced, which can have any LISP object as an element.

SIMPLE-BIT-VECTOR-P object

T if object is an ART-1B array of rank 1, with no fill pointer and not displaced.

SIMPLE-STRING-P object

T if object is a string with no fill pointer and not displaced.

PACKAGEP object

T if object is a package.

COMPILED-FUNCTION-P object

T if object is a compiled function.

COMMONP object

T if object is of a type that Common LISP defines operations on. See the definition of the type specifier COMMON, above.

SPECIAL-FORM-P symbol

Non-NIL if symbol is defined as a function that takes some unevaluated args.

CHARACTERP, PATHNAMEP, HASH-TABLE-P, RANDOM-STATE-P, READTABLEP, and STREAMP are also defined.

Chapter 2

Evaluation

CLI:EVAL form &optional nohook

Evaluates form, Common Lisp style, in an empty lexical environment but using the current bindings of special variables. If nohook is non-NIL, the current *EVALHOOK* if any is not invoked.

EVAL form &optional nohook

Evaluates form in the traditional manner (all variables special). If nohook is non-NIL, the current *EVALHOOK* if any is not invoked.

SI:INTERPRETER-ENVIRONMENT Variable

SI:INTERPRETER-FUNCTION-ENVIRONMENT Variable]

These two variables hold the current lexical environment, and also serve as a flag to indicate if traditional nonlexical evaluation is being done. They are both NIL for Common Lisp evaluation in the global environment. The first is NIL and the second is T for traditional nonlexical evaluation.

To evaluate a form in a specified environment, use EVALHOOK and specify NIL for the evalhook and the applyhook.

EVALHOOK Variable

The Common Lisp name for the variable EVALHOOK. If this variable is non-NIL, any kind of EVAL will call it as a function instead of doing the usual work of EVAL.

The arguments passed to the hook function, if it is non-NIL, are:

- The form that was to be evaluated, and

- An object representing the environment of this application, suitable for passing as the environment argument to *EVAL, EVALHOOK or APPLYHOOK

which is everything the hook function needs to continue with the evaluation by calling EVALHOOK.

APPLYHOOK

Variable

The Common Lisp name for the variable APPLYHOOK. If this variable is non-NIL, any kind of EVAL will call it as a function instead of applying a function found in the car of a form to its arguments.

The arguments passed to the APPLYHOOK function, if it is non-NIL, are:

- The function that was to be applied

- The list of arguments that it was to be applied to

- An object representing the environment of this application, suitable for passing as the environment argument to *EVAL, EVALHOOK or APPLYHOOK.

which is everything the hook function needs to continue with the application by calling APPLYHOOK.

EVALHOOK and *APPLYHOOK* are bound to NIL before either kind of hook function is called.

EVALHOOK form evalhook applyhook &optional environment

Evaluates form in the specified environment, with evalhook and applyhook in effect for all recursive evaluations of subforms of form. However, the evalhook is not called for the evaluation of form itself.

environment is a list whose car is used for SI:INTERPRETER-ENVIRONMENT and whose cadr is used for SI:INTERPRETER-FUNCTION-ENVIRONMENT.

If environment is NIL, it specifies Common Lisp evaluation in an empty lexical environment. Traditional (all variables special) evaluation results from using (NIL . T) for environment.

Aside from this, the only reasonable way to get a value to pass for environment is to use the argument passed to an evalhook or applyhook function, use the &ENVIRONMENT argument of a macro, or do (LIST SI:INTERPRETER-ENVIRONMENT SI:INTERPRETER-FUNCTION-ENVIRONMENT) or the equivalent using WITH-STACK-LIST. Whichever way you obtain an environment object, you must take care not to use it after the context in which it was made is exited, because both SI:INTERPRETER-ENVIRONMENT and SI:INTERPRETER-FUNCTION-ENVIRONMENT normally contain stack lists which will become invalid then.

environment has no effect on the evaluation of a variable which is regarded as special. This is always done using SYMEVAL. However, environment contains the record of the local SPECIAL declarations currently in effect, so it does enter in the decision of whether a variable is special.

APPLYHOOK function list-of-args evalhook applyhook &optional environment

Applies function to list-of-args in the specified environment, with evalhook and applyhook in effect for all recursive evaluations of subforms of function's body. However, applyhook is not called for this application of function itself. See the description of EVALHOOK, above, for more information on environment.

LAMBDA-PARAMETERS-LIMIT Constant

Has as its value the number of parameters that a lambda-list may have. At least, if a lambda-list has fewer than this many parameters, that is guaranteed not to be too many. More parameters may or may not work.

CALL-ARGUMENTS-LIMIT Constant

Has as its value the number of arguments that can be dealt with in a function call. Fewer than this many arguments will never be more than the system can deal with. More arguments may or may not work.

Note that if you pass a list of arguments with APPLY to a function that takes "&rest" arguments, there is no limit on the number of elements this list may have (except the size of memory).

Chapter 3

Declarations

When you are running Common Lisp code, the interpreter pays attention to SPECIAL declarations just as the compiler does. Variables that are not declared special may be used only in accordance with lexical scoping. The interpreter is now totally compatible with the compiler in this regard.

This change has not been made for all programs because some users still do not compile their programs, assuming that all variables are special. All users should start compiling their programs, because at some time in the future, this new interpreter will become the default.

PROCLAIM &rest declarations

> This is a Common Lisp replacement for a DECLARE at top level. In Common Lisp, DECLARE is used only for local declarations. PROCLAIM differs from traditional top-level DECLARE in evaluating its arguments; thus, you would say (PROCLAIM '(SPECIAL X)) instead of (DECLARE (SPECIAL X)).
>
> Note that top-level SPECIAL declarations (such as the one used in the previous example) are no longer recommended (instead DEFVAR, DEFCONSTANT or DEFPARAMETER are recommended). And, since the Lisp machine does not pay attention to type declarations, the function PROCLAIM is of little use.

THE &quote type-specifier &eval value

> Is effectively the same as value. It declares that value is an object of the specified type. Since the Lambda does all type checking at run time, this has no effect. It has not been set up to do an error check at run time because its whole purpose is to allow open compilation on machines lacking fast microcode type checking.

DEFPARAMETER name initial-value [documentation] Macro

Is the Common Lisp name for DEFCONST.

DEFCONSTANT name value [documentation] Macro

Is similar to DEFPARAMETER but more powerful; it defines a compile-time constant. The compiler is allowed to substitute the value of name into functions that refer to name, making the assumption that name's value will never change. You will usually get a warning if you try to SETQ or bind such a symbol.

3.1 Control Constructs

CLI:CATCH is a synonym for *CATCH. CATCH is still the Maclisp function, for compatibility with old programs.

CLI:THROW is a synonym for *THROW. THROW is still the Maclisp function.

APPLY is now generalized to be identical to LEXPR-FUNCALL, making the latter somewhat obsolete.

FLET local-functions body . . . Special Form

Executes body with local-functions available as function definitions.

Each element of local-functions should look like

(name lambda-list function-body . . .)

just like the cdr of a DEFUN form; name is defined as a local function. Within the body, using name as a function name (using name as the car of a form, or using (FUNCTION name)) will access the local definition.

Local functions are like local variables; they are visible only in the lexical scope of the FLET.

Each local function is closed in the environment outside the FLET. As a result, the local functions cannot see each other.

LABELS local-functions body . . . Special Form

Is like FLET except that the local functions can call each other. They are closed in the environment inside the MACROLET, so all the local function names are accessible inside the bodies of the local functions.

MACROLET local-macros body. . . Special Form

Is like FLET except that the local function definitions are actually macros.

local-macros looks like local-functions, but each element is understood as the cdr of a DEFMACRO rather than the cdr of a DEFUN. If a local macro name appears as the car of a form lexically within the body, it will expand according to the local definition in the MACROLET.

CASE

Is a synonym for SELECTQ.

TYPECASE object clauses Macro

Picks a clause to execute by testing the type of object (with TYPEP). Each clause begins with a type specifier (which is not evaluated). In execution, the clauses are tested one by one by testing the value of object against that type specifier. As soon as a type specifier matches, that clause's body is executed and its values are the values of the TYPECASE.

A clause is allowed to start with OTHERWISE instead of a type specifier. Then it matches any object. A clause starting with T also does that, since T is a type specifier which all objects fit.

LOOP forms . . . Macro

In Common Lisp is equivalent to (DO-FOREVER forms . . .). This presents a problem because it is incompatible with the LOOP that has been installed traditionally. The two ways of using LOOP are distinguished by looking at the second element of the list; in the traditional LOOP macro, it must be a symbol. So if it is a symbol the LOOP is assumed to be the traditional kind. Otherwise, it is treated as a Common Lisp LOOP.

MAPL

Is Common Lisp's name for MAP. MAP in Common Lisp means something else, but MAP in non-Common-Lisp programs has not been changed.

3.2 Multiple Values

MULTIPLE-VALUES-LIMIT Constant

Smallest number of values that might possibly fail to work. Returning a number of values less than this many cannot possibly run into trouble with an implementation limit on number of values returned.

MULTIPLE-VALUE-SETQ

Is Common Lisp's name for MULTIPLE-VALUE. MULTIPLE-VALUE still works.

MULTIPLE-VALUE-CALL function argforms . . . Special Form

Evaluates the argforms, saving all of their values, and then calls function with all those values as arguments. This differs from

```
(FUNCALL function argforms)
```

because that would get only one argument for function from each argform, whereas MULTIPLE-VALUE-CALL will get as many args from each argform as the argform cares to return. This works by consing.

MULTIPLE-VALUE-PROG1 form forms . . . Special Form

Evaluates form, saves its values, evaluates the forms, discards their values, then returns whatever values form produced. This does not cons.

3.3 Macros

A macro's expander function now receives two arguments. The first one is the macro call being expanded, as before. The second one, which is new, is the environment argument passed to MACROEXPAND-1. An environment records many things, but the one which is relevant to expanding macros is the set of local macro definitions (made with MACROLET) currently in effect.

For compatibility, a macro expander function is allowed to demand only one argument. Then the environment is not passed. Macro definitions compiled before system 98.6 will in fact accept only one argument.

MACROEXPAND-1 form &optional environment

> Expands form if it is a macro call (or a call to a DEFSUBST function).
>
> environment is used only to supply the set of local MACROLET macro definitions in effect for this form.

MACROEXPAND form &optional environment

> Expands form if it is a macro call, then expands the result, and so on, until a form which is not a macro call is obtained; that form is returned.

DEFMACRO now allows additional lambda list keywords &WHOLE and &ENVIRONMENT.

&WHOLE is used, followed by a variable name, at the front of the argument list. That variable is bound to the entire macro call being expanded. Additional arguments to be bound as usual to parts of the macro call may follow the &WHOLE argument.

&ENVIRONMENT is used, followed by a variable name, anywhere in the argument list. That variable is bound to the environment object passed as the second argument to the expander function.

DEFMACRO allows the macro name to be any function spec. Normally only symbols are used, since the interpreter and compiler provide no standard way to look anywhere else for a macro definition. However, it can sometimes be useful to DEFMACRO a :PROPERTY function spec, if some part of the system is going to look on a property for a macro definition. For example, this is how you define how to do LOCF on some kind of form (see below).

MACROEXPAND-HOOK Variable

Holds a function used by MACROEXPAND to apply a macro's expander function to its argument, the macro call. The default value of this variable is FUNCALL. However, when the interpreter invokes macroexpansion, it will instead be another function that clobbers the original call so it looks like the expansion; this is called displacing the macro call and is used to speed up execution.

MACRO-FUNCTION function-spec

If function-spec is defined as a macro, then this returns its expander-function: the function which should be called, with a macro call as its sole argument, to produce the macro expansion. Otherwise, MACRO-FUNCTION returns NIL.

In fact, a definition as a macro looks like (MACRO . expander-function).

You can define function-spec as a macro with expander function expander by doing (SETF (MACRO-FUNCTION function-spec) expander). This is equivalent to (FSET function-spec (CONS 'MACRO expander)).

3.4 SETF And Related Things

It used to be the case that SETF could accidentally evaluate something twice. For example, (SETF (LDB %%FOO (BAR X)) 5) would evaluate (BAR X) twice. Macros based on SETF, such as INCF and PUSH, were even more likely to do so; (PUSH X (CAR (FOO))) would evaluate (FOO) twice. Now this never happens.

SETF now accepts any number of places and values, just as SETQ does.

PSETF place value place value...

Stores each value into the corresponding place, with the changes taking effect in parallel. The subforms of the places, and the values, are evaluated in order; thus, in

```
(PSETF (AREF A (TYI)) (TYI) (AREF B
(TYI)) (AREF A (TYI)))
```

the first input character indexes A, the second is stored, the third indexes B, and the fourth indexes A. The parallel nature of PSETF implies that, should the first and fourth characters be equal, the old value of that element of A is what is stored into the array B, rather than the new value which comes from the second character read.

SHIFTF place...

Sets the first place from the second, the second from the third, and so on.

The last place is not set, so it doesn't really need to be a SETF'able place; it can be any form.

The value of the SHIFTF form is the old value of the first place.

ROTATEF place...

Sets the first place from the second, the second from the third, and so on, and sets the last place from the old value of the first place.

Thus, the values of the places are permuted among the places in a cyclic fashion.

To define how to SETF a function, you now use DEFSETF. There are two ways to do this, the trivial way and the complicated way.

DEFSETF function setting-function

Says that the way to store into (function args...) is to do (setting-function args... new-value). For example, the DEFSETF for CAR looks like (DEFSETF CAR SYS:SETCAR), so that (SETF (CAR X) Y) expands into (SETCAR X Y). (SETCAR is like RPLACA except that SETCAR returns its second argument).

DEFSETF function (function-args...) (value-arg) body...

Says how to store into (function args...) by providing something like a macro defininition to expand into code to do the storing.

body computes the code; the last form in body returns a suitable expression.

function-args should be a lambda list, which can have optional and rest args.

body can substitute the values of the variables in this lambda list, to refer to the arguments in the form being SETF'd. Likewise, it can substitute in value-arg to refer to the value to be stored.

In fact, the function-args and value-arg will not actually be the subforms of the form being SETF'd and the value to be stored; they will be gensyms. After body returns, the corresponding expressions may be substituted for the gensyms, or the gensyms may remain as local variables with a suitable LET provided to bind them. This is how SETF ensures a correct order of evaluation.

DEFINE-SETF-METHOD function (function-args...) (value-arg) body...

Defines how to do SETF on places starting with function, with more power and generality than DEFSETF provides, but it is more complicated to use.

The function-args will be the actual subforms of the place to be SETF'd, and the full power of DEFMACRO arglists can be used to match against it.

value-arg will be the actual form used as the second argument to SETF.

body is executed, and it must return five values which describe how to do SETF on this place. It must identify all the subforms of the place which need to be evaluated (generally the function-args arglist is arranged to make each arg get one subform) and a temporary variable should be made for each one by calling GENSYM. Another temporary variable should be made to correspond to the value to be stored, again by calling GENSYM. Then the five values to be returned are:

A list of the temporary variables for the subforms of the place.

A list of the subforms that they correspond to.

A list of the temporary variables for the values to be stored. Currently there can only be one value to be stored, so there is only one variable in this list, always.

A form to do the storing. In this form, only the temporary variables should appear, none of the parts of the original SETF.

A form to get the value of the place (in case this is PUSH or INCF rather than SETF, and will need to examine the old value). This too should contain only the temporary variables.

This information is everything that the macro (SETF or something more complicated) needs to know to decide what to do.

Example:

```
(DEFINE-SETF-METHOD CAR (FUNCTION-SPEC)
    (LET ((TEMPVARS (LIST (GENSYM)))
          ((TEMPARGS (LIST (LIST-FORM)))
          (STOREVAR (GENSYM)))
      (VALUES TEMPVARS TEMPARGS (LIST STOREVAR)
              `(SI:SETCAR ,(FIRST TEMPVARS) ,STOREVAR)
              `(CAR ,(FIRST TEMPVARS))))))
```

is a definition of how to SETF (CAR list) which is equivalent to the simple (DEFSETF CAR SI:SETCAR) which is actually used.

Here it is appropriate to say that the way to define how to do LOCF on a function has been changed. LOCF properties are not used. Instead, you define a SI:LOCF-METHOD property, which should be either

- A symbol. Then that symbol should be the function to use to compute the locative. For example, (DEFPROP AREF ALOC SI:LOCF-METHOD).

- A macro definition, (MACRO . expander-function). The macro definition is expanded, with the form to be LOCF'd as its argument, and should return a form to compute the locative. For example,

  ```
  (DEFMACRO (:PROPERTY AREF SI:LOCF-METHOD) (ARRAY
  &REST INDICES) `(ALOC ,ARRAY . ,INDICES))
  ```

  would be equivalent to the simpler DEFPROP shown above.

GET-SETF-METHOD form

Invokes the SETF method for form (which must be a list) and returns the five values produced by the body of the DEFINE-SETF-METHOD for the symbol which is the car of form. The meanings of these five values are given immediately above. If the way to SETF that symbol was defined with DEFSETF, you still get five values, which you can interpret in the same ways; thus, DEFSETF is effectively an abbreviation for a suitable DEFINE-SETF-METHOD.

There are two ways to use GET-SETF-METHOD:

- In a DEFINE-SETF-METHOD for something like LDB, which is SETF'd by setting one of its arguments. You would append your new tempvars and tempargs to the ones you got from GET-SETF-METHOD to get the combined lists which you return. The forms returned by the GET-SETF-METHOD you would put into the forms you return.

- In a macro which, like SETF or INCF or PUSH, wants to store into a place.

An example of a DEFINE-SETF-METHOD that uses GET-SETF-METHOD is that for LDB:

```
(DEFINE-SETF-METHOD LDB (BYTESPEC INT)
  (MULTIPLE-VALUE-BIND
      (TEMPS VALS STORES STORE-FORM ACCESS-FORM)
      (GET-SETF-METHOD INT)
    (LET ((BTEMP (GENSYM))
          (STORE (GENSYM))
          (ITEMP (FIRST STORES)))
      (VALUES (CONS BTEMP TEMPS)
              (CONS BYTESPEC VALS)
              (LIST STORE)
              `(PROGN
                 ,(SUBLIS (LIST (CONS ITEMP
                                      `(DPB ,STORE ,BTEMP
                                            ,ACCESS-FORM)))
                          STORE-FORM)
                 ,STORE)
              `(LDB ,BTEMP ,ACCESS-FORM)))))
```

An example of a macro which uses GET-SETF-METHOD is PUSHNEW. (The real PUSHNEW is a little hairier to handle the test, test-not and key arguments).

```
 (DEFMACRO PUSHNEW (VALUE PLACE)
"Add ITEM to the front of the list PLACE,
     if it's not already MEMQ there."
 (MULTIPLE-VALUE-BIND
              (TEMPVARS TEMPARGS STOREVARS STOREFORM REFFORM)
          (GET-SETF-METHOD PLACE)
        (SI:SUBLIS-EVAL-ONCE
              (CONS `(-VAL- . ,VALUE) (PAIRLIS TEMPVARS
                                                TEMPARGS))
             `(IF (MEMQ -VAL- ,REFFORM)
                          ,REFFORM
                    ,(SUBLIS
                                      (LIST (CONS (CAR STORE
                               `(CONS -VAL- ,REFFORM)))
                     STOREFORM))
                      T T)))
```

SI:SUBLIS-EVAL-ONCE alist form &optional reuse-tempvars sequential-flag

Replaces temporary variables in form with corresponding values, but generates local variables when necessary to make sure that the corresponding values are evaluated exactly once and in the same order that they appear in alist. (This complication is skipped when the values are constant). The result is a form equivalent to

```
`(LET ,(MAPCAR #'(LAMBDA (ELT) (LIST (CAR
ELT) (CDR ELT))) alist) ,form)
```

but containing, usually, fewer temporary variables and faster to execute.

If reuse-tempvars is non-NIL, the temporary variables which appear as the cars of the elements of alist are allowed to appear in the resulting form. Otherwise, none of them appears in the resulting form, and if any local variables turn out to be needed, they are made afresh with GENSYM. reuse-tempvars should be used only when it is guaranteed that none of the temporary variables in alist is referred to by any of the values to be substituted; as, when the temporary variables have been freshly made with GENSYM.

If sequential-flag is non-NIL, then the value substituted for a temporary variable is allowed to refer to the temporary variables preceding it in alist. SETF and similar macros should all use this option.

DEFINE-MODIFY-MACRO macro-name (lambda-list...) combiner-function [doc-string] Macro

Is a quick way to define SETF'ing macros which resemble INCF. For example, here is how INCF is defined:

```
(DEFINE-MODIFY-MACRO INCF (&OPTIONAL
(DELTA 1)) + "Increment PLACE's value by
DELTA.")
```

The lambda-list describes any arguments the macro accepts, but not the first argument, which is always the place to be examined and modified. The old value of this place, and any additional arguments such as DELTA, are combined using the combiner-function (in this case, +) to get the new value which is stored back in the place.

3.5 Modules

In Common LISP, a module is a name given to a group of files of code. However, nothing in the LISP system records what the "contents" of any particular module may be. Instead, one of the files that defines the module will contain a PROVIDE form that says, when that file is loaded, "module FOO is now present." Other files may say, using REQUIRE, "I want to use module FOO." Normally the REQUIRE form also specifies the files to load if FOO has not been PROVIDEd already. If it does not, the module name FOO is used as a system name in MAKE-SYSTEM in order to load the module.

PROVIDE module-name

> Adds module-name to the list *MODULES* of modules already loaded.
>
> module-name should be a string; case is significant.

REQUIRE module-name &rest pathnames

> If module module-name is not already loaded (on *MODULES*), the files' pathnames are loaded in order to make the module available.
>
> module-name should be a string; case is significant. If pathnames is NIL, then (MAKE-SYSTEM module-name :NOCONFIRM) is done.
>
> NOTE: case is not significant in the argument to MAKE-SYSTEM!

MODULES Variable

> A list of module names PROVIDEd so far.

3.6 Numbers

= &rest numbers

T if all the arguments are numerically equal. They need not be of the same type; 1 and 1.0 are considered equal. Character objects are also allowed, and in effect coerced to integers for comparison.

CTRL-Z &rest numbers //= &rest numbers

these two synonymous functions return T if no two arguments are numerically equal. Ctrl-Z is an existing name, extended to more than two arguments; the name //= is new.

LCM integer &rest integers

Returns the least common multiple of the specified integers.

PI Constant

Is the value of Ctrl-G.

3.6.1 Division Functions

CLI:// number &rest numbers

With one argument, takes the reciprocal of number. With more than one argument, divides number by each of numbers, one by one. If an integer is divided by an integer, the result is a rational number, and is exactly correct. This is how CLI:// differs from ordinary //, which would behave like TRUNCATE in that case.

// may be converted in the future to divide integers exactly, and be the same as CLI:// is now.

Note that in Common LISP syntax you would write just / rather than //.

MOD number divisor

Returns the root of number modulo divisor. This is a number between 0 and divisor, or possibly 0, whose difference from number is a multiple of divisor. It is also the second value of (FLOOR number divisor). Examples:

```
(MOD 2 5)   =>  2
(MOD -2 5)  =>  3
(MOD -2 -5) => -2
(MOD 2 -5)  => -3
```

CLI:REM number divisor

Is a synonym for \. It is the second value of (TRUNCATE number divisor); a kind of remainder whose sign is the same as that of number. Only the absolute value of divisor matters.

The traditional REM function is, of course, a function for removing elements from a list with copying. That is why there is a separate CLI:REM function for Common LISP.

3.6.2 Floating Point Functions

FLOAT number &optional flonum

Converts number to a floating point number and returns it.

If flonum is specified, it specifies the float format to use--namely, the same format that flonum is--and if number is a float of a different format then it is converted.

If flonum is omitted, then number is converted to a single-float, but if number is already a floating point number it is returned unchanged.

DECODE-FLOAT flonum

Returns three values that express, in a different fashion, the value of flonum. The first value is a positive flonum of the same format having the same mantissa, but with an exponent chosen to make it between 1/2 and 1, less than 1. The second value is the exponent of flonum: the power of 2 by which the first value needs to be scaled in order to get flonum back. The third value expresses the sign of flonum.

It is a flonum of the same format as flonum, whose value is either 1 or -1.

Example: (DECODE-FLOAT 38.2) => 0.596875 6 1.0

INTEGER-DECODE-FLOAT flonum

Like DECODE-FLOAT except that the first value is scaled so as to make it an integer, and the second value is modified by addition of a constant so that it goes with the first argument.

Example:
(INTEGER-DECODE-FLOAT 38.2) => #O 11431463146 -25. 1.0

SCALE-FLOAT flonum integer

Multiplies flonum by 2 raised to the integer power. flonum can actually be an integer; if so, first it is converted to a flonum and then scaled.

(SCALE-FLOAT 0.596875 6) => 38.2
(SCALE-FLOAT #O11431463146 -25.) => 38.2

FLOAT-SIGN float1 &optional float2

Returns a flonum whose sign matches that of float1 and whose magnitude and format are those of float2. If float2 is omitted, 1.0 is used as the magnitude and float1's format is used.

(FLOAT-SIGN -1.0s0 35.3) => -35.3
(FLOAT-SIGN -1.0s0 35.3s0) => -35.3s0

FLOAT-RADIX flonum

Returns the radix used for the exponent in the format used for flonum. On the Lambda, floating point exponents are always powers of 2, so FLOAT-RADIX ignores its argument and always returns 2.

FLOAT-DIGITS flonum

Returns the number of bits of mantissa in the floating point format of which flonum is an example. It is 17. for small flonums and 32. for regular size ones.

FLOAT-PRECISION flonum

Returns the number of significant figures present in in the mantissa of flonum. This is always the same as (FLOAT-DIGITS flonum) for normalized numbers, and on the Lambda all flonums are normalized, so the two functions are the same.

3.6.3 Bit-Hacking Functions

LOGIOR, LOGAND, LOGXOR changed.

These functions now allow zero arguments, and return an identity for the operation. For LOGIOR or LOGXOR, the identity is zero. For LOGAND, it is -1.

LOGEQV &rest integers

Combines the integers together bitwise using the equivalence operation, which, for two arguments, is defined to result in 1 if the two argument bits are equal. This operation is asociative. With no args, the value is -1, which is an identity for the equivalence operation.

LOGNAND integer1 integer2

Returns the bitwise-NAND of the two arguments. A bit of the result is 1 if at least one of the corresponding argument bits is 0. Exactly two arguments are required because this operation is not associative.

LOGNOR integer1 integer2

Returns the bitwise-NOR of the two arguments. A bit of the result is 1 if both of the corresponding argument bits are 0. Exactly two arguments are required because this operation is not associative.

LOGORC1 integer1 integer2

Returns the bitwise-OR of integer2 with the complement of integer1.

LOGORC2 integer1 integer2

Returns the bitwise-OR of integer1 with the complement of integer2.

LOGANDC1 integer1 integer2

Returns the bitwise-AND of integer2 with the complement of integer1.

LOGANDC2 integer1 integer2

Returns the bitwise-AND of integer1 with the complement of integer2.

BOOLE-CLR Constant

The BOOLE opcode for the trivial operation that always returns zero.

BOOLE-SET Constant

The BOOLE opcode for the trivial operation that always returns one.

BOOLE-1 Constant

The BOOLE opcode for the trivial operation that always returns the first argument.

BOOLE-2 Constant

The BOOLE opcode for the trivial operation that always returns the second argument.

BOOLE-IOR Constant
BOOLE-AND Constant
BOOLE-XOR Constant
BOOLE-EQV Constant
BOOLE-NAND Constant
BOOLE-NOR Constant
BOOLE-ORC1 Constant
BOOLE-ORC2 Constant
BOOLE-ANDC1 onstant
BOOLE-ANDC2 Constant

The BOOLE opcodes that correspond to the functions LOGIOR, LOGAND, etc.

LOGTEST integer1 integer2

T if (LOGAND integer1 integer2) is nonzero. This is a Common LISP synonym for BIT-TEST.

LOGBITP index integer

T if the bit index up from the least significant in integer is a 1. This is equivalent to (LDB-TEST (BYTE index 1) integer).

LOGCOUNT integer

The number of 1 bits in integer, if it is positive.
The number of 0 bits in integer, if it is negative.
(There are infinitely many 1 bits in a negative integer.)

```
(LOGCOUNT #O15) =>  3
(LOGCOUNT #O-15)  =>  2
```

INTEGER-LENGTH integer

The minimum number of bits (aside from sign) needed to represent integer in two's complement.

3.6.4 Byte Functions

BYTE size position

Returns a byte-spec that specifies the byte of size bits, positioned to exclude the position least significant bits. This byte-spec can be passed as the first argument to LDB, DPB, %LOGLDB, LOGDPB, MASK-FIELD, %P-LDB, %P-LDB-OFFSET, and so on.

BYTE-POSITION byte-spec BYTE-SIZE byte-spec

Return, respectively, the size and the position of byte-spec. It is always true that (BYTE (BYTE-SIZE byte-spec) (BYTE-POSITION byte-spec)) equals byte-spec.

3.6.5 Random Numbers

A random-state is a structure whose contents specify the future actions of the random number generator. Each time you call the function RANDOM, it uses (and updates) one random-state. Random-states print as #S(RANDOM-STATE ...more data...) so that they can be read back in.

RANDOM &optional number state

Returns a randomly generated number. If number is specified, the random number is nonnegative and less than number, and of the same type as number (floating if number is floating, etc.).

According to Common LISP, number must always be specified. But you are still allowed to omit it, for the sake of compatibility. If number is omitted, the result is a randomly chosen fixnum.

state is a random-state object. RANDOM uses that object to choose the number to return, and updates the object so a different number would be chosen next.

RANDOM-STATE-P object

T if object is a random-state.

RANDOM-STATE Variable

This random-state is used by default when RANDOM is called (if you do not specify the state argument).

MAKE-RANDOM-STATE &optional random-state

Creates and returns a new random-state object.

If random-state is NIL, the new random-state is a copy of *RANDOM-STATE*.

If random-state is a random-state, the new one is a copy of that one.

If random-state is T, the new random-state is initialized truly randomly (based on the value of (TIME)).

3.6.6 Machine Precision Information

Common LISP defines some constants whose values give information in a standard way about the ranges of numbers representable in the individual LISP implementation.

MOST-NEGATIVE-FIXNUM Constant

Any integer smaller than this must be a bignum.

MOST-POSITIVE-FIXNUM Constant

Any integer larger than this must be a bignum.

MOST-POSITIVE-SHORT-FLOAT Constant

No short float can be greater than this number.

LEAST-POSITIVE-SHORT-FLOAT Constant

No positive short float can be closer to zero than this number.

LEAST-NEGATIVE-SHORT-FLOAT Constant

No negative short float can be closer to zero than this number.

MOST-NEGATIVE-SHORT-FLOAT Constant

No short float can be less than this (negative) number.

MOST-POSITIVE-SINGLE-FLOAT Constant
LEAST-POSITIVE-SINGLE-FLOAT Constant
LEAST-NEGATIVE-SINGLE-FLOAT Constant
MOST-NEGATIVE-SINGLE-FLOAT Constant

Similar to the above, but for single-floats (ordinary flonums) rather than for short-floats (small-flonums).

MOST-POSITIVE-DOUBLE-FLOAT Constant
LEAST-POSITIVE-DOUBLE-FLOAT Constant
LEAST-NEGATIVE-DOUBLE-FLOAT Constant
MOST-NEGATIVE-DOUBLE-FLOAT Constant
MOST-POSITIVE-LONG-FLOAT Constant
LEAST-POSITIVE-LONG-FLOAT Constant
LEAST-NEGATIVE-LONG-FLOAT Constant
MOST-NEGATIVE-LONG-FLOAT Constant

These are defined by Common LISP to be similar to the above, but for double-floats and long-floats. On the Lambda, there are no distinct double and long floating formats; they are synonyms for single-floats. So these constants exist but their values are the same as those of MOST-POSITIVE-SINGLE-FLOAT and so on.

SHORT-FLOAT-EPSILON Constant

Smallest positive short float that can be added to 1.0s0 and make a difference.

SINGLE-FLOAT-EPSILON Constant
DOUBLE-FLOAT-EPSILON Constant
LONG-FLOAT-EPSILON Constant

Smallest positive float that can be added to 1.0 and make a difference. The three names are synonyms on the LISP machine, for reasons explained above.

SHORT-FLOAT-NEGATIVE-EPSILON Constant

Smallest positive short float that can be subtracted from 1.0s0 and make a difference.

SINGLE-FLOAT-NEGATIVE-EPSILON Constant
DOUBLE-FLOAT-NEGATIVE-EPSILON Constant
LONG-FLOAT-NEGATIVE-EPSILON Constant

Smallest positive float that can be subtracted from 1.0 and make a difference.

3.7 Symbol and Plist Functions

SYMBOL-PLIST symbol

Is a new name for PLIST; it returns the contents of the property list of symbol. (SETF (SYMBOL-PLIST symbol) newvalue) can be used to set the property list.

SYMBOL-NAME symbol

Returns the pname string of symbol. This is a new name for GET-PNAME.

SYMBOL-VALUE symbol

Returns the value of symbol. This is a new name for SYMEVAL. (SETF (SYMBOL-VALUE symbol) newvalue) is used to alter a symbol's value. There is no equivalent of the function SET in Common LISP; this SETF construct is the only way to do it.

SYMBOL-FUNCTION symbol

Returns the function definition of symbol. This is a new name for FSYMEVAL. (SETF (SYMBOL-FUNCTION symbol) newvalue) is used to alter a symbol's function definition. There is no equivalent of the function FSET in Common LISP; this SETF construct is the only way to do it.

COPY-SYMBOL symbol &optional copy-props

Makes a new uninterned symbol whose name is the same as that of symbol. If copy-props is non-NIL, the value, function definition and property list of symbol are copied as well. This is a new name for COPYSYMBOL.

GENTEMP &optional (prefix "T") (a-package PACKAGE)

Creates and returns a new symbol whose name starts with prefix, interned in a-package, and is distinct from any symbol already present there. This is done by trying names one by one until a name not already in use is found.

GETF place property &optional default — Macro

Equivalent to (GET (LOCF place) property default), except that GETF is allowed in Common LISP, which does not have LOCF or locatives of any kind.

REMF place property — Macro

Equivalent to (REMPROP (LOCF place) property default), except that REMF is allowed in Common LISP.

GET-PROPERTIES place list-of-properties — Macro

The Common LISP replacement for GETL. It is like (GETL (LOCF place) list-of-properties) except that it returns slightly different values. Specifically, it searches the property list for a property name which is MEMQ in list-of-properties, then returns three values:

The property name found;

The value of that property;

The cell (in the property list) whose car is the property name found.

If nothing is found, all three values are NIL.

GENSYM

Now allows the prefix you specify to be any string. It used to have to be a single character.

3.8 Character Functions and Related Constants

Character objects were introduced in system version 97. In the traditional syntax, the character object A appears as #(CTRL-Z)/A. In Common LISP syntax, it looks like #\A.

Common LISP programs typically work with actual character objects but programs traditionally use integers to represent characters. The new Common LISP functions for operating with characters have been implemented to accept integers as well, so that they can be used equally well from traditional programs.

CHARACTERP object

T if object is a character object.

3.8.1 Components of Character Objects

Common LISP says that each character object has a character code, a font, and a bunch of bits. Each of these things is an integer from a fixed range.

However, you cannot necessarily take any valid code, any valid font, and any valid bits and make a character out of them. And if you can make a character out of them, it cannot necessarily be stored in a string.

CHAR-CODE char

Returns the code of char. This is what used to be done with (LDB %%CH-CHAR char).

CHAR-FONT char

Returns the font of char, a number less than CHAR-FONT-LIMIT.

CHAR-BITS char

Returns the bits of char, a number less than CHAR-BITS-LIMIT.

CHAR-CODE-LIMIT Constant

A constant whose value is a bound on the maximum code of any character. In the Lambda, currently, it is 400 (octal).

CHAR-BITS-LIMIT Constant

A constant whose value is a bound on the maximum bits value of any character. In the Lambda, currently, it is 40 (octal).

CHAR-FONT-LIMIT Constant

A constant whose value is a bound on the maximum font value of any character. In the Lambda, currently, it is 400 (octal).

The "bits" of a character are just the familiar Control, Meta, Super and Hyper bits, plus one more (the Mouse bit).

CHAR-CONTROL-BIT Constant
CHAR-META-BIT Constant
CHAR-SUPER-BIT Constant
CHAR-HYPER-BIT Constant

Constants with value 1, 2, 4 and 8. These give the meanings of the bits within the bits-field of a character object. Thus, (BIT-TEST CHAR-META-BIT (CHAR-BITS char)) would be non-NIL if char is a meta-character.

CHAR-BIT char name

T if char has the bit named by name. name is a symbol, one of :CONTROL, :META, :SUPER, or :HYPER. Thus, (CHAR-BIT #\META-X ':META) is T.

SET-CHAR-BIT char name newvalue

Returns a character like char except that the bit specified by name is present or absent according to newvalue (which is T or NIL). Thus, (SET-CHAR-BIT #\X ':META T) returns #\META-X.

3.8.2 Classifying Characters

STRING-CHAR-P char

T if char is a character that can be stored in a string. On the Lambda, this is if the bits and font of char are zero.

STANDARD-CHAR-P char

T if char is a standard character, according to Common LISP. This is a character that belongs to the standard Common LISP character set consisting of the 95 ASCII printing characters (including Space) and the Return character. This (STANDARD-CHAR-P #\END) is NIL.

GRAPHIC-CHAR-P char

T if char is a graphic character; one which has a printed shape. A, -, Space and are all graphic characters; Return, End and Abort are not. A character whose bits are nonzero is never graphic.

Ordinary output to windows prints graphic characters using the current font. Nongraphic characters are printed using lozenges unless they have a special formatting meaning (as Return does).

Common LISP says that programs may assume that graphic characters of font 0 may be assumed to be all of equal width. Since the Lambda allows you to use any font at any time, this clearly cannot always be true.

ALPHA-CHAR-P char

T if char is a letter, with zero bits.

UPPER-CASE-P char

T if char is an upper case letter, with zero bits.

LOWER-CASE-P char

T if char is a lower case letter, with zero bits.

BOTH-CASE-P char

T if char is a character which has distinct upper and lower case forms-- that is to say, a letter--with zero bits.

DIGIT-CHAR-P char &optional (radix 10.)

If char is a digit available in the specified radix, returns the "weight" of that digit. Otherwise, it returns NIL. If the bits of char are nonzero, the value is NIL. Thus, (DIGIT-CHAR-P #\8 8) is NIL but (DIGIT-CHAR-P #\8 9) is 8. Radices greater than ten use letters as additional digits, so (DIGIT-CHAR-P #\F 16.) is 15..

ALPHANUMERICP char

T if char is a letter or a digit 0 through 9, with zero bits.

CHAR= char1 &rest chars
CHAR//= char1 &rest chars
CHAR> char1 &rest chars
CHAR< char1 &rest chars
CHAR>= char1 &rest chars
CHAR<= char1 &rest chars

These are the Common LISP functions for comparing characters and including the case, font and bits in the comparison. On the Lambda they are synonyms for the numeric comparison functions =, >, etc. Note that in Common LISP syntax you would write CHAR/=, not CHAR//=.

CHAR-EQUAL char1 &rest chars
CHAR-NOT-EQUAL char1 &rest chars
CHAR-LESSP char1 &rest chars
CHAR-GREATERP char1 &rest chars
CHAR-NOT-LESSP char1 &rest chars
CHAR-NOT-GREATERP char1 &rest chars

These are the Common LISP functions for comparing characters, ignoring differences in case, font and bits.

3.8.3 Making Characters

CHARACTER object

Coerces object into a character and returns the character as a fixnum for traditional programs.

CLI:CHARACTER object

Coerces object into a character and returns the character as a character object for Common LISP programs.

CODE-CHAR code &optional (bits 0) (font 0)

Returns a character object made from code, bits and font, IF that is possible. Not all combinations of valid code, bits and font can go together. If the specified arguments do not go together, the value is NIL.

MAKE-CHAR char &optional (bits 0) (font 0)

Like CODE-CHAR except that the first argument is a character whose code is used, not an integer. In the Lambda, this and CODE-CHAR are identical and either one will accept a character or a number.

DIGIT-CHAR weight &optional (radix 10.) (font 0)

Returns a character that is digit with the specified weight, and with font as specified. However, if there is no suitable character that has weight weight in the specified radix, the value is NIL. If the "digit" is a letter (if weight is > 9), it is upper case.

CHAR-INT char

Returns the integer whose pointer field matches char.

INT-CHAR integer

Returns the character object whose pointer field matches integer.

CHAR-UPCASE char
CHAR-DOWNCASE char

When given a character object, these functions now return a character object. When given an integer, they still return an integer.

3.8.4 Character Names

CHAR-NAME char

Returns the standard name (or one of the standard names) of char, or NIL if there is none. The name is returned as a string. (CHAR-NAME #\SPACE) is the string "SPACE". If char has nonzero bits, the value is NIL. Names such as Control-X are not constructed by this function.

NAME-CHAR symbol

Returns the character for which symbol is a name, as a character object, or returns NIL if symbol is not recorded as a character name. Compound names such as Control-X are not recognized. Strings are allowed as well as symbols.

3.9 Hash Tables

The base flavor for hash tables is now called HASH-TABLE rather than EQ-HASH-TABLE. EQL hash tables now exist standardly as well as EQ and EQUAL hash tables. There are these new functions:

HASH-TABLE-P object

> T if object is a hash table. (TYPEP object 'HASH-TABLE).

HASH-TABLE-COUNT hash-table

> The number of filled entries in hash-table.

The function MAKE-HASH-TABLE takes new arguments:

> The keyword argument rehash-threshold may now be an integer, in which case it is the exact number of filled entries at which a rehash should be done. If so, it will be increased in proportion to the hash table size when teh rehash happens. The threshold can still be a flonum between zero and one, interpreted as a fraction of the total size.
>
> The new keyword argument test can be used to specify the type of hashing. It must be EQ, EQL or EQUAL.

3.10 Lists

TREE-EQUAL x y &key test test-not

> Compares two trees recursively to all levels. Atoms must match under the function test (which defaults to EQL). Conses must match recursively in both the car and the cdr.
>
> If test-not is specified instead of test, two atoms match if test-not returns NIL.

ENDP list

> Returns T if list is NIL, NIL if list is a cons cell. Gets an error if list is not a list. This is

the way Common LISP recommends for terminating a loop that cdr's down a list. However, Lambda system functions generally prefer to test for the end of the list with ATOM; it is regarded as a feature that these functions do something useful for dotted lists.

REST list

Is a synonym for CDR.

LIST-LENGTH list

Returns the length of list, or NIL if list is circular.

The function LENGTH would loop forever if given a circular list.

COPY-LIST, COPY-ALIST, COPY-TREE

Common LISP names for COPYLIST, COPYALIST, COPYTREE.

REVAPPEND list tail

Like (NCONC (REVERSE list) tail), but a little faster.

BUTLAST list &optional (n 1)

Returns a list like list but missing the last n elements.

NBUTLAST list &optional (n 1)

Modifies list to remove the last n elements, by changing a cdr pointer, and then returns list.

CLI:SUBST new old tree &key test test-not key

Replaces with new every atom or subtree in tree that matches old, returning a new tree. List structure is copied as necessary to avoid clobbering parts of tree. This differs from the traditional SUBST function, which always copies the entire tree.

test or test-not is used to do the matching. If test is specified, a match happens when test returns non-NIL; otherwise, if test-not is specified, a match happens when it returns NIL. If neither is specified, then EQL is used for test.

The first argument to the test or test-not function is always old. The second argument is normally a leaf or subtree of tree. However, if key is non-NIL, then it is called with the subtree as argument, and the result of this is passed to the test or test-not function.

Because (SUBST NIL NIL tree) is a widely used idiom for copying a tree, even though it is obsolete, it will be impractical to install a new function as the standard SUBST for a long time.

NSUBST new old tree &key test test-not key

Like CLI:SUBST but modifies tree itself and returns it. No new list structure is created.

SUBST-IF new predicate tree &key key

Replaces with new every atom or subtree in tree that satisfies predicate. List structure is copied as necessary so that the original tree is not modified. key, if non-NIL, is a function applied to each element to get the object to match against. If key is NIL, the element itself is used.

SUBST-IF-NOT new predicate tree &key key

Similar but replaces tree nodes that do not satisfy predicate.

NSUBST-IF, NSUBST-IF-NOT

Just like SUBST-IF and SUBST-IF-NOT except that they modify tree itself and return it.

SUBLIS alist tree &key test test-not key

Performs multiple parallel replacements on tree, returning a new tree. tree itself is not modified because list structure is copied as necessary. Each element of alist specifies one replacement; the car is what to look for, and the cdr is what to replace it with.

test or test-not is used to do the matching. If test is specified, a match happens when test returns non-NIL; otherwise, if test-not is specified, a match happens when it returns NIL. If neither is specified, then EQL is used for test.

The first argument to test or test-not is the car of an element of alist. The second argument is normally a leaf or subtree of tree. However, if key is non-NIL, then it is called with the subtree as argument, and the result of this is passed to the test or test-not function.

NSUBLIS alist tree &key test test-not key

Is like SUBLIS but modifies tree and returns it.

CLI:MEMBER item list &key test test-not key

Searches the elements of list for one which matches item, then returns the tail of list whose car is that element. If no match is found, NIL is returned.

test, test-not and key are used in matching the elements, just as described under CLI:SUBST. If neither test nor test-not is specified, the default is to compare with EQL. For this reason, CLI:MEMBER is thoroughly incompatible with traditional MEMBER, which uses EQUAL for the comparison.

MEMBER-IF predicate list &key key

Searches the elements of list for one which satisfies predicate. If one is found, the value is the tail of list whose car is that element. Otherwise the value is NIL.

If key is non-NIL, then predicate is applied to (FUNCALL key element) rather than to the element itself.

MEMBER-IF-NOT predicate list &key key

Searches for an element that does not satisfy predicate. Otherwise like MEMBER-IF.

ADJOIN item list &key test test-not key

Returns a list like list but with item as an additional element if no existing element matches item. It is done like this:

```
(IF (MEMBER item list other-args...)
    list
    (CONS item list))
```

PUSHNEW item list-place &key test test-not key

PUSHes item onto list-place unless item matches an existing element of the value stored in that place. Much like

(SETF list-place (ADJOIN item list-place keyword-args...))

except for order of evaluation.

CLI:UNION list1 list2 &key test test-not key

Returns a list that has all the elements of list1 and all the elements of list2. If list1 and list2 have elements in common, these elements need appear only once in the resulting list. Elements are compared for this purpose using the test function or the test-not function, or using EQL if neither arguemnt was specified.

If key is non-NIL, then key is applied to each of the elements to be compared to get a key that is then passed to test or test-not. Thus, you can say that elements are duplicates if their cars are EQL by using CAR as key.

If there are duplicate elements within list1 itself, or within list2, then there may be duplicate elements in the result. Elements of each list are matched against elements of the other, but not against other elements of the same list.

CLI:NUNION list1 list2 &key test test-not key

Like UNION but modifies list1, list2 or both to get the cells to make the list that is returned.

CLI:INTERSECTION list1 list2 &key test test-not key

Returns a list that has all the elements of list1 that match some element of list2. test, test-not and key are used in comparing elements just as they are used in UNION. If list1 contains duplicate elements, the duplicates can both appear in the result, as elements of list1 are not compared against other elements of list1.

CLI:NINTERSECTION list1 list2 &key test test-not key

Like INTERSECTION but destructively modifies list1 to produce the value.

SET-DIFFERENCE list1 list2 &key test test-not key

Returns a list that has all the elements of list1 that do not match any element of list2. test, test-not and key are used in comparing elements just as they are used in UNION. If list1 contains duplicate elements, the duplicates can both appear in the result, as elements of list1 are not compared against other elements of list1.

NSET-DIFFERENCE list1 list2 &key test test-not key

Like SET-DIFFERENCE but destructively modifies list1 to produce the value.

SET-EXCLUSIVE-OR list1 list2 &key test test-not key

Returns a list that has all the elements of list1 that do not match any element of list2, and also all the elements of list2 that do not match any element of list1. test, test-not and key are used in comparing elements just as they are used in UNION. If either list contains duplicate elements, the duplicates can both appear in the result, as comparisons are done only between an element of list1 and an element of list2.

NSET-EXCLUSIVE-OR list1 list2 &key test test-not key

Like SET-EXCLUSIVE-OR but may destructively modify both list1 and list2 to produce the value.

SUBSETP list1 list2 &key test test-not key

T if every element of list1 matches some element of list2. test, test-not and key are used in comparing elements.

PAIRLIS cars cdrs &optional tail

Returns (NCONC (MAPCAR 'CONS cars cdrs) tail).

ACONS acar acdr tail

Returns (CONS (CONS acar acdr) tail).

CLI:ASSOC item alist &key test test-not

Returns the first element of alist whose car matches item, or NIL if there is no such element. Elements that are NIL are ignored; they do not result in comparing item with NIL.

test and test-not are used in comparing elements.

This differs from the traditional function ASSOC in that by default it uses EQL rather than EQUAL for the comparison.

CLI:RASSOC item alist &key test test-not

Like CLI:ASSOC but compares against the cdr of each element rather than the car.

ASSOC-IF predicate alist

Returns the first element of alist whose car satisfies predicate, or NIL if there is no such element. Elements that are NIL are ignored; they do not result in applying predicate to NIL.

ASSOC-IF-NOT predicate alist

Returns the first element of alist whose car does not satisfy predicate, or NIL if there is no such element. Elements that are NIL are ignored; they do not result in applying predicate to NIL.

RASSOC-IF predicate alist
RASSOC-IF-NOT predicate alist

Like RASSOC-IF and RASSOC-IF-NOT but test the cdr of each element rather than the car.

MAKE-LIST

This function now takes a keyword argument initial-element, which specifies the value to store in each word of the newly made list. The old name for this argument was initial-value. Both names are accepted.

Chapter 4

Arrays

Traditionally the elements of a string are fixnums that represent characters. According to Common LISP, the elements of a string are character objects. Therefore, a different version of AREF has been provided for Common LISP programs. This version, CLI:AREF, returns a character object when the first argument is a string. It behaves just like AREF on arrays other than strings.

An array that allows arbitrary elements is called a general array. An array whose elements are restricted to a certain type is a specialized array. The only specialized arrays in the LISP machine system are strings, whose elements are characters, and numeric arrays, whose elements are restricted to be numbers (of particular types).

An array of rank one is called a vector. There are many new functions, called the generic sequence functions, which work equally well on vectors and on lists.

VECTORP object

T if object is a vector.

BIT-VECTOR-P object

T if object is a bit vector, an array of type ART-1B and rank-1.

SIMPLE-VECTOR-P object

T if object is a simple general vector; a rank-1 array that is not displaced and has no fill pointer, and whose elements may be any LISP object.

SIMPLE-BIT-VECTOR-P object

T if object is a simple bit vector; a rank-1 ART-1B array that is not displaced and has no fill pointer.

SIMPLE-STRING-P object

T if object is a simple string; a rank-1 ART-STRING or ART-FAT-STRING array that is not displaced and has no fill pointer.

MAKE-ARRAY takes three new keyword arguments:

- initial-element specifies a value to which each array element may be initialized. It is equivalent to the initial-value argument, which is still accepted.

- element-type is a new way to specify the array type. Its value is a Common LISP type specifier. The array type used is the most specialized that can allow as an element anything that fits the type specifier. For example, if element-type is (MOD 4), you will get an ART-2B array. Specifying element-type is an alternative to specifying type.

- initial-contents specifies the entire contents for the new array, as a list of lists of lists... Array element 1 3 4 of a three-dimensional array would be (NTH 4 (NTH 3 (NTH 1 initial-contents))).

adjustable-p

Is another argument that is allowed for compatibility with other Common LISP implementations. A non-NIL adjustable-pn says that the array should be made so that its size can be changed later. On the Lambda, any array size can be changed.

ARRAY-RANK-LIMIT Constant

A constant giving the limit on the rank of an array. It is 8, indicating that 7 is the highest possible rank.

ARRAY-DIMENSION-LIMIT Constant

Any one dimension of an array must be smaller than this constant.

ARRAY-TOTAL-SIZE-LIMIT Constant

The total number of elements of any array must be smaller than this constant.

VECTOR &rest elements

Creates and returns a general vector whose elements are as specified.

ARRAY-ELEMENT-TYPE array

Returns a type specifier that describes what elements could be stored in array.

Thus, if array is a string, the value is STRING-CHAR. If array is an ART-1B array, the value is BIT. If array is an ART-Q array, the value is T (the type that all objects belong to).

ARRAY-TOTAL-SIZE array

The total number of elements in array. The same as ARRAY-LENGTH.

ARRAY-ROW-MAJOR-INDEX array &rest indices

Calculates the cumulative index in array of the element at indices indices.

(AR-1-FORCE array (ARRAY-ROW-MAJOR-INDEX array indices...)) is equivalent to (AREF array indices...).

SVREF vector index

A special accessing function defined by Common LISP to work only on simple general vectors. Some other LISP systems may be able to open code SVREF so that it is faster than AREF, but on the LISP machine SVREF is a synonym for AREF.

BIT bit-vector index
SBIT bit-vector index
CHAR bit-vector index
SCHAR bit-vector index

Special accessing functions defined to work only on bit vectors, only on simple bit vectors, only on strings, and only on simple strings, respectively. On the Lambda, they are all synonyms for AREF.

BIT-AND bit-array-1 bit-array-2 &optional result-bit-array
BIT-IOR bit-array-1 bit-array-2 &optional result-bit-array
BIT-XOR bit-array-1 bit-array-2 &optional result-bit-array
BIT-EQV bit-array-1 bit-array-2 &optional result-bit-array
BIT-NAND bit-array-1 bit-array-2 &optional result-bit-array
BIT-NOR bit-array-1 bit-array-2 &optional result-bit-array
BIT-ANDC1 bit-array-1 bit-array-2 &optional result-bit-array
BIT-ANDC2 bit-array-1 bit-array-2 &optional result-bit-array
BIT-ORC1 bit-array-1 bit-array-2 &optional result-bit-array

BIT-ORC2 bit-array-1 bit-array-2 &optional result-bit-array

Perform boolean operations element by element on bit arrays. The arguments must match in their size and shape, and all of their elements must be integers. Corresponding elements of bit-array-1 and bit-array-2 are taken and passed to one of LOGAND, LOGIOR, ... to get an element of the result array.

If the third argument is non-NIL, the result bits are stored into it, modifying it destructively. Otherwise a new ART-1B array is created and used for the result. In either case, the value returned is the array where the results are stored.

Common LISP defines these operations only when all arguments are specialized arrays that hold only bits (ART-1B arrays, in the Lambda).

BIT-NOT bit-array &optional result-bit-array

Performs LOGNOT on each element of bit-array to get a result bit.

If result-bit-array is non-NIL, the result bits are stored in it; it must match bit-array in size and shape. Otherwise, a new ART-1B array is created and used to hold the result.

Each bit of the result is 0 if the argument was 1.

ARRAY-HAS-FILL-POINTER-P array

T if array has a fill pointer. It must have a leader and leader element 0 must be an integer.

VECTOR-PUSH new-element vector
VECTOR-PUSH-EXTEND new-element vector &optional amount

Exactly like ARRAY-PUSH and ARRAY-PUSH-EXTEND except that the first two arguments are interchanged.

VECTOR-POP vector

A synonym for ARRAY-POP.

ADJUST-ARRAY array new-dimensions &key element-type
initial-element initial-contents fill-pointer displaced-to
displaced-index-offset

Modifies various aspects of an array. array is modified in place if that is possible; otherwise, a new array is created and array is forwarded to it. In either case, array is returned. The arguments have the same names as arguments to MAKE-ARRAY, and signify approximately the same thing.

However: element-type is just an error check. ADJUST-ARRAY cannot change the array type. If the array type of array is not what element-type would request, you get an error.

If displaced-to is specified, the new array is displaced as specified. If array itself was displaced, it is modified in place provided that either

- array has an index offset and displaced-index-offset is non-NIL, or

- array has no index offset and displaced-index-offset is NIL.

Otherwise, if initial-contents was specified, it is used to set all the contents of the array. The old contents of array are irrelevant.

Otherwise, each element of array is copied forward into the new array to the slot with the same indices, if there is one. Any new slots whose indices were out of range in array are initialized to initial-element, or to NIL or 0 if initial-element was not specified.

fill-pointer, if specified, is used to set the fill pointer of the array.

Chapter 5

String Functions

CLI:STRING string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING//= string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING CTRL-Z string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING< string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING> string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING>= string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING>= string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING<= string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING CTRL-\ string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING CTRL-] string1 string2 &optional (start1 0) end1 (start2 0) end2

Compares all or portions of string1 and string2, not ignoring case or font. For STRING=, the value is T when the strings match completely. For the other functions, when the condition is met, the value is the index in string1 of the point of first difference.

There is a distinct Common LISP version of STRING= because a function STRING= already exists with the same purpose but a different calling sequence.

Note that in Common LISP syntax you would write STRING/=, not STRING//=.

CLI:STRING-EQUAL string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING-NOT-EQUAL string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING-LESSP string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING-GREATERP string1 string2 &optional (start1 0) end1 (start2 0) end2

STRING-NOT-GREATERP string1 string2 &optional (start1 0) end1 (start2 0) end2
STRING-NOT-LESSP string1 string2 &optional (start1 0) end1 (start2 0) end2

Compares all or portions of string1 and string2, ignoring case and font. For STRING-EQUAL, the value is T when the strings match completely. For the other functions, when the condition is met, the value is the index in string1 of the point of first difference.

There is a distinct Common LISP version of STRING-EQUAL because a function STRING-EQUAL already exists with the same purpose but a different calling sequence.

MAKE-STRING size &key (initial-element 0)

Creates and returns a string of length size, with each element initialized to initial-element.

STRING-UPCASE string &key (start 0) end

Makes and returns a copy of string in which all, or the specified portion, has been converted to upper case.

The value may be string itself if no characters in string require conversion.

STRING-DOWNCASE string &key (start 0) end

Makes and returns a copy of string in which all, or the specified portion, has been converted to lower case.

The value may be string itself if no characters in string require conversion.

STRING-CAPITALIZE string &key (start 0) end

Makes and returns a copy of string in which all, or the specified portion, has been processed by capitalizing each word. For this function, a word is any maximal sequence of letters or digits. It is capitalized by putting the first character (if it is a letter) in upper case and any letters in the rest of the word in lower case.

The value may be string itself if no characters in string require conversion.

NSTRING-UPCASE string &key (start 0) end
NSTRING-DOWNCASE string &key (start 0) end
NSTRING-CAPITALIZE string &key (start 0) end

Like the previous functions except that they modify string itself and return it.

Chapter 6

I/O Streams

Common LISP speaks of objects called "I/O streams", but as far as Common LISP is concerned these are simply whatever you can legitimately pass to functions such as READ and READ-CHAR (the Common LISP replacement for TYI) and PRINT and WRITE-CHAR (which replaces TYO). Common LISP has no concept of streams as message-handling objects, or of users defining their own types of streams. It has a few standard functions that produce streams.

Of course, on the Lambda, the streams still are message-handling objects. You can still define streams just as before, and they will work properly with the Common LISP I/O functions (which all work by invoking standard stream operations).

STANDARD-INPUT — Variable

An alias for STANDARD-INPUT

STANDARD-OUTPUT, *TERMINAL-IO*, *QUERY-IO*
DEBUG-IO, *TRACE-OUTPUT*, *ERROR-OUTPUT* — Variables
more synonyms.

MAKE-SYNONYM-STREAM symbol

Returns a stream that does its work by invoking the value of symbol as a stream. This is a new name for MAKE-SYN-STREAM, and therefore works on locatives too.

MAKE-CONCATENATED-STREAM &rest streams

Returns an input stream that will read its input from the first of streams until that reaches its eof, then read input from the second of streams, and so on until the last of streams has reached eof.

MAKE-TWO-WAY-STREAM input-stream output-stream

Returns a bidirectional stream that passes input operations to input-stream and passes output operations to output-stream.

This works by attempting to recognize all standard input operations; anything not recognized is passed to output-stream.

MAKE-ECHO-STREAM input-stream output-stream

Like MAKE-TWO-WAY-STREAM except that each input character read via input-stream is output to output-stream before it is returned to the caller.

MAKE-STRING-INPUT-STREAM string &optional (start 0) end

Returns a stream that can be used to read the contents of string (or the specified portion of it) as input. Eof will occur on reading past position end or the end of string.

MAKE-STRING-OUTPUT-STREAM string

Returns an output stream that will accumulate all output in a string.

GET-OUTPUT-STREAM-STRING string-output-stream

Returns the string of output accumulated so far by a stream that was made by calling MAKE-STRING-OUTPUT-STREAM. The accumulation is reset, so the output obtained this time will not be obtained again if GET-OUTPUT-STREAM-STRING is called again later on the same stream.

STREAMP object

According to Common LISP, T if object is a stream. In the Lambda, a stream is any object that can be called as a function with certain calling conventions. It is theoretically impossible to test for this. However, STREAMP does return T for any of the usual types of streams, and NIL for any Common LISP datum that is not a stream.

INPUT-STREAM-P stream

T if stream handles input operations (at least, if it handles :TYI).

OUTPUT-STREAM-P stream

T if stream handles output operations (at least, if it handles :TYO).

STREAM-ELEMENT-TYPE stream

Returns a type specifier that describes the the typical object input from or output to stream. The value is always a subtype of INTEGER or a subtype of CHARACTER. If it is a subtype of CHARACTER, a Common LISP program should use READ-CHAR or WRITE-CHAR; otherwise it should use READ-BYTE or WRITE-BYTE.

However, the value returned is not intended to be rigidly accurate. Typical data transferred will fit the type, but there may be unusual cases in which other data are transferred that do not fit it; also, not all objects of the type may be possible as input or even make sense as output. The element type may be CHARACTER, FIXNUM or UNSIGNED-BYTE if that is as much as the system knows about the stream, even if in fact only some characters or only bytes of a certain size really mean anything.

CLI:CLOSE stream &key abort

Like traditional CLOSE but the calling sequence is different. If abort is non-NIL, the file (if it is being written) is not kept around.

6.1 Functions Changed

WITH-INPUT-FROM-STRING (var string &key index start end) body...

The calling sequence is changed. start and end specify, optionally, a portion of string to be read. index, if specified, is a variable in which the current index of reading in string will be stored when the WITH-INPUT-FROM-STRING is exited.

The old calling sequence was

(var string index end) body...

This sequence is still recognized, for the sake of compatibility.

Chapter 7

The Reader; Input Functions

7.1 Syntax Extensions

These extensions to the reader syntax are upward compatible and apply to Common LISP and traditional syntax alike.

...|...|...

Vertical bars can now be used within a symbol, to quote some portion of it.

For example, AB| ... "quoted stuff" :; |CDE is one symbol. AB|cd|EF|gh| is also one symbol. It used to be the case that if vertical bars were used in a symbol they had to go around the whole thing.

#(...)

Signifies a vector. It can contain any number of elements, of any type.

Thus, #(A 5 "Foo") reads as a vector containing a symbol, an integer and a string.

The vector created will always be of type ART-Q.

#*bbb...

Signifies a bit vector; bbb... are the bits (characters "1" or "0").

A vector of type ART-1B is created and filled with the specified bits, starting with element 0. The length is however many bits you specify. Alternatively, specify the length with a decimal number between the # and the *. The last "1" or "0" specified is duplicated to fill the additional bits.

#O causes numbers in the following object to be read in octal.
#X causes numbers in the following object to be read in hex.
#B causes numbers in the following object to be read in binary.

#nA(...contents...)

Signifies an array of rank n, containing contents. The entire list in which the contents appear is passed as the initial-contents arg to MAKE-ARRAY to produce the array. The array dimensions are determined from the contents as well. The rank is specified explicitly so that the reader can distinguish whether a list in the contents is a list of array elements or a single array element. The array type is always ART-Q.

#S(type slot value slot value slot value ...)

Signifies a structure of type type. Any structure type defined with DEFSTRUCT can be used as type provided it has a standard constructor macro taking slot values as keyword arguments. The slot names and values appearing in the read syntax are passed to the constructor so that they initialize the structure.

#C(real imag)

Is a new input syntax for complex numbers, equivalent to real+imagi. Unfortunately, the superior traditional LISP Machine syntax is not allowed in Common LISP files. It is still allowed in non-Common-LISP files, of course.

#(...)

Within a backquote expression means "construct a vector" just as (...) within a backquote expression means "construct a list". Thus, `(A #(B ,C)) expands into something like (LIST 'A (VECTOR 'B C)).

7.1.1 Syntax for circular structure

Syntax for circular structure:

#n# refers to the object with label n. n is a numeral made of decimal digits.

#n= says that the object that follows is the antecedent of label n.

Thus, #1=(A . #1#) is a way of notating a circular list such as would be produced by (CIRCULAR-LIST 'A). The cdr of this list is the list itself.

7.1.2 Syntax for floating point numbers

Syntax for floating point numbers:

Common LISP defined four sizes of floating point numbers, but in a given implementation they need not all be distinct. The four sizes are called SHORT-FLOAT, LONG-FLOAT, SINGLE-FLOAT and DOUBLE-FLOAT. They are specified in read syntax by the use of S, L, F or D to start the exponent (thus, 1.5L6 indicates a LONG-FLOAT). On the LISP machine, there are really only two sizes: SHORT-FLOAT (also known as small flonums) and SINGLE-FLOAT (also known as regular flonums). If you ask to make a double or long float, you get a single-float.

The exponent can also be delimited with E; this does not specify the size of float, just as a number with no exponent (such as 1.5) does not. Then the variable *READ-DEFAULT-FLOAT-FORMAT* determines the size of flonum used. On output, flonums of the default size are printed with no exponent (if that's convenient) or with E, and flonums of the other size are printed with a letter that indicates the size (S or F).

7.1.3 Common LISP versus Traditional Read Syntax

These are the incompatible changes in reader syntax that apply only to programs marked as Common LISP. There is no incompatible change to the syntax of existing programs.

/	Is a symbol constituent, just like A and =.
\	Is the single-character-quote character.
#\	Produces a character object rather than a fixnum representing a character code.
#/	Is not defined as reader syntax.

Rational numbers are written with / rather than \, as in 1/2, and they are read using the current radix rather than always decimal. Thus, the #B, #O, etc. prefixes affect them.

Complex numbers must be written with the new #C syntax, as the traditional 1+3i syntax is not allowed in Common LISP.

7.2 Readtables, etc.

READTABLE Variable

An alias for the variable READTABLE. The value is the current readtable.

COPY-READTABLE &optional from-readtable to-readtable

Copies the contents of from-readtable into to-readtable. to-readtable is returned.

If to-readtable is NIL, a new readtable is constructed, the contents of from-readtable are copied into it, and it is returned.

If from-readtable is omitted, it defaults to something. (The Common LISP Manual contradicts itself on what this something should be.)

READTABLEP object

T if object is a readtable.

SET-MACRO-CHARACTER char function &optional non-terminating-p in-readtable

Sets the syntax of character char in readtable in-readtable to be that of a macro character that is handled by function. When that character is read by READ, function is called. The arguments given to function are the input stream READ is reading from, and the character just read (ie. char).

function should return zero or more values, which are the objects that the macro construct "reads as". Zero values causes the macro construct to be ignored (the semicolon macro character would do this), and one value causes the macro construct to read as a single object (most macro characters do this). More than one value is allowed only within a list.

If non-terminating-p is non-NIL, then char will be recognized as a macro character only at the start of a token. If found in the middle of a symbol, it will be alphabetic.

is a non-terminating macro character.

GET-MACRO-CHARACTER char in-readtable

Returns two values that describe the macro character status of char in in-readtable.

If char is not a macro character, both values are NIL. Otherwise, the first value is the function and the second value is the non-terminating-p for this character.

Those two values, passed to SET-MACRO-CHARACTER, are sufficient to specify exactly the syntax that char currently has.

MAKE-DISPATCH-MACRO-CHARACTER char &optional non-terminating-p in-readtable

Makes char be a dispatch macro character in in-readtable. This means that when char is seen, the reader will read one more character to decide what to do.

is an example of a dispatch macro character.

non-terminating-p means the same thing as in SET-MACRO-CHARACTER.

SET-DISPATCH-MACRO-CHARACTER char subchar function &optional in-readtable

Sets the syntax of the two-character sequence char subchar, assuming that char is a dispatch macro character like #. When this two-character sequence is seen by READ, it will call function with three arguments:

- the input stream,
- the subchar that caused function to be invoked,
- the infix argument (in #5R, this is the number 5) or NIL if there is no infix argument this time.

If subchar is lower case, it is converted to upper case. Case is never significant for the character that follows a dispatch macro character. The decimal digits may not be defined as subchars since they are always used for infix numeric arguments as in "#5R".

GET-DISPATCH-MACRO-CHARACTER char subchar &optional in-readtable

> Returns the function for subchar following dispatch macro character char in readtable in-readtable. The value is NIL if subchar is not defined.

SET-SYNTAX-FROM-CHAR to-char from-char &optional to-readtable from-readtable

> Copies the syntax of from-char in from-readtable to character to-char in to-readtable.
>
> from-readtable defaults to the current readtable and to-readtable defaults to standard Common LISP syntax.

Common LISP has an unusual idea of what it means to copy the syntax of a character. The only aspect of syntax that the readtable supposedly specifies is the choice among token constituent (digits, letters, random things like @, !, $, and also colon!):

- Whitespace
- Escape character (/ traditionally, \ in Common LISP)
- Multiple escape character (vertical bar)
- Macro character, which includes characters ()",.'`;
- Nonterminating macro character (#)

The differences among macro characters are determined entirely by the functions that they invoke. The differences among token constituents (including the difference between A and colon) are fixed! You can make A be a macro character, or whitespace, or a quote character, but if you make it a token constituent then it always behaves the way it normally does. Likewise, if you make open-paren into a token constituent, there is only one kind of token constituent it can be (it forces the token to be a symbol, like $ or @ or %).

So, if from-char is some kind of token constituent, this function makes to-char into a token constituent of the kind that to-char is supposed to be--not the kind of token constituent that from-char is.

This is precisely how SET-SYNTAX-FROM-CHAR differs from the traditional function COPY-SYNTAX. COPY-SYNTAX would make to-char have exactly the same syntactic properties that from-char has now.

7.3 Input Functions

In all of these functions, the argument stream defaults to STANDARD-INPUT.

READ-BASE Variable

An alias for IBASE.

CLI:READ &optional stream (eof-errorp T) eof-value recursive-p

This is like READ but with slightly different arguments. End of file is an error if eof-errorp is non-NIL. Otherwise, end of file not in the middle of an object causes CLI:READ to return eof-value. End of file in the middle of an object is always an error.

recursive-p should be non-NIL for calls to READ from macro characters. This affects the processing of #n# and #n= labels, and of trailing whitespace.

READ-PRESERVING-WHITESPACE &optional stream (eof-errorp T) eof-value recursive-p

Like CLI:READ but binds READ-PRESERVE-DELIMITERS to T. This is the Common LISP way of requesting the READ-PRESERVE-DELIMITERS feature.

READ-DELIMITED-LIST char &optional stream recursive-p

reads s-expressions from stream until the character char is seen at top level, then returns a list of the objects read.

For example, if char is], and the text to be read from stream is A (B C)] ... then the objects A and (B C) will be read, the] will be seen as a terminator and discarded, and the value will be (A (B C)). recursive-p works like CLI:READ, affecting only #n= labels. End of file within this function is always an error since it is always "within an object".

READ-DEFAULT-FLOAT-FORMAT Variable

Its value is the type for read to produce by default for flonums whose precise type is not specified by the syntax. The value should be either

GLOBAL:SMALL-FLOAT or GLOBAL:SINGLE-FLOAT, these being the only distinct floating formats that the LISP machine has.

READ-LINE &optional stream

A synonym for READLINE.

READ-CHAR &optional stream (eof-errorp T) eof-value

Reads a character from stream and returns it as a character object. End of file is an error if eof-errorp is non-NIL; otherwise, it causes READ-CHAR to return eof-value. Uses the :TYI stream operation.

READ-CHAR-NO-HANG &optional stream (eof-errorp T) eof-value

Similar but returns NIL immediately when no input is available on an interactive stream. Uses the :TYI-NO-HANG stream operation.

UNREAD-CHAR char &optional stream

Untyi's char on stream. char may be an integer or a character object.

Uses the :UNTYI stream operation.

PEEK-CHAR peek-type &optional stream (eof-errorp T) eof-value

If peek-type is NIL, this is like READ-CHAR, except it leaves the character to be read again by the next input operation.

If peek-type is T, it skips whitespace characters and peeks at the first nonwhitespace character. That character is the value, and is also left to be reread.

If peek-type is a character, it reads input until that character is seen. That character is unread and also returned.

LISTEN &optional stream

T if input is available on stream. Uses the :LISTEN operation.

CLEAR-INPUT &optional stream

Discards any input now available on stream, if it is an interactive stream. Uses the :CLEAR-INPUT stream operation.

CLI:READ-FROM-STRING string &optional (eof-errorp T) eof-value &key (start 0) end preserve-whitespace

Reads input from the contents of string, or the portion of it specified by start and end. The value of this function is the result of calling READ. eof-errorp and eof-value are passed to READ. If preserve-whitespace is non-NIL, READ-PRESERVING-WHITESPACE is used. This function differs from READ-FROM-STRING in having some additional arguments.

PARSE-INTEGER string &key (start 0) end (radix 10.) junk-allowed

Parses the contents of string (or the portion from start to end) as a numeral for an integer, and returns the integer it describes, using the specified radix.

Radices larger than ten are allowed, and they use letters as digits beyond 9.

Leading whitespace is always allowed and ignored. A leading sign is also allowed and considered part of the number.

When junk-allowed is NIL, the entire specified portion of string must consist of an integer and leading and trailing whitespace. Otherwise, an error happens.

If junk-allowed is non-NIL, parsing just stops when a non-digit is encountered. The number parsed so far is returned as the first value, and the index in string at which parsing stopped is returned as the second value. This number equals end (or the length of string) if there is nothing but a number.

If non-digits are found without finding a number first, the first value is NIL.

READ-BYTE stream &optional (eof-errorp T) eof-value

Like TYI except for handling the eof arguments just like the other functions above.

Chapter 8

The Printer; Output Functions

There are now many special variables that you can bind to control various options of printing.

PRINT-ESCAPE Variable

If non-NIL, quoting characters (slashes or backslashes) are printed where necessary to make the output readable by LISP. Most of the output functions bind this variable to T or to NIL, so you rarely use the variable itself. PRINC binds the variable to NIL and all the other output functions bind it to T.

PRINT-BASE Variable

The radix to use for printing integers. This is a synonym for BASE. Its default value is supposed to be ten, but it is actually eight at present. If the Common LISP readtable is in use, this radix is used for printing ratios as well; the standard readtable currently always prints ratios in decimal.

PRINT-RADIX Variable

If non-NIL, integers and ratios are output with something to indicate the radix that was used to print them. The radix can be indicated with a prefix such as #X or #3R, or (for an integer) with a trailing decimal point. This is a replacement, though not a synonym, for *NOPOINT. If *NOPOINT is NIL then the value of this variable is irrelevant. The normal means for selection of Common LISP syntax also make *NOPOINT be T and thus make *PRINT-RADIX* take effect.

PRINT-CIRCLE Variable

If non-NIL, the printer recognizes circular structure and prints it using #n= labels so that it has a finite printed representation (which can be read back in). The default is NIL, since it makes printing slower.

PRINT-PRETTY Variable

If non-NIL, the printer actually calls GRIND-TOP-LEVEL so that it prints extra whitespace for the sake of formatting. The default is NIL. Currently the grinder does not know how to detect circular structure, so *PRINT-CIRCLE* is ignored in this case.

PRINT-GENSYM Variable

If non-NIL, uninterned symbols are printed with the prefix #: to mark them as such (but only when *PRINT-ESCAPE* is non-NIL). If NIL, no prefix is used for uninterned symbols. The default is T.

PRINT-ARRAY Variable

If non-NIL, arrays are printed in the new #(...), #* or #nA(...) syntax so that you can see their contents. The default is NIL.

PRINT-CASE Variable

Controls the case used for printing upper-case letters in the names of symbols. Its value should be :UPCASE, :DOWNCASE or :CAPITALIZE. These mean, respectively, to print those letters as upper case, to print them as lower case, or to capitalize each word (see STRING-CAPITALIZE). Any lower case letters in the symbol name will be printed as lower case and quoted suitably; this flag does not affect them. Note that the case used for printing the upper case letters has no effect on reading the symbols back in, since they are case-converted by READ.

PRINT-LEVEL, *PRINT-LENGTH* Variable

Aliases for PRINLEVEL and PRINLENGTH.

8.1 Print Functions

The argument stream always defaults to STANDARD-OUTPUT.

PRIN1
PRINT object &optional stream
PRINC object &optional stream
PRIN1-THEN-SPACE object &optional stream

These functions are unchanged, and mentioned just to remind you of them.

PPRINT object &optional stream

Like PRINT except that it binds *PRINT-PRETTY* to T so that the grinder is used. Also, it returns zero values.

WRITE object &key stream escape radix base circle pretty level length case gensym array

Prints object on stream, having bound all the printing flags according to the other keyword arguments. Thus, the keyword argument array specifies how to bind *PRINT-ARRAY*; if array is omitted, the ambient value of *PRINT-ARRAY* is used. This function is redundant given that the flag variables themselves are advertised for users to bind. The value is object.

WRITE-TO-STRING object &key escape radix base circle pretty level length case gensym array

Like WRITE but puts the output in a string and returns the string.

PRIN1-TO-STRING object
PRINC-TO-STRING object

Like PRIN1 and PRINC bit put the output in a string and return the string.

8.2 Output Functions

WRITE-CHAR char &optional stream

Outputs char to stream (using :TYO). char may be an integer or a character object; in the latter case, it is converted to an integer before the :TYO.

WRITE-STRING string &optional stream &key (start 0) end

Outputs string (or the specified portion) to stream.

WRITE-LINE string &optional stream &key (start 0) end

Outputs string (or the specified portion) to stream, followed by a Return character.

TERPRI &optional stream

Outputs a Return character to stream.

FRESH-LINE &optional stream

Outputs a Return character to stream unless either nothing has been output to stream yet, or the last thing output was a Return character, or stream cannot tell what previous output there has been. This uses the :FRESH-LINE stream operation. The value is T if a Return was output (because all :FRESH-LINE methods have been fixed to return this value).

FINISH-OUTPUT &optional stream
CLEAR-OUTPUT &optional stream
FORCE-OUTPUT &optional stream

Invoke the :FINISH, :CLEAR-OUTPUT and :FORCE-OUTPUT stream operations.

WRITE-BYTE number &optional stream

Outputs number to stream using :TYO.

Chapter 9

FORMAT Changes

There are some new format operations, and a few that are changed in Common LISP. There is a distinct symbol and function CLI:FORMAT that is used in Common LISP programs, and this is how the FORMAT package knows which interpretation to use for the format operations that are incompatible.

Here are the format operations that are new, and that work the same in all programs:

~B — Like ~O and ~D, but prints in binary.

~X — Like ~O and ~D, but prints in hex.

~@T — Spaces horizontally a specified amount.

> ~rel,period@T first outputs rel spaces, and then zero or more spaces to get to a column that is a multiple of period. If the stream being used cannot tell its cursor position, however, then period is ignored. Then this always outputs rel spaces.

~@ — Equivalent to the old (and current in non-Common-LISP) meaning of ~G.

~? — Indirect format string. This gobbles two of the arguments given to FORMAT, and uses the first as a FORMAT string and the second as a list of arguments for that string.

Example:
FORMAT "~? ~D" "~O ~O" '(4 20.) 9)   prints 4 24 9

~(...~) — Case converted output. The text within the construct is processed by FORMAT as usual, but all letters output are converted to lower case.

> ~:(...) converts case so that each word is capitalized.
> ~@(...) capitalizes the first word,

and converts all the rest of the output to lower case.
~:@(...) converts all letters to upper case.

Numeric parameters in the format string can now have minus signs.

Here are the format operations that are changed incompatibly for Common LISP programs only.

~E and ~F — Now take many hairy arguments that control formatting.

~G — Is now a new kind of floating point output mode; use ~@* to get the traditional meaning of ~G. For more information on these output modes, see the Common LISP manual.

Chapter 10

Pathnames

Common LISP does not change much in the nature of pathnames, but there are a few new functions.

PATHNAMEP object

T if object is a pathname.

PATHNAME object

Converts object to a pathname and returns that, if possible.
If object is a string or symbol, it is parsed.
If object is a plausible stream, it is asked for its pathname with the :PATHNAME operation.
If object is a pathname, it is simply returned.
Anything else is not allowed.

TRUENAME object

Returns the truename of the file specified somehow by object. If object is a plausible stream, it is asked for the truename with the :TRUENAME operation. Otherwise, object is converted to a pathname and that pathname is opened to get its file's truename.

```
PARSE-NAMESTRING thing &optional with-respect-to
    (defaults *DEFAULT-PATHNAME-DEFAULTS*)
    &key (start 0) end junk-allowed
```

Is equivalent to FS:PARSE-PATHNAME except it takes some keyword arguments where the other function takes all positional arguments.

PARSE-NAMESTRING parses thing into a pathname and returns it.

thing can be a pathname, a string or symbol, or a Maclisp-style namelist.

If it is a pathname, it is returned unchanged, and the other arguments do not matter.

with-respect-to can be NIL or a host or host-name; if it is not NIL, the pathname is parsed for that host and it is an error if the pathname specifies a different host.

If with-respect-to is NIL, then defaults is used to get the host if none is specified. defaults may be a host object in this case.

start and end are indices specifying a substring of thing to be parsed. They default to 0 for start and NIL (meaning end of thing) for end.

If junk-allowed is non-NIL, parsing stops without error if the syntax is invalid, and this function returns NIL. The second value is the index in thing at which parsing stopped, which is the index of the invalid character if there was invalid syntax.

If junk-allowed is NIL, invalid syntax signals an error.

DEFAULT-PATHNAME-DEFAULTS Variable

An alias for DEFAULT-PATHNAME-DEFAULTS. WARNING!! According to the Common LISP manual, this variable is supposed to have a pathname as its value. On the Lambda, the value is and has always been an alist of hosts versus pathnames. This is how the *DEFAULTS-ARE-PER-HOST* feature is implemented. A separate variable CLI:*DEFAULTS-ARE-PER-HOST* would not work: which one would be used for actual defaulting?

If you set this variable to a pathname as a way of setting the defaults, you will get into trouble.

The only way to protect against this is to flush the *DEFAULTS-ARE-PER-HOST* feature. So far, we have chosen not to do this; so be careful.

MERGE-PATHNAMES pathname &optional
(defaults *DEFAULT-PATHNAME-DEFAULTS*)
(default-version ':NEWEST)

Merges defaults from defaults into pathname to get a new pathname, which is returned.

pathname can be a string (or symbol); then it is parsed and the result is defaulted.

default-version is used as the version when pathname has a name but no version.

NAMESTRING pathname

Returns a string containing the printed form of pathname, as you would type it in. This uses the :STRING-FOR-PRINTING operation.

FILE-NAMESTRING pathname

Returns a string showing just the name, type and version of pathname. This uses the :STRING-FOR-DIRED operation.

DIRECTORY-NAMESTRING pathname

Returns a string showing just the device and directory of pathname. This uses the :STRING-FOR-DIRECTORY operation.

ENOUGH-NAMESTRING pathname &optional defaults

Returns a string showing just the components of pathname that would not be obtained by defaulting from defaults. This is the shortest string that would suffice to specify pathname, given those defaults. It is done by using the :STRING-FOR-PRINTING operation on a modified pathname.

USER-HOMEDIR-PATHNAME &optional (host FS:USER-LOGIN-MACHINE) reset-p user force-p

A synonym for FS:USER-HOMEDIR.

TRANSLATED-PATHNAME pathname

Returns the translation of pathname, the same as pathname unless it is a logical pathname. This uses the :TRANSLATED-PATHNAME operation. As of November 1983, this function is no longer part of the Common LISP specification.

BACK-TRANSLATED-PATHNAME logical-pathname actual-pathname

Returns a logical pathname whose host is that of logical-pathname and that would translate into actual-pathname. This uses the

:BACK-TRANSLATED-PATHNAME operation. As of November 1983, this function is no longer part of the Common LISP specification.

The following functions are now in GLOBAL rather than just in the FS package:

INIT-FILE-PATHNAME program-name &optional (host FS:USER-LOGIN-MACHINE)
PATHNAME-HOST pathname
PATHNAME-DEVICE pathname
PATHNAME-DIRECTORY pathname
PATHNAME-NAME pathname
PATHNAME-TYPE pathname
PATHNAME-VERSION pathname
PATHNAME-PLIST pathname
MAKE-PATHNAME &key host device directory name type version defaults raw-device raw-directory raw-name raw-type defaults canonical-type original-type (only host device directory name type version and defaults are standard Common LISP.)
ADD-LOGICAL-PATHNAME-HOST logical-host actual-host default-device translations

Chapter 11

File Access Functions

An argument named file can be a pathname, a string or symbol that can be parsed into a pathname, or a stream on which a file is open.

OPEN and WITH-OPEN-FILE have new keyword arguments that are the Common LISP replacements for the keyword arguments used so far. These are if-exists, if-does-not-exist, and element-type. In addition, the direction argument allows new values that are synonyms for the others.

For output opens, if-exists specifies what to do if a file with the specified name already exists. There are several values you can use:

:NEW-VERSION Means create a new version. This makes sense only when the pathname has :NEWEST as its version, and it is the default in that case.

:SUPERSEDE Means make a new file that, when closed, replaces the old one.

:OVERWRITE Means write over the data of the existing file, starting at the beginning, and set the file's length to the length of the newly written data.

:TRUNCATE Is like :OVERWRITE except that it does not free the disk storage allocated to the file. This might be faster.

:APPEND Means add new data onto the existing file at the end.

:RENAME Means rename the existing file and then create a new one.

:RENAME-AND-DELETE
Means rename the existing file, create a new one, and delete the old file when the new one is closed.

:ERROR — Means signal an error (FS:FILE-ALREADY-EXISTS). This is the default when the pathname's version is not :NEWEST.

NIL — Means return NIL from OPEN in this case.

if-does-not-exist specifies what to do when the file you ask for does not exist. There are three values you can use:

:CREATE — Means create a file. This is the default for output opens, except when you use :APPEND, :OVERWRITE or :TRUNCATE as the if-exists argument.

:ERROR — Means signal an error. This is the default for input opens, and also for output opens when you use :APPEND, :OVERWRITE or :TRUNCATE as the if-existsx argument.

NIL — Means return NIL from OPEN. This is the default for :PROBE opens.

element-type specifies what kind of objects the stream will want to read or write. This combines the effect of the characters and byte-size arguments. The value is a type specifier; it must be one of the following:

STRING-CHAR — Means read or write characters as usual. The default.

CHARACTER — Means read or write characters, dealing with characters that are more than 8 bits. You can succeed in writing out any sequence of character objects and reading it back, but the file does not look anything like a text file.

(UNSIGNED-BYTE n)
Means read or write n-bit bytes. Like characters = NIL, byte-size = n.

(SIGNED-BYTE n)
Means read or write n-bit bytes, sign extended. Each byte read from the file is sign-extended so that its top bit serves as a sign bit.

UNSIGNED-BYTE or SIGNED-BYTE
Is similar but specifies :DEFAULT as the

byte size. The file system will use the byte size of the file you have opened.

(MOD n) Same as UNSIGNED-BYTE for a big enough byte size to hold all numbers less than n. BIT is also accepted, and means (MOD 2).

:DEFAULT Is allowed, even though it is not a type specifier. It is the same as using :DEFAULT as the value of characters.

direction now allows the values :PROBE and :IO.

:PROBE Is the same as NIL, to do a probe open.

:IO Means you want a bidirectional stream (both input and output), but this is not now supported by any file system.

Two other direction values allowed are :PROBE-DIRECTORY and :PROBE-LINK.

:PROBE-DIRECTORY
Is used to see whether a directory exists. If the opened pathname's directory is found, then the OPEN completes (returning a non-I/O stream) as if the specified file exists, whether it really exists or not.

:PROBE-LINK Is used to find out the truename of a link. If the file specified exists as a link, then the OPEN completes returning a non-I/O stream that describes the link itself rather than the file linked to. If the file exists and is not a link, the OPEN also completes for it as with any probe.

DELETE-FILE file &key error query

Deletes the specified file. Like DELETEF, but takes its args keyword style.

UNDELETE-FILE file &key error query

Undeletes the specified file. This function is not Common LISP, but is useful.

RENAME-FILE file new-name &key error query

Renames file to have the name new-name. Like RENAMEF but takes keyword args.

COPY-FILE file new-name &key error ...lots more args...

Not Common LISP, but analogous to the others. See SYS98 MSG for all the args it takes.

PROBE-FILE pathname

If a file named pathname exists, returns the file's truename; otherwise, returns NIL. (Actually, failure to open the file with any error condition except FS:FILE-NOT-FOUND is not caught.)

FILE-WRITE-DATE file

Returns the creation date/time of file, as a universal time.

FILE-AUTHOR file

Returns the name of the author of file (the user who wrote it), as a string.

FILE-LENGTH file-stream

Returns the length of the file open on file-stream, in terms of the units in which I/O is being done on that stream. (A stream is needed, rather than just a pathname, in order to specify the units.)

FILE-POSITION file-stream &optional new-position

With one argument, returns the current position in the file of file-stream, using the :READ-POINTER stream operation. It may return NIL meaning that the position cannot be determined. In fact, it will always return NIL for a stream open in character mode and not at the beginning of the file.

With two arguments, sets the position using the :SET-POINTER stream operation, if possible, and returns T if the setting was possible and NIL if not. You can specify :START to position to the beginning of the file, or :END to position to the end.

DIRECTORY pathname

Returns a list of pathnames (truenames) of the files in the directory specified by pathname.

LOAD-VERBOSE Variable

The default value for the verbose argument to LOAD.

LOAD pathname &key verbose print (if-does-not-exist T) set-default-pathname package

Loads the specified file. The old calling sequence with positional arguments is still accepted. The new calling sequence differs mainly by having keyword instead of positional arguments; the argument print and the feature it controls is new. The package argument is not standard Common LISP, but everybody uses it. The set-default-package argument was in an earlier version of the Common LISP standard but has been removed from the latest standard.

If verbose is non-NIL (it defaults to the value of *LOAD-VERBOSE*), then a message may be printed saying which file is being loaded and into which package.

If set-default-pathname is non-NIL, the pathname defaults are set to the name of the file loaded. The default for set-default-pathname is T.

If print is non-NIL, the value of each expression evaluated from the file is printed on STANDARD-OUTPUT.

Chapter 12

Error Signaling and Checking

Note that Common LISP does not define any way of handing errors, so there has been no change in the facilities for doing so.

CLI:ERROR format-string &rest args

> Signals an uncorrectable error whose error message is printed by passing format-string and args to CLI:FORMAT. (CLI:FORMAT is used because only someone trying to write transportable Common LISP code would want to use this calling sequence.)

CERROR continue-format-string error-format-string &rest args

> When the first argument to CERROR is a string, it means you are using this Common LISP calling sequence. Then the error prints its error message by passing error-format-string and args to CLI:FORMAT, and it has one proceed type, which documents itself by passing continue-format-string and args to CLI:FORMAT.
>
> The old calling sequence for CERROR is still accepted, and even preferred.

WARN format-string &rest args

> Prints a warning on ERROR-OUTPUT by passing the args to FORMAT, starting on a fresh line, and then returns.
>
> If *BREAK-ON-WARNINGS* is non-NIL, however, WARN signals an error sort of like CERROR. You can proceed from the error, in which case WARN simply returns.

BREAK-ON-WARNINGS Variable

> If non-NIL, WARN signals an error rather than just printing a message.

BREAK format-string &rest args

BREAK passes format-string and args to FORMAT to print a message, then enters a read-eval-print loop reading from TERMINAL-IO. When you type the Resume character, BREAK returns NIL.

This is an incompatible change in the function BREAK. It used to accept an unevaluated string or symbol as its first argument and print it literally. To allow old code to continue to work, BREAK currently does not actually evaluate its first argument. However, if you call BREAK with a symbol as the first argument, the compiler will print a warning urging you to change the symbol to a string.

CHECK-TYPE place typespec [string] Macro

Signals an error if the value of place does not fit the type typespec. This is tested by (TYPEP place 'typespec). Note how typespec is not evaluated; its value is used at compile time.

string is used in the error message to say what the object was supposed to be; it should start with an indefinite article, as in "a number". Usually you do not need to specify string as a default as it will be computed from the typespec. The error message will also include both place itself and place's value on this occasion (the value that failed the type test).

The error is of condition SYS:WRONG-TYPE-ARGUMENT and you can proceed, specifying a new value that is stored into place with a SETF. The new value is then tested, and so on until a value passes the test. Then CHECK-TYPE returns.

ASSERT test-form [(places...) [string args...]] Macro

Signals an error if test-form evaluates to NIL. The rest of the ASSERT is relevant only if the error happens.

First of all, the places are forms that can be SETF'd, and which are used (presumably) in test-form. The reason that the places are specified again in the ASSERT is so that the expanded code can arrange for the user to be able to specify a new value to be stored into any one of them when he proceeds from the error. When the error is signaled, one proceed-type is provided for each place that can be set.

If the user does proceed with a new value in that fashion, the test-form is evaluated again, and the error repeats until the test-form comes out non-NIL.

The string and args are used to print the error message. If they are omitted, a message "Failed assertion" is printed.

The args are evaluated only when an error is signaled, and are evaluated again each time an error is signaled. SETF'ing the places may also involve evaluation, which will happen each time the user proceeds and sets one.

Example:

```
(ASSERT (NEQ (CAR A) (CAR B)) ((CAR A) (CAR B))
        "A and B are EQ: ~S and ~S" (CAR A) (CAR B))
```

The places here are (CAR A) and (CAR B). The args happen to be the same two forms; the current values of the places will often be useful in the error message.

ETYPECASE keyform clauses...

Like TYPECASE except that an uncorrectable error is signaled if every clause fails. A clause looks like (typespec forms...). The clause succeeds if keyform's value matches typespec with TYPEP; then the forms are evaluated and the values of the last form are returned by the ETYPECASE. The first successful clause is the one that is used.

TYPECASE allows OTHERWISE clauses also, but ETYPECASE does not allow them.

CTYPECASE keyform clauses...

Like ETYPECASE except that keyform must be SETF'able and the error signaled is a correctable one. The user can proceed with a new value; this value is stored into keyform with SETF and the clauses are tested again.

ECASE keyform clauses...

Like CASE (or, SELECTQ) except that an uncorrectable error is signaled if every clause fails. Each clause starts with a value or a list of values, followed by forms. The clauses are tested by matching the value of keyform against the value or values specified in the clause. If there is a match, the clause succeeds; its forms are evaluated and the values of the last form are returned from the ECASE. If all the clauses have been tested and failed, the error is signaled.

CCASE keyform clauses...

Like ECASE except that keyform must be SETF'able and the error signaled is a correctable one. The user can proceed with a new value; this value is stored into keyform with SETF and the clauses are tested again.

Chapter 13

The Compiler

COMPILE function-spec &optional definition

If function-spec is non-NIL, its definition is compiled. If definition is non-NIL, it should be a lambda-expression; it is compiled and the result becomes the definition of function-spec. The value is function-spec.

If function-spec is NIL, definition is compiled and the result is returned without storing it anywhere.

COMPILE-FILE input-file &key output-file set-default-pathname

Compiles the file specified by input-file, a pathname.

If output-file is specified, it is a pathname used for the compiled file. Otherwise, the output file name is computed from the input file name.

set-default-pathname, if non-NIL, means that the defaults should be set to the input file's name. set-default-pathname defaults to T. The set-default-package argument was in an earlier version of the Common LISP standard but has been removed from the latest standard.

Chapter 14

Miscellaneous

DOCUMENTATION name doc-type

Returns the documentation of name in the role doc-type. If doc-type is FUNCTION, then name may be any function-spec, and the documentation string of its function definition is returned. Otherwise, name must be a symbol, and doc-type may be anything. However, only these values of doc-type are standardly used:

VARIABLE	Documentation of name as a special variable. Put on by doc strings in DEFVAR, DEFCONST, DEFCONSTANT, DEFPARAMETER.
TYPE	Documentation of name as a type for TYPEP. Put on by doc strings in DEFTYPE forms.
STRUCTURE	Documentation of name as a DEFSTRUCT type. Put on by doc strings in DEFSTRUCTs.
SETF	Documentation on what it means to SETF a form that starts with name. Put on by a doc string in a DEFSETF of name.
DEFFLAVOR	Documentation of the flavor named name. Put on by the :DOCUMENTATION option in DEFFLAVOR. It would be more consistent to use FLAVOR as the doc-type for this, but it is desirable not to have to put FLAVOR in GLOBAL.

Documentation strings for any doc-type can also be added to symbols by means of (SETF (DOCUMENTATION name doc-type) string).

TIME form

Evaluates form and prints the length of time that the evaluation took. The values of form are returned.

TIME with no argument still returns a fixnum time value counting in 60'ths of a second.

DRIBBLE &optional pathname

This function with an argument replaces DRIBBLE-START. With no argument, it replaces DRIBBLE-END.

LISP-IMPLEMENTATION-TYPE

Returns a string saying what kind of LISP implementation you are using. On the Lambda it is always "ZetaLISP".

LISP-IMPLEMENTATION-VERSION

Returns a string saying the version numbers of the LISP implementation. It looks something like "System 98.3, CADR 3.0, ZMAIL 52.2".

MACHINE-TYPE

Returns a string describing the kind of hardware in use. It is "LAMBDA" or "CADR".

MACHINE-VERSION

Returns a string describing the kind of hardware and microcode version. It starts with the MACHINE-TYPE. Example: "LAMBDA Microcode 286".

MACHINE-INSTANCE

Returns a string giving the name of this machine. Do not be confused; the value is a string, not an instance. Example: "LAMBDA-118".

SOFTWARE-TYPE

Returns a string describing the type of operating system software that LISP is working with. On the LISP Machine, it is always "Zetalisp", since the LISP Machine LISP software is the operating system.

SOFTWARE-VERSION

Returns a string describing the version numbers of the operating system software in use. This is the same as LISP-IMPLEMENTATION-VERSION on the LISP

Machine since the same software is being described.

SHORT-SITE-NAME

Returns a string giving briefly the name of the site you are at. A site is an institution that has a group of LISP Machines. The string you get is the value of the :SHORT-SITE-NAME site option as given in SYS: SITE; SITE LISP. Example: "MIT AI Lab".

LONG-SITE-NAME

Returns a string giving a verbose name for the site you are at. This string is specified by the site option :LONG-SITE-NAME. Example: "Massachusetts Institute of Technology, Artificial Intelligence Laboratory"

FEATURES Variable

The value is a list of symbols describing the features available in the LISP system you using. The #+ and #- read-time conditionals check for the presence or absence of keywords on this list. Comparison against elements of *FEATURES* is done using STRING-EQUAL so that package is irrelevant.

Chapter 15

Time Functions

GET-INTERNAL-RUN-TIME
GET-INTERNAL-REAL-TIME

Both of these functions are equivalent to TIME with no argument. They return the current time in 60'ths of a second, counting from an arbitrary instant. This time value wraps around every so often and therefore values should be compared using the already existing functions TIME-DIFFERENCE and TIME-INCREMENT.

INTERNAL-TIME-UNITS-PER-SECOND   Variable

The number of time units in a second, for the values returned by GET-INTERNAL-RUN-TIME. The value is 60. The value may be different in other Common LISP implementations.

SLEEP seconds

Equivalent to (PROCESS-SLEEP (* 60. seconds))

GET-UNIVERSAL-TIME, DECODE-UNIVERSAL-TIME, ENCODE-UNIVERSAL-TIME

These functions, which already existed in the TIME package, are now in GLOBAL.

DECODE-UNIVERSAL-TIME returns one additional value, the time zone your machine is in. Also, the year it returns is now a number greater than 1900 rather than a number less than 100.

GET-DECODED-TIME

Equivalent to (DECODE-UNIVERSAL-TIME (GET-UNIVERSAL-TIME)); however, it can return NIL if the system does not know the time.

One additional change is that a year number less than 100 is now extended to a year within 50 years of the present, so that 27 stands for 2027 rather than 1927.

are now in GLOBAL.

DECODE-UNIVERSAL-TIME returns one additional value, the time zone your machine is in. Also, the year it returns is now a number greater than 1900 rather than a number less than 100.]

GET-DECODED-TIME

> Equivalent to (DECODE-UNIVERSAL-TIME (GET-UNIVERSAL-TIME)); however, it can return NIL if the system does not know the time.
>
> One additional change is that a year number less than 100 is now extended to a year within 50 years of the present, so that 27 stands for 2027 rather than 1927.

LMI Beta Release 2.0 Package

April 1985

24-0100335-0001

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Information in the enclosed manuals is beta-release. It may be changed to reflect improvements to the software.

Introduction

This is your Beta Release 2.0 documentation. It is organized to reflect basic information about the software, general conditions of its use, and specific new features.

Here is what you will find in this packet:

- *Release 2.0 Overview*—**Read this first** to see the advantages and new capabilities of Release 2.0.
- *Installation*—How to install your new system.
- *Release 2.0 Notes*—System release notes, covering all features of the system in detail.
- *Release 2.0 Conversion*—Converting older ZetaLISP code to Release 2.0.
- *Interprocessor Communication: The Extended STREAMS Interface*—Using STREAMS to communicate among processors in your Lambda.
- *Lambda Tape Software*—Full information on tape functions.
- *Editing the Lambda's Site Information*—Customizing your site for your software upgrade or new Lambda.

Other information relevant to Release 2.0 is included in separate packets with this release. You will receive one or more of the following, depending on your site configuration:

- *ZMAIL Overview*—A user-friendly guide to the LISP ZMAIL mail system.
- *ZMAIL Manual*—A reference guide to ZMAIL.
- *Lambda Kermit User's Guide*—Using the file transfer program Kermit on your LMI Lambda.
- *Kermit User Guide, Fifth Edition*—Complete information on Kermit, from its developers at Columbia.
- *Multibus Driver Guide*—Designing device drivers for peripherals.
- *The Microcompiler*—Information on compiling LISP directly to microcode.

Release 2 Overview

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Introduction

Release 2 offers subtantial improvements over Release 1 in almost every aspect; it is faster, has more features, and is less susceptible to bugs. The most important advance of Release 2 on the Lambda is the adoption of 25-bit pointers; now the address space is twice as large as before, so that user programs now have up to 16000K words more space to use. Execution speed has been increased. Release 2 contains improvements in the user interface (notably in the rubout handler and Zmacs, the editor). In addition, Common LISP is supported in full.

Common LISP

The most noticeable software change for Release 2 is the addition of support for Common LISP. Although Common LISP and ZetaLISP-Plus are very close, there are differences; you have a choice of what incompatible Common LISP functions you would like to use. Here are the general differences:

- Constructs (functions and variables) tend to have more regular names. (The names of many ZetaLISP-Plus system variables are the same as those in Maclisp and Franz LISP.) For example, in Common LISP, **base** becomes ***print-base***.
- Some functions and situations are more well-defined.
- Some Common LISP features are oriented towards conventional architectures. (On the LISP Machine they are legal, but superfluous.)

Common LISP and ZetaLISP-Plus exist side by side in LMI software. For incompatible functions with the same name in both dialects, and when the input syntax is slightly different, the machine interprets the dialect that the form is in according to the time when the form is entered. (This is implemented with readtables.) When the machine is running code, instead of declaring a "mode" for Common LISP, one tells the editor or the LISP Listener what dialect is desired.

- Programmers can use new, upward-compatible Common LISP functions from ZetaLISP-Plus without special arrangement.
- Incompatible Common LISP functions can be used from ZetaLISP-Plus programs by explicitly referencing the Common LISP incompatible (**CLI**) package.
- Incompatible ZetaLISP-Plus functions can be used from Common LISP, if needed, by explicitly referencing the **GLOBAL** package.
- Both the old and the new, Common LISP names of variables can be used; for ZetaLISP-Plus, no "preferred" name is enforced.

This set of features make it possible to port an existing ZetaLISP-Plus program to Common LISP function by function, file by file.

What does Common LISP bring to ZetaLISP-Plus ?

Many aspects of Common LISP can be adopted by programmers without using incompatible functions. The following is a list of what is immediately available for programmers in ZetaLISP-Plus:

- Better names for certain functions and variables.
- Long awaited printer/reader features. A wider range of structures can be read and printed by the LISP function **(READ)**.
- Compatibility with other LISP implementations (the next NIL, Spice LISP, Standard LISP).
- Lexical scoping. Many locally special declarations are no longer needed. Local function and macro definitions are now possible.
- Upward compatible changes to many existing ZetaLISP-Plus functions. (One of the more important is the tighter definition of **SETF**.)
- A more comprehensive type system.
- Character objects, rational and complex numbers.
- Generic sequence functions that accept both vectors (one-dimensional arrays) and lists. Many of these functions express common programming idioms that are usually not defined as part of LISP.

Other Improvements

Besides the adoption of Common LISP, Release 2 features some new functions, many bug fixes, and improved implementations of important system facilities.

- **DEFSUBST** is fixed; certain bugs associated with the previous implementation have gone away.
- Translation for logical hosts can now be automatically updated.
- The rubout handler has been improved. For example, one can edit the end of one's input, and have it be re-input, if syntactically possible, with a single keystroke.
- The default font **CPTFONT** is more readable. The lower-case letters have been enlarged slightly, so LISP code in lower case should be more readable.
- More fonts have a wider range of characters. Earlier on, only a few fixed-width fonts actually had glyphs for every printing character in the LISP Machine character set. However, many of the text-oriented fonts (the **HL**/Helvetica series, the **TR**/Times Roman series) have had more characters added to them. For example, backquote (the character ‘ should now be available in all text fonts.
- Notifications are handled differently. Now, only the appropriate flavors of windows will print out notifications on on themselves. Most windows will handle notifications by telling you about them (via the documentation window at the bottom of the screen); there are commands to view notifications at your leisure with a ⟨TERMINAL⟩ command.

Editor and ZMail Improvements

- The most useful improvement to the editor (meaning Zmacs, which implements both **Zwei** and **ZMail**) is that "undoability" of editing is now allowed for all modifications. Before, only a few operations had methods for undo the changes; now, any change to text can be undone. The editing history is maintained on a per-section basis, so that you can now undo changes selectively in one unit of your buffer. Usually, that unit will be a LISP defining form (**defun**, **defmethod**, and so on) or a paragraph.

A common application for such generalized 'undoability' is program modification. You no longer need to save away changed sections of programs while modifying them, but can simply undo the changes that are local to a section.

- In **Zwei**, some commands have been renamed to more cleverly exploit command completion.
- **Zwei** also has commands for supporting Common LISP. (These commands the attribute list of the buffer you are editing, to allow to use either dialect on a per-buffer basis.) **Zwei** understands the slight differences between Common LISP and ZetaLISP-Plus syntax.
- The 'modified' flag (*) in editor modelines has been moved to the left so that a modified buffer is more noticeable. Formerly, it was sometimes hard to discern a modified buffer if one was editing a file with a long name.
- **ZMail** performance has been improved; in addition, several bugs have been fixed. There are new profile variables for more customization, and a new Undigestify command (for reading mailing list digests) has been provided.

Using Release 2

Because the internals of Release 2 are so different from Release 1, all user files (including init files and ZMail init files) will need to be recompiled. The details are specified in the *Release 2 Conversion Notes*. The compiler, of course, will pick up the errors, but you will probably want to read the conversion advice to ease the transition. Note that many old constructs are still supported, even though newer ones are preferred.

Documentation

Because Release 2 has so many changes, there is a good amount of documentation to accompany it. Other documents describe some aspects of Release 2 in more detail; this is an overview of the major features of Release 2.

- *Common LISP Release Notes*: Describe the new features from Common LISP. Almost of all the document is now contained in the latest version of the *LISP Machine Manual*, but the *Notes* provide a convenient way to peruse the many features. These *Notes* are already provided in your manual set.
- *Release 2 Notes*: A comprehensive summary of Release 2 changes, with special attention to the package system and the **DEFSTRUCT** facility. All programming, user interface, editor, and site maintenance changes are documented here in detail.
- *Release Conversion Notes*: General advice for converting from Release 1 (Lambda System 1, CADR System 94) to this Release.
- *LISP Machine Manual, Sixth Edition*: The new edition, available as the LISP I and LISP II binders of the LMI documentation set. (Itis also available separately, in one volume, as an orange-covered paperback.) The manual corresponds to the current release.

- *Common LISP:* Written by Guy Steele and available from Digital Press, this is the definitive document of Common LISP. It is not necessary, however, to acquire this book to use Common LISP with LMI software.

In the document package you receive with this release is documentation corresponding to further features included in Release 2.

Installation Packet
LMI Beta Release 2.0–3/29/85

24-0100323-0002

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Table of Contents

Cautionary Preface

It is always a good idea to back up your files (LISP and UNIX) on the system before a software update.

Installing UNIX usr files will write over any files in the UNIX "/usr" directory that have the same name as files in the new usr file tape. This is because UNIX does not have file version numbers.

Be sure you back up any modified distribution files.

1. Updating an Existing System

1.1 The Root Image

Installing the UNIX root image will write over all UNIX files in "/" and "**/sdu**". Therefore, any valuable files here should be backed up before the update and restored after it. There are limitations on this, however, and a system operator at your site will have to be involved with this procedure.

For instance, the **/etc/rc** file has been changed to work with Beta Release 2.0. Because of this, the system will work improperly if you merely restore your previous **/etc/rc** file. A UNIX system operator at your site should merge your changes with the new release file in order to maintain system release integrity.

UNIX files in **/usr** will not be changed by the root change.

In a standard UNIX environment, at least the following files will be customized: **/etc/passwd**, **/eet/ttytype**, **/etc/ttys**, **/etc/myhostname**, and **/etc/hostbin**. These files contain user ID, terminal, and site file information. Even if nothing else is customized at your site, you should back up these customized files and restore them after installing the new root.

The following procedure backs up the **/etc/passwd** file; this procedure could be extended to other important, customized files.

Procedure:

Before installing the software update, mount a tape and type in UNIX

```
cd /etc
tar cvfb /dev/rmt0 20 passwd ttytype ttys myhostname host-
bin
```

After the installation, you can restore these files to replace the standard files distributed with the release by typing in UNIX

```
cd /etc
tar xvpf /dev/rmt0
```

Note: Under no circumstances should you back up the "**/dev**" directory; attempting to do so will inappropriately open the "devices" needed to run the system (tape, disk, memory, terminal, etc.), and in doing so may crash the system.

1.2 Installation

The following is a command synopsis of the installation procedure.

1. Back up customized files
2. Install the Beta Release 2.0 Root Image at the SDU:

   ```
   copy tape $disk
   ```

   This will take about 20 minutes.

   Note that there are different versions of the root for systems with UNIX and systems without. The root with UNIX is labeled "Beta Release 2.0 Root with UNIX." The non-UNIX version is labeled "Beta Release 2.0 non-UNIX Root."
3. Use the SDU program **config** to configure the system. Refer to Chapter 2, "The Config Program".
4. Install the LISP microcode and band using the SDU **load** program. Refer to Chapter 3, "The Load Program".

   Be sure not to load the microcode and band over any valuable microcode or band.

   Loading will take about 30 minutes.
5. Use the load program to select the newly installed microcode and band, then boot the system using the SDU "superboot" program.
6. Once LISP is booted, install the Beta Release 2.0 LISP Sources on the slot 0 Lambda processor by typing in LISP

   ```
   (fs:restore-magtape ':query nil)
   ```

   This will take about one and one-half hours, depending on the amount of memory in the system.
7. If the system has UNIX, boot UNIX multiuser and install the Beta Release 2.0 UNIX "usr" files tape by typing in multiuser UNIX

   ```
   cd /
   tar xvpf /dev/rmt0
   ```

   This will take about 20 minutes.
8. From here, you should update the site files. Refer to Chapter 5, "Site File Installation Notes".

Additional Information

If you prefer, you can install the Beta Release 2.0 microcode, band and sources on the machine before you install the new root image. In this case, install the microcode and band in LISP instead of at the SDU, using the LISP command **(fs:restore-magtape)**. After that, while still booted on Release 1.2, install the Beta Release 2.0 LISP Sources in LISP with

```
(fs:restore-magtape ':query nil)
```

After this LISP update is completed, proceed with steps 1-3 as described above, and with step 8 if the system has UNIX.

2. Beta Release 2.0: The config Program

Config version 131 is the current version of config in Beta Release 2.0.

As with config 89, you can type in partition names in either upper or lower case, and they are forced to upper case.

Config 131 is compatible with the **load** program in assigning page and file partition names. Both programs now default to FILE and PAGE for the slot 0 Lambda processor, FIL1 and PAG1 for the slot 4 Lambda processor.

Config 131 allocates 1MB to UNIX by default. This fixes the config 89 bug that allocated too little memory to UNIX.

To configure your system, type at the SDU

config

The first time config is run after a new root image is installed in a machine that has UNIX, config will prompt for the UNIX boot console. Type

ttya

to use a Z29-type terminal as the boot console, or

sharetty

to use the high resolution monitor as the boot console. If **sharetty** is used as the UNIX console, you will have to halt the slot 0 LISP processor anytime you need to boot UNIX (single or multiuser).

After this, config will print the configuration and ask if you want to change anything. In most cases, the default configuration will be appropriate. When config prompts,

```
Do you want to change anything? (y/n)
```

respond by typing "**n**" followed by a RETURN. The specified configuration will be saved onto the disk. This does not need to be done every time you power up the system; you do not need to run config again unless you need to change one of these basic configuration options.

Additional Information

The config program, run from the SDU, locates boards, allocates memory, sets the system console device and several other configuration options.

This config version differs from the config version 89 of Release 1.2, both cosmetically and internally.

The first time config is run after a new root image is installed, config will say what each slot has become before it prompts for changes. In subsequent runs, config will print out the current configuration.

Example

The following is a transcript of a typical config run. *Slanted text* indicates user input.

```
>> config

using 64K in slot 10
config version 131

Slot 0 has lambda
Slot 4 has lambda (disabled)
Slot 8 has vcmem
Slot 9 has vcmem
slot 10 has two-meg
slot 11 has has 68000 (disabled)
slot 12 has two-meg
slot 13 has half-meg
slot 14 has half-meg
slot 15 has sdu

Total memory = 5120K bytes

Lambda V4.0 in slot 0
     vcmem in slot 8
               (pool room)
     has processor switches 013600000000
        parity-enable byte 00 (all off)
        tram file /disk/lambda/c.tram-n-n, boot speed 1-1
        microcode band <default>, load <default>, page PAGE,
        file FILE
        scan-line-size 32.
        5030K bytes memory
```

```
System parameters:
     user-defined shared area is 20K
     sdu code area is 64K
     reserved multibus space is 8K
               (from 0xEE000 to 0xEFFFF)
     system-configuration shared area is 6K
Do you want to change anything? (y/n) n

Writing config file "/disk/lambda/shr-config.1"
Initializing SDU ...
```

If you answer "y" when asked if you want to change anything, the following example might occur. The prompts used by config are **"cmd:"** at top level, **"lambda cmd:"** in "lambda" mode, and **"unix cmd:"** in "unix" mode.

```
Do you want to change anything? (y/n) y
Type "?" for instructions.

cmd: ?
The usual procedure is to disable any boards that you don't
want to use, then change any of the LAMBDA or unix options,
and then write the file.  When a number is asked for,
<return> always defaults or aborts. Control-C aborts and
exits without changing the file.

Commands are:
     dflmem   -   reset memory allocation to default
     disable  -   turn off a board
     enable   -   turn on a board
     lambda   -   change lambda options
     loc      -   edit console location strings
     mem      -   change memory allocation
     print    -   print current config info
     reset    -   reset config file to defaults
     system   -   changes system-wide and sdu options
     unix     -   change unix options
     write    -   write new config file and exit
     x        -   exit
     ?        -   instructions

cmd: enable
Enable board in which slot? 11
cmd: unix
unix cmd: console
The console devices are: ttya sharetty
Enter console device: ttya
```

```
unix cmd: x
cmd: loc
          SDU port A
           pool room
Type new string, or <return> to leave unchanged.
pool room terminal 3
cmd: write

Writing config file "/disk/lambda/shr-config.1"
Initializing SDU ...
```

Typing "?" at a sub-level command prompt ("lambda cmd:" or "unix cmd:", for example) will give you help on the commands for that level.

3. Beta Release 2.0: The load Program

The load program can load microcode and band tapes, copy from one disk partition to another, edit disk label comments, and write out microcode and bands from the disk to tape. It reads the type of disk out of the CMOS RAM. It can deal with 1/2" tape or 1/4" tape, 474MB disks and 169MB disks.

3.1 How to Invoke the load Program

Invoke the load program from the SDU by typing

```
load
```

It will print a message that it is "load version 107", and will return a "disk loader>" prompt. To get a list of all the possible commands, type

```
help
```

at the prompt.

The first time you try to access the disk in the load program, it will look in the CMOS RAM to find out what type of drive you are using. Then this information will be printed on your screen. For example, if the first thing you do is print the disk label on a Lambda/PLUS, it will display

```
842 cylinders, 20 heads, 25 sectors per track
disk drive is Fujitsu Eagle
```

3.2 Functions of the load Program

The following is a description of the functions of the program.

printlabel
This prints the disk label. Note that this disk label differs from the old one. "FILE" is the name of the first file partition and "FIL1" is the name of the second (formerly "FIL1" and "FIL2"). "PAGE" is the name of the first page partition and "PAG1" is the name of the second (formerly "PAG1" and "PAG2").

initlabel This initializes the disk label. Note that it writes a different disk label than the old load program did. When you type **initlabel** at the `disk loader>` prompt, you will be asked to specify whether you have a 2X2 configuration. Appropriately, a 2X2 disk label is written if you respond **y**, and a single Lambda label is written if you respond **n**.

setmload This sets the current microcode to whatever you specify. You may type the partition name in upper or lower case. Be sure to type `lmc`*n* instead of just *n*, or the machine will not boot.

Example
(User responses are given in *slanted text*, for clarity:)

disk loader> *setmload lmc2*
Setting current microcode to LMC2

setband This sets the current band to whatever you specify. You may type the partition name in upper or lower case. Be sure to type **lod***n* instead of just *n*, or the machine will not boot. **setband** does not check to see whether you have selected the microcode that corresponds to the band you specify; if you are unsure of which microcode to select for the band you want, use the **prefmic** command and then the **setmload** command.

prefmic When you specify a band partition, this tells you which microcode it goes with. This will make sure you can change bands and still have a compatible microcode. Previously, this could be done only in LISP.

Example

disk loader> *prefmic*
select a partition *lod1*
microcode version 152
disk loader>

ttod This command (short for "tape to disk") is the equivalent of the old **load** command. Only the name has changed. Known, minor bug: when it gets to the end of the tape, it will print the message **Read error in header, probably end of tape.**

dtot This command ("disk to tape") is used to write out a partition to tape (for making a backup, for example). When prompted, you supply the tape drive type and partition name for which partition you want written onto the tape, and later specify whether you want to copy another partition to the tape. Previously, this could be done only by the LISP copy-disk-partition command.

Example

disk loader> *dtot*
What kind of tape drive do you have
(1 = 1/2", 2 = 1/4")? *1*
select a partition *lod1*
Copying xxx bytes
....

```
copy done
do you wish to copy another partition? n
disk loader>
```

dtod This ("disk to disk") can be used to copy one disk partition to another or to compare two disk partitions. When prompted, you specify whether to do a copy or a compare, and which partitions to use. . Previously, this could be done only by the LISP copy-disk-partition command.

Known, non-fatal bug: although the first time you specify a nonexistent partition name (e.g. "lmc9"), it will gracefully tell you that it couldn't find the partition, the second time it may break and give you a series of "unknown command" messages. In this case, you will have to exit the program by typing **CTRL-C**, then type **init**.

Example

```
disk loader> dtod
copy or compare (1 to copy, 2 to compare) 1
select a partition lod1
select a partition lod2
copying xxx bytes
......
copy done
disk loader> dtod
copy or compare (1 to copy, 2 to compare) 2
select a partition lod1
select a partition lod2
comparing "102.92 B" to "102.92 B"
.......
compare done
0 errors
disk loader> dtod
copy or compare (1 to copy, 2 to compare) 2
select a partition lod1
select a partition lod3
comparing "102.92 B" to "102.92 B"
................
..........compare error at location xx
...............
compare done
1 errors
disk loader>
```

change-comment
Use this to edit the disk label comment fields, to comment out a bad copy or to customize. You must specify the partition, and then type in the new comment

when prompted. Remember that the comment cannot exceed 16 characters or it will be truncated. Previously, this could be done only with the LISP edit-disk-label command.

Example
This example shows how to name LOD3 "Experimental Copy".

> **disk loader>** *change-comment*
> **select a partition:** *lod3*
> **change comment for partition LOD3 to:**
> *Experimental Copy*
> **disk loader>**

size When you specify a partition, this tells the actual measured size of the band in that partition, rather than the allocated length as written in the disk label.

Example

> **disk loader>** *size*
> **select a partition** *lod4*
> **partition LOD4 has physical size 30000**
> **measured size 18211**

tapetype Allows you to specify more than once whether you are using 1/2" tape or 1/4" tape. This is useful only if you are making a backup and want to make both 1/2" tape copies and 1/4" tape copies during the same load session.

exit This is the correct way to leave the load program when you are done. (Previously, you had to type **CTRL-C**, then type **init**, before doing anything else.) When you type **exit** at the "**disk loader>**" prompt, the SETUP and ATTN lights on the front panel will come on because the machine is automatically doing an "init." In the usual amount of time, the two red lights will go off and the green RUN light will appear.

At this time, you can run any program.

4. Beta Release 2.0 Boot Procedure

The Beta Release 2.0 boot software changes the boot procedure considerably from that used in Release 1.2.

To boot a configured machine, type on the Z29 terminal

superboot -a

This program will print its version number, clear the screen, and boot all processors in the system without further prompting. Since this is self-explanatory, the rest of the discussion here deals with **superboot** behavior without the **-a** option, and with returning to the boot menu after booting.

If you type at the SDU console the command

superboot

(*without* the "**-a**"), a version number will print, the screen will clear, and a boot menu will appear. One such menu will print on each processor's console, and the **boot** command on each refers to that processor only.

The high-resolution monitor can be used to return to the boot menu at any time by typing **ctrl-meta-ctrl-meta-**⟨LINE⟩. This will halt the LISP processor associated with that console, and will exit to the boot menu.

The specific behavior of this superboot differs according to the processor configuration and the system and also according to console options specified when using "config". Because of this, behavior will be treated here in specific modules, before any integrated documentation is attempted.

4.1 LISP Console Boot Menu

Figure 1 shows the menu as it appears on the LISP console. It includes processor and console information, commands, and a prompt.

Typing commands at the command prompt has the following results.

boot Prints "**Cold booting Lambda**" and boots the LISP processor associated with that console. This is equivalent to **ctrl-meta-ctrl-meta-**⟨RUB OUT⟩ on that LISP processor.

```
This processor:
     LMI LAMBDA V4.0 in slot 0
     (system console)
          console is vcmem in slot 8
          location:  pool room terminal 3

Commands are:
          boot                 cold-boot lambda
          warm                 warm-boot lambda
          why                  diagnose reason for halt
          unix                 connect to unix console
          reset-68000          reset 68000
          herald               print sign-on message
          INIT                 reset SDU and all processors

Command:
```

Figure 1. Typical LISP Console Boot Menu

warm Prints the console location and then tries to reboot that LISP processor, saying **"Warm-booting lambda: If this works, save your files and cold boot"**. This command is equivalent to **ctrl-meta-ctrl-meta-**⟨RETURN⟩ on that LISP processor.

why Fills the window to the bottom with debug information, and then wraps to the top of the screen and continues printing. The information takes less than a screenful to display, and so you do not have to fear that it will overwrite itself.

unix Connects to the UNIX console and boot menu on the high-resolution monitor if **sharetty** was set in **config** as the UNIX console. Otherwise, it says **"You can't connect to unix from here"**. If **sharetty** is used as the UNIX console, you should boot UNIX before booting LISP or else you will have to halt the slot 0 LISP processor using **ctrl-meta-ctrl-meta-**⟨LINE⟩ in order to boot UNIX.

reset-68000
Says nothing on the console, but kills and then reboots UNIX if UNIX has been booted during this session. If not, it is a no-op. This is to be used in case UNIX wedges in some way that previously would have required pressing ⟨RESET⟩ to recover. If you just want to bring down a working UNIX, it is better to exit UNIX "gracefully" using the usual UNIX commands at the 68000 CPU.

herald Redisplays the **"This processor"** message.

INIT (Note that you must type this in capital letters.) Says **"Initializing SDU ..."** and does so. This really does reset all processors, no matter who is doing what on them; it should, therefore, be treated with respect. All users should be prepared for the machine to be powered down before this command is used. **INIT** may create inconsistencies in the filesystems of the various processors if it is used during file operations.

? Redisplays the command list. (This is not on the menu.)

4.2 UNIX Console Boot Menu

Figure 2 shows the menu as it appears on the UNIX console. It includes processor, LISP processor, and console information, commands, and a prompt.

```
This processor:
     68000 in slot 11
          console is ttya

Other processors:
     LMI Lambda V4.0 in slot 0
     (system console)
          console is vcmem in slot 8
          location:  pool room terminal 3

Commands are:
          boot                      boot unix
          herald                    print sign-on message
          INIT                      reset sdu and ALL processors

Command:
```

Figure 2. Typical UNIX Console Boot Menu

Figure 2 shows a typical UNIX console boot menu. It is quite similar to the LISP menu. As with the LISP console boot menu, **INIT** resets all processors and **herald** prints the **"This processor"** information.

Typing **boot** at the UNIX menu prints the messages

```
Booting 68000 in slot 11 with /disk/unix.new
reading file '/disk/unix.new'
starting 68000
```

and then prints the singleuser UNIX herald and prompt.

If **ttya** was specified as the UNIX console during the config program, each LISP and UNIX processor will have its own boot console and menu and therefore booting is completely independent. If **ttya** is the UNIX console, **superboot -a** is the preferred boot method.

If **sharetty** was specified as the UNIX console during the config program, UNIX will share a boot console with the slot 0 LISP processor. When using this configuration, use **superboot** instead of **superboot -a**. Boot UNIX multiuser before booting LISP. If you used **superboot**, you would have to halt the slot 0 processor with **CTRL-META-CTRL-META-**⟨LINE⟩ in order to boot UNIX multiuser; this is not recommended, because it is very inefficient.

Since it is preferable NOT to halt LISP once it is booted, you should use **superboot** instead of **superboot -a** and should boot UNIX multiuser before booting LISP if **sharetty** is the UNIX console; if **ttya** is the UNIX console, **superboot -a** is the preferred boot method.

4.3 Boot Sequence on Lambda/PLUS with superboot -a

Here is the boot sequence on a Lambda/PLUS using **superboot -a** and **ttya** as the UNIX console.

Type at the SDU

superboot -a

When you get the singleuser UNIX prompt on the z29, type **CTRL-D**. At the end of this process, all processors will be booted completely.

Here is the boot sequence on a Lambda Plus using **superboot -a** and **sharetty** as the UNIX console. Type at the SDU

superboot -a

The machine will print some information. Type at the booted slot 0 Lambda console the following. (Your input is indicated by *slanted text*. This is an edited session; most of the machine response is indicated by ellipsis dots. Note the change of prompts, >> to #.)

```
>>    CTRL-META-CTRL-META-⟨LINE⟩
      Command: unix
      .
      .
      Command: boot
      .
      .
      CTRL-D
#
      .
```

.

CTRL-META-CTRL-META-`LINE`

.

Command: *boot*

The last command, **boot**, will reboot the halted slot 0 Lambda.

4.4 Boot Sequence on Lambda/PLUS—superboot with sharetty

The boot sequence on a Lambda Plus using **superboot** with **sharetty** as the UNIX console is:

```
>>   superboot          ;;;Type this at the SDU
     .
     .
     Command: unix     ;;;Type this at the slot 0 Lambda console
     .
     .
     Command: boot
     .
     .
     CTRL-D
#
     .
     .
     CTRL-META-CTRL-META-LINE
     .
     .
     Command: boot
```

At the end of this procedure, both LISP and UNIX will be booted.

A second LISP processor does not affect this sequence; since it is not sharing a console, it can simply be booted via **boot** at its boot menu command prompt at any time.

As always, since booting requires heavy disk activity, multiple processors boot more slowly than a single processor.

5. Site File Installation Notes

5.1 Introduction

There are a number of differences between site files in Release 1.2 and Beta Release 2.0. Some of the syntax has changed (particularly in **SITE.LISP**). A new file, **SYS.TRANSLATIONS**, has been added.

To simplify the process of updating and editing site files, we have included in this release a dedicated site file editor, invoked in LISP by

```
(sited)
```

Updating site files for Beta Release 2.0 is a straightforward procedure.

- First the old site information is read into the "sited" editor. The site information is checked for consistency and completeness, then is written into the Beta Release 2.0 site directory ("**QL.CUSTOMER-SITE;**").
- The new site files are read into a Beta Release 2.0 load band, and the band is saved.
- This band can then be copied to the other Lambdas on the network, so that the site information on all the systems is consistent.

The following procedure assumes that you have loaded the Beta Release 2.0 load and microcode bands onto the SYS HOST (i.e., the machine with the source and site files for Release 1.2). You will need to know the "name" of the SYS HOST and the directory where the Release 1.2 site files can be found (usually "**RELEASE-1.CUSTOMER-SITE;**"). You will also need to find a spare load band which can be used for saving the updated Beta Release 2.0 load band.

5.2 Read Release 1.2 Site Files:

If your SYS HOST is not currently running the Beta Release 2.0 software, proceed with steps 1-3:

1. Log in to the SYS HOST:

   ```
   (login 'lispm t)
   ```

2. Set the current load and microcode bands to the Beta Release 2.0 software. If this software has been loaded into "LOD3", for example, you should enter:

```
(set-current-band "LOD3")
```

(If you are prompted to change the microcode band as well, respond with "y".)

3. Cold boot your machine by pressing CTRL-META-CTRL-META-⟨RUB OUT⟩.

   At this point your SYS HOST should be running Beta Release 2.0.

4. Once your SYS HOST is running the Beta Release 2.0 software, log in:

```
(login 'lispm t)
```

5. Enter the following, substituting the name of your SYS HOST for "**george**":

```
(site:set-sys-host-for-sited "george")
```

   (The name you supply here will be used in **SYS.TRANSLATIONS as** the value for ":physical-host".)

6. Enter the following:

```
(fs:set-logical-pathname-host "sys"
:physical-host "lm"
:translations '(("site;" "ql.customer-site;")
                ("chaos;" "ql.customer-site;")))
```

   The logical pathnames

   SYS:SITE;SITE and
   SYS:CHAOS;HOSTS TEXT

   will be translated into physical pathnames

   LM:QL.CUSTOMER-SITE;SITE.LISP and
   LM:QL.CUSTOMER-SITE;HOSTS.TEXT

   respectively.

7. If the Beta Release 2.0 source files have not been loaded, you will need to create a directory for the new site files. (If the source files already exist, this command will have no effect.) Type the command

```
(fs:create-directory "lm:ql.customer-site;")
```

8. Start the sited editor:

   (sited)

9. Read the Release 1.2 site information into the editor:

   SITED command> *readfiles*

   You will be prompted for the name of the site file directory:

   Directory: *lm:release-1.customer-site;*

   Respond with "y" when asked whether you wish to proceed. If you have renamed the site directory, substitute the name you supplied in place of "**customer-site;**".

5.3 Verify, Save New Site Information

1. Use the site editor to verify the Release 1.2 site information.

   Follow this procedure. (In case of doubt, your input is given in *slanted text.*)

   SITED command> *checkinfo*

   It is quite likely that the site editor will discover omissions in your Release 1.2 site files. For example:

   ```
   The PRETTY-NAME property for OURSITE-
   LAMBDA-C doesn't have any value
   specified for it.

   Should it be set to the default value
   of "Oursite Lambda C" ? (y or n)
   ```

   You should always respond with "y" at this stage. If you wish to use non-default values, you may specify them later. (See the site editor documentation for further details.)

   After the site information has been determined to be complete and correct, the following will be displayed:

```
No errors found in the information.
To generate new site files, use the
WRITEFILES command.
```

2. Copy the site file information into the Beta Release 2.0 site directory by entering:

   SITED command> *writefiles*

   You will be prompted for the name of the directory where you wish to write the new site information:

   Directory: *sys:site;*

   The site editor will generate the four site files (**SYS.TRANSLATIONS**, **SITE.LISP**, **LMLOCS.LISP**, and **HOSTS.TEXT**) and write them into the specified directory.

3. Quit from the site editor:

   SITED command> *quit*

5.4 Edit LMLOCS File

1. Start ZMACS by pressing ⟨SYSTEM⟩ **E**.
2. Read the **LMLOCS.LISP** file into an editor buffer. (Your input is given in *slanted text.*)

   CTRL-X CTRL-F
   Find file: *sys:site;lmlocs*

3. Use ZMACS delete commands to remove "**; Patch-file T**" from the first line and "**(REMPROP 'MACHINE-LOCATION-ALIST :SOURCE-FILE-NAME)**" from the eighth line of the file.
4. Save the file with **CTRL-X CTRL-S**.
5. Return to the Lisp listener by pressing ⟨SYSTEM⟩ **L**.

5.5 Compile Site Files; Save in Load Band

1. Compile the new site files:

   ```
   (make-system 'site :compile :noload
   :no-reload-system-declaration)
   ```

2. Cold boot the machine by pressing **ctrl-meta-ctrl-meta-**⟨RUB OUT⟩.

3. Log in.

```
(login 'lispm t)
```

4. Enter the following:

```
(fs:set-logical-pathname-host "sys"
    :physical-host "lm"
    :translations '(("site;" "ql.customer-site;")))
```

This tells **(update-site-configuration-info)**, below, where to find the site files.

5. Bring site information into the current Lisp environment:

```
(update-site-configuration-info)
```

6. Save the current world into a spare band. If you do not mind clobbering "LOD4", for example, you enter:

```
(disk-save "LOD4")
```

You will be prompted for comments to be used by the **print-herald** and **print-disk-label** commands. The current Lisp world will be saved in the specified band, then the machine will be rebooted on the newly saved band.

7. Set the current band to this band by entering, for example:

```
(set-current-band "LOD4")
```

5.6 Copy New Band to Other Machines on the Chaosnet

1. Log onto the machine that you want to receive the newly saved band:

```
(login 'lispm t)
```

2. Locate a spare band on the machine:

```
(print-disk-label)
```

3. Receive the new band over the Chaosnet. (In this example the SYS HOST is "**george**", the new band was originally saved into "LOD4", and the local band into which the new band will be copied is "LOD2"):

```
(si:receive-band "george" "LOD4" "LOD2")
```

4. Repeat this process for all machines at your site.

6. Beta Release 2.0 UNIX

Once UNIX comes up singleuser via the superboot UNIX **boot** command, it can be brought up multiuser using **CTRL-D** as usual. The main difference is that if **sharetty** was specified in config as the UNIX console, this work will be done on the high-resolution monitor instead of on the SDU Port A terminal.

When you get the singleuser prompt (#), type **CTRL-D** to bring up multiuser UNIX. The UNIX file system consistency check **(/etc/fsck)** program will run, automatically correcting inconsistencies. This differs from the previous release; in Release 1.2, the user had to answer each consistency question when prompted.

If anything is inconsistent in the root filesystem **(/dev/dk0a)**, UNIX corrects it, halts, and reboots automatically. This removes the problems in Release 1.2 when an inconsistent UNIX root caused the user to reset the entire system (including LISP).

If UNIX reboots itself, type **CTRL-D** at the prompt again to bring up multiuser UNIX. **/etc/fsck** will run again, but this time the root will be consistent and the program will continue.

A login prompt will appear on every enabled UNIX terminal when the initialization sequence begun by typing **CTRL-D** is complete.

7. Beta Release 2.0 UNIX Usr File Installation

In this distribution, files are written onto the tape with relative pathnames (e.g. **../usr/bin/mail**) to simplify installation.

Remember that UNIX does not have file version numbers. The files on the new distribution tape will overwrite the file on your disk with the same name; be sure to back up any files you have customized.

NOTE: Be sure UNIX is booted multiuser before installing usr files!

To install the Beta Release 2.0 UNIX usr files, mount the distribution tape and type in multiuser UNIX

```
cd /
tar xvpf /dev/rmt0
```

This will extract the 1153 files from the tape in about 20 minutes.

8. Improvements to the UNIX System

The following improvements have been made to the system:

- The spell program now works.
- Various options to nroff (e.g. the **-ms** macro option) are implemented.
- The **man -k** keyword option works.
- The help facility "?" in mail is implemented.

Appendix A. Beta Release 2.0 UNIX Files Size

Below are the number and size information of all files in the UNIX root, sdu, and usr file systems. This can be used to verify system integrity as a cross-check using the **/etc/fsck** file system consistency check program and the **df** disk space status program.

Beta Release 2.0 UNIX system with UNIX usr files installed:

```
/etc/fsck:
/dev/dk0a: 303  files 5303 blocks 1360 free
/dev/dk0e: 1153 files 8251 blocks 30147 free
/dev/dk0g: 57   files 905 blocks 53 free

df:
/dev/dk0a: 80% full
/dev/dk0e: 21% full
/dev/dk0g: 94% full
```

Appendix B. Ascertaining Software Version

You can easily determine which LISP microcode and band you are using by printing the LISP disk label. This can be done either using the **printlabel** command in the load program from the SDU or via the **(print-disk-label)** function in LISP.

To determine which version of the root is installed, you can type at the SDU the command

```
cat /uroot/version
```

This reads a file (the "**version**" file) that has existed and been updated appropriately since Release 1.0. The number of the release is on the first line of the file.

To read this version file in UNIX instead of at the SDU, type

```
cat /version
```

For the Beta Release 2.0 Root with UNIX, the version file says

```
Beta Release 2.0 -- 3/4/85

superboot linked to newboot 111
config 131
load 107
unix 3.275
```

For the Beta Release 2.0 Non-UNIX Root, the version file says

```
Beta Release 2.0 -- 3/4/85

superboot linked to newboot 111
config 131
load 107
```

If you are dealing with a system that contains UNIX, the files **/root.distr** and **/usr/usr.distr** are "cat"-able files containing a listing of the names, creation dates, permissions and owners of every file on the root and the usr file systems, respectively. These files were made at the time of the release with the **ls -lR** command in UNIX, and are on the system for baseline referencing.

Appendix C. Disk Labels for the Lambda Family

These are current as of Release 1.2 and hold for 2.0.

For a single Lambda or Lambda/PLUS:

Name	*Start*	*Length*
LMC1	25	500
LMC2	525	500
LMC3	1025	500
LMC4	1525	500
PAGE	2025	120000
FILE	122025	100000
LOD1	222025	35000
LOD2	257025	35000
LOD3	292025	35000
LOD4	327025	35000
METR	362025	8000

For a Lambda/2x2 or Lambda/2x2/PLUS:

Name	*Start*	*Length*
LMC1	25	500
LMC2	525	500
LMC3	1025	500
LMC4	1525	500
PAGE	2025	80000
PAG1	82025	80000
FILE	162025	75000
FIL1	237025	5000
LOD1	242025	30000
LOD2	272025	30000
LOD3	302025	30000
LOD4	337025	30000
METR	362025	8000

Release 2 Notes

24-0100330-0001

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

This document corresponds to Release 2.0. It supersedes the *System 98 Release Notes.*

LMI Lambda is a trademark of Lisp Machine, Incorporated.

Symbolics is a trademark of Symbolics, Inc. UNIX is a trademark of AT&T Bell Laboratories. VMS is a trademark of Digital Equipment Corporation.

Principal writers of this document were Richard M. Stallman and Richard Mlynarik of MIT, and Robert Krajewski of LMI. Formatted with BoTeX.

Table of Contents

Introduction

This manual describes these changes to the system in Release 2:

- COMMON LISP Support: New functions; usage on the LMI Lambda.
- Incompatible Changes: Some incompatible changes are due to COMMON LISP. Many of the others involve the increase in size of the LAMBDA's address space, or rationalizing the behavior of some constructs (like **eval-when**) that sometimes had hard-to-understand conseqeunces.
- Compatible LISP Programming Changes: This covers certain basic changes that are neccessary for and compatible with COMMON LISP. Improvements have been made to the rubout handler, the "system" system, the file system interface, networks, and miscellaneous low-level constructs in ZETALISP.
- Window System Changes
- User Interface Changes: This covers the way the user interacts with the machine (usually the window system) and ZMail, which is a new part of the LAMBDA software in Release 2.
- Editor Changes
- Defstruct Changes
- Package Changes
- Site File Changes: If you are responsible for maintaining site information, you should read this.

The changes for the **defstruct** facility and the package system are noteworthy enough to be placed in their own chapters. Many changes related to COMMON LISP are documented in the *Common Lisp Release Notes*; the new edition of the *Lisp Machine Manual* also documents these changes.

Although the new edition of the *Lisp Machine Manual* has been produced only recently, there are some omissions and changes which are included in this document.

In this document, fonts are used to highlight words and text in a number of ways:

arg — Here, the text mentions *arg*, which is an argument to the current definition. This is applicable for the documentation of functions, macros, operations, and special forms. *arg* could also stand for a value that was passed in some some pattern, as in **(:option** *arg***)**.

cons — Here, we are mentioning a LISP construct in text.

c-X — This same font is also used for mention names of characters, or sequences of keystrokes.

`FOO` — Usually, this kind of font is used in examples that are set off from the rest of the text. If this font is used inline, it is usually to emphasise the way something could be entered into the Lisp Machine.

The "bucky" shift keys have the names **Control**, **Meta**, **Super**, and **Hyper**. The character formed by typing the **X** key while holding down the **Control** and **Super** keys can be written as **Control-Super-X**. Sometimes, this is abbreivated to use only the first letters of the bucky bits: **c-s-X**. If the **Shift** key is also used, the character is represent as **Control-Super-Shift-X** or c-s-sh-X,

1. Common LISP Is Supported

Most of the differences between Common LISP and the traditional LISP machine dialect of LISP are compatible extensions. These extensions are available in all programs. They are described online in

- `SYS: DOC; COMMON LISP`
- `SYS: DOC; SYS98 DEFSTRUCT`
- `SYS: MAN; GENERIC TEXT`

The *Common Lisp Release Notes* covers the material included in those files. Many of these extensions are very useful, so all users should read these files even if they have no plans to write portable COMMON LISP code.

There are some incompatibilities between COMMON LISP and the traditional Lisp machine system, however. For the sake of existing programs, in most cases the system still works the traditional way.

In Release 2.0, the Lisp Machine will boot with ZETALISP being the default in the initial Lisp Listener. To use make a Lisp Listener use COMMON LISP reader syntax and functions by default, do

(setq *readtable* si:common-lisp-readtable)
or equivalently, **(common-lisp t)**

To use traditional ZETALISP syntax and functions, do

(setq *readtable* si:standard-readtable) or **(common-lisp nil)**

To make a file read in using COMMON LISP syntax, and use incompatible COMMON LISP functions when neccessary, put **Readtable: Common-Lisp** or **Readtable: CL** in the attribute list (the **-*-** line) of the file. To make a file always read using traditional syntax and functions, use **Readtable: Traditional** or just **Readtable: ZL**. The attribute **Syntax** is a synonym for **Readtable.**

The new editor command **m-X Set Readtable** is the recommended way to change the readtable attributes of a file (and the editor buffer). See section 8.11.1, page 82 for more information.

The readtable variable is bound at the top level of each process, so setting it applies only to the current process. Lisp Listeners check the variable before each form, so it works to set the variable while operating in the listener.

The current value of ***readtable*** is important for two reasons:

- First, there are some (small) differences between the syntax of COMMON LISP and ZETALISP.
- The readtable also has information about the symbols that name functions that are incompatible between the two dialects. The **CL** readtable will substitute **cli:listp** for **listp** at read time, for example. Because the support is implemented this way, there is no special variable that needs to tell the **Lambda** which way to behave at run time, in the case of an incompatibility.

A few things in COMMON LISP are not yet supported completely:

* transcendental functions of complex numbers
* "alternative" definitions (as macros) of nonstandard special forms
* **inline** and **notinline** declarations.

Some features of COMMON LISP are not yet supported exactly per the COMMON LISP spec:

* **&rest** arguments are not valid beyond the dynamic extent of a function; incorrect but legal values will result if such an argument is returned out of a function. To return a **&rest** argument as a true list, use the function **copy-list**.
* Zero-dimensional arrays may not be displaced or indirected.
* A number of COMMON LISP special forms are actually ZETALISP macros, a situation which could confuse some program-analyzing tools. **special-form-p** is COMMON LISP compatible, however; one should use that predicate and dispatch when needed before checking if a form is a macro-call.
* There are COMMON LISP names that are ZETALISP special forms and thus are not redefinable.
* If one makes a free reference to a variable in the interpreter, but does not declare it special, one gets thrown into the error handler. Currently, there is a proceed option that will subsequently allow this to happen without an error occuring. This proceed option will go away after Release 2.
* The ZETALISP implementation uses the tag values (**catch** and **throw** tags) of **t**, **nil**, and **0** for internal purposes. User programs should refrain from using these tags.
* In order to for files with font changes in them to be read correctly, reader macros must use the functions **si:xr-xrtyi** and **si:xr-xruntyi** instead of **read-char** and **unread-char**. Use of the special functions will be made unnececessary in a future release.

2. Incompatible Changes

This chapter describes various changes to ZETALISP that are likely to affect user programs. Most of the COMMON LISP changes have either been taken in through the **cli** package and readtable mechanism or have been documented elsewhere.

2.1 Tail Recursion

The variable **tail-recursion-flag** has no effect on the behavior of the system in Release 2. It is unlikely that it will ever be reinstated.

2.2 Returning Storage

Stricter conventions need to be observed when using **return-storage** and **return-array** than are alluded to in the *Lisp Machine Manual.* It implies that returning the storage and then clobbering the pointer (using **without-interrupts**) is adequate protection against improper reference. In fact, the PDL-buffer management in the microcode makes this not so. The only guaranteed technique is the

(return-storage (prog1 *pointer* **(setq** *pointer* **nil)))**

idiom, which the compiler optimizes into code that actually clobbers *pointer before* it calls *return-storage*. It is probably not a terribly good idea to call these functions from the interpreter. (Actually, with the new garbage collector on the way, it's probably not a good idea to call them at all.)

2.3 Clarification on Fill Pointers

Two clarifications need to be made about fill pointers in ZETALISP:

- Apparently, the documentation for **fill-pointer** has been wrong since Release 1. The documentation (including the most recent edition of the **Lisp Machine Manual**) states that **fill-pointer** returns **nil** if it argument (a vector) does not have a fill pointer. However, it has always signalled an error. (The condition flavor is **sys:array-has-no-leader** because, in ZETALISP, fill pointers are implemented as element zero of the array's leader.)

  Interestingly enough, the actual behavior of **fill-pointer** agrees with what COMMON LISP specifies. So, only the documentation will change.
- Fill pointers are only defined for vectors. Arrays that are not vectors, of course, may have leaders, but element zero of such a leader will *not* be considered a fill pointer by any system array function; nor will the function **array-has-fill-pointer-p** return **t** for such an array.

2.4 Changes Related to Common Lisp

The following changes have been made to ZETALISP for compatibility with COMMON LISP.

2.4.1 Decimal Radix Has Become the Default

Base **10.** is now the default. This is a COMMON LISP change. However, it is still possible to specify the radix for each file individually. To avoid any difficulties, place `Base: 8;` in the attribute list (the `-*-` line) of any file which is supposed to be in octal.

To get back the old behaviour, do

```
(setq *print-base* 8. *read-base* 8. *nopoint nil)
```

2.4.2 Ratio Reading and Printing

Ratios used to be always read and printed using decimal notation when using ZETALISP syntax. Thus, **#5r-10\12** (or **#5r-10/12** in COMMON LISP syntax) now represents "minus five sevenths."

2.4.3 Case-Sensitive and -Insensitive String Comparison Functions.

The function **equal** now considers the strings **"A"** and **"a"** to be distinct. Use **equalp** if you wish to ignore case in the comparison. This is a COMMON LISP change.

Because **equal** has changed, the ZETALISP functions **member**, **assoc**, **rassoc**, **remove**, **delete**, and **find-position-in-list-equal** are affected when strings are involved. (Note that the eponymous COMMON LISP functions use **eql** as the default comparison function.) Also affected are hash tables which use **equal** as the comparison function.

char-equal and **string-equal** always ignore case. To consider case in comparing characters or strings this way, use **char=** for characters and the new function **string=** for strings.

Here are some ways to compensate for the change in **equal** when you have been using strings as "keys" in lists (as sets), association lists, or in **equal**-based hash tables.

— In lists:

Release 1: `(member key *known-words*)`
Release 2: `(cli:member key *known-words* :test #'string-equal)`

— In association lists:

Release 1: `(assoc person nickname-alist)`
Release 2: `(cli:assoc person nickname-alist :test #'string-equal)`

— In hash tables

Release 1: `(defvar *things* (make-equal-hash-table :size 42))`
Release 2:
```
(defvar *things* (make-equal-hash-table
                  :size 42 :comparison-function #'string-equal))
```

samepnamep now considers case significant.

The functions of the **string-search** series now take an extra optional argument which says whether to consider case.

alphabetic-case-affects-string-comparison Variable

The old flag **alphabetic-case-affects-string-comparison** is now used only by the **%string-search** and **%string-equal** microcode primitives. These primitives now consider font significant as well as case when the flag is non-**nil**.

2.4.4 'COMPILE No Longer Needed in PROGN

Any **progn** encountered at top level by the compiler is now handled by treating each element as if it had been found at top level. Macros that used to expand into **(progn 'compile** *forms...***)** can now expand into just **(progn** *forms...***)**

2.4.5 Arrays Stored in Row-Major Order

Arrays used to be stored in column-major order. Now, they are stored in row-major order, which means that successive locations differ in the last subscript. The value of **sys:array-index-order** is now **t**; it was **nil** in Release 1. The change is *irreversible* and cannot be affected by changing the value of this variable. The change in storage layout does not affect user programs except when they do one of these four things:

1. Access screen arrays of windows using **aref**. Since the TV hardware has not been changed, the horizontal dimension is still the one that varies fastest in memory, which means it is now the second dimension rather than the first. The function **ar-2-reverse** (and its related versions for setting and getting locatives) was introduced in Release 1 so that code which was really using two-dimensional arrays in x/y terms would work no matter what the status of the index order was. If you used this function in such an application, you should have no problems.
2. Use multidimensional displaced arrays or arrays displaced to multidimensional arrays.
3. Deal with large multidimensional arrays and want to optimize paging behavior. For example, this piece of code will run with acceptable paging behavior in Release 1, but "pessimally" in Release 2 because it touches the elements of the array that are the farthest apart (in the first dimension) in the innermost loop.

   ```
   (defvar *space* (make-array '(100 100 100)))

   (defun make-space-centered ()
     (dotimes (iz 100)
       (dotimes (iy 100)
          (dotimes (ix 100)
            (setf (aref *space* ix iy iz)
                  (make-space-vector (- ix) (- iy) (- iz)))))))
   ```

   Notice that this example is written starting with an outer loop concerned with the z-axis. Most programmers would probably use the opposite approach if they did not care about paging performance at all, since it is "natural" to nest loops according the order of the indices as they are written.
4. Store multidimensional arrays in QFASL files. A QFASL file records the elements of an array in the order they appear in storage. Therefore, if an array is dumped in an earlier system and loaded into Release 2, it will appear to be transposed.

The functions **ar-2-reverse, make-pixel-array** and others are provided to make it easier for you to change your code so that it works in both Release 2 and older system versions. See these functions in the *Lisp Machine Manual.*

2.4.6 &KEY Arguments

It is no longer ever an error to omit a keyword argument defined with **&key**. **&Optional** now has no effect on the treatment of **&key** arguments. This change is for COMMON LISP.

2.4.7 Common Lisp Package Conventions

For more information about changes to the package system, see section 5.1, page 57. The changes documented here simply give a very cursory overview of the most obvious visible changes.

2.4.7.1 Keywords

The **user** package is now just like all other packages in requiring that colons be used in front of keyword symbols. For example, you can no longer write just **tyi** instead of **:tyi** if your program is in **user**.

All symbols in the **keyword** package – that is to say, symbols that you write with a colon, such as **:string-out** –are now automatically set up to evaluate to themselves. Thus, you can now write

```
(send stream :tyi)
```
instead of
```
(send stream ':tyi)
```

This Common LISP change ought not to invalidate any reasonable programs.

2.4.7.2 Referring to Packages

In COMMON LISP you must refer to an internal symbol of another package by using two colons (::). ZETALISP does not actually require you to use this construct, and you are free to access internal symbols with a plain colon in a package prefix. You can also suppress local package nicknames with **#:**. As of Release 2, the situation is:

foo:bar Refers to external symbol of package **foo** (but actually you can use it for any symbol in **foo**).

foo::bar Refers to an internal symbol of package *foo*, in strict Common LISP.

foo#:bar Refers to external (really, any) symbol in the package whose global name or nickname is **foo**, ignoring any local nickname *foo* for any other package.

#:bar Makes an uninterned symbol named **bar**.

Currently, internal symbols in other packages are indicated with ::. However, only the COMMON LISP readtable enforces the distinctions between external and internal symbols.

2.4.8 Local SPECIAL Declarations to Change in Meaning

For the sake of Common LISP, a **special** declaration within a function will have to be present in the construct (**let**, **prog**, etc.), which binds a variable in order to make the binding be special. Thus, for example,

```
(defun foo (a)
  (declare (special b))
  (let (b)
    ...))
```

will no longer make **b** special. Instead, you must write

```
(defun foo (a)
  (let (b)
    (declare (special b))
    ...))
```

where the local declaration appears just inside the construct that binds the variable in question.

A further unfortunate consequence of this is that **local-declare** cannot be used any more to make a binding special, as in

```
(defun foo (a)
  (local-declare ((special b))
    (let (b)
      ...)))
```

because this too would fail to put the declaration just inside the **let**.

To facilitate the changeover, this change has not actually been made. Local **special** declarations will still affect code just as they used to. However, any code that depends on this will get a warning reminding you to fix the code. The actual change will occur in a future system version.

Note that **local-declares** of **special** around an entire function, affecting arguments of the function, will continue to work. Also, if you are just examining or setting the variable, as in

```
(local-declare ((special a))
  ... (+ a 5) ...)
```

and not rebinding it, then your code will not be affected.

2.4.9 SELECTQ now uses EQL as its test function.

selectq formerly performed all its comparisons using **eq**. Since everything that is **eq** is also **eql**, and the only things which are **eql** but not **eq** are flonums, bignums and ratios (which should never have been used as tests for **selectq** in the past for this very reason) there should be no effect on any existing code. **selectq** and the COMMON LISP macro **case** are thus now identical.

2.4.10 CATCH and THROW

catch and **throw** used to be defined in a way which was compatible with Maclisp. (**catch** *form* **tag**) used to be what (**catch 'tag** *form*) is now, and (**throw** *form* **tag**) used to be what (**throw 'tag** *form*) is now. Since Maclisp itself has been issuing warnings for years saying to use ***catch**, this should cause no problems.

The implementation-related restrictions and general weirdness associated with the values from **catch** (a/k/a ***catch**) in older system versions have been fixed; **catch** now returns all the values from the last form executed (if no **throw** occurs) or else the values supplied by the second argument to **throw**.

In Release 2, **throw** can pass multiple values to **catch**: **catch** used to return exactly four arguments, of which the first one was a single value given to **throw**; the other three had complicated meanings. Now, **catch** returns any number of values: either the values thrown, or the values of the last form inside the **catch**, if no **throw** was done.

To throw more than one value, make the second subform of a ***throw** something which returns multiple values. Thus,

```
(catch 'foo (throw 'foo (values 'a 'b)))
```

returns the two values **a** and **b**.

In addition, **catch-all** now returns all the values of the body or all the values thrown, plus three more: the tag, action and count, *a la* ***unwind-stack**. (Yes, it is peculiar for a function to return *n* values followed by three specific ones, but it has to work that way.)

If you want to receive all these values, you should use **catch-all** within a **multiple-value-list** and then use **(butlast list 3)** to get the values thrown or returned and **(nleft 3 list)** to get the three specific values.

2.4.11 EVALHOOK/APPLYHOOK Incompatible Change

Evalhook and applyhook functions are now passed two additional arguments, which describe the interpreter environment that the evaluation or application was going to take place in. See the section on evaluation in the *Common Lisp Release Notes* for more information.

2.4.12 Changes to FORMAT control argument

~X (*HeX*)

Usage: **~***width,padchar,commachar***X** — Prints its argument in hexadecimal (analogous to **~O**, **~B** and **~D**). This command used to be used to insert spaces into the output.

Use **~***number-of-spaces***@T** to achieve the same result as the old **~***number-of-spaces***X** directive. .

~F (*Floating point*)

Usage: **~***width,decimal-places,scale,overflowchar,padchar***F** — Prints a floating-point number in nonexponential notation. Multiplies by 10^{scale} before printing if *scale* is specified. Prints in *width* positions, with *decimal-places* digits after the decimal point. Pads on left with *padchar* if necessary. If the number doesn't fit in *width* positions, and *overflowchar* is specified, this command just fills the *width* positions with that character.

This directive used to just take one optional prefix control arg, which specified how many mantissa digits to print. This is the same as *decimal-places*+2 for the new **format**. Use **~,***n+2***F** to achieve the same result as the old **~***n***F** directive.

~E (*Exponential*)

Usage: **~***width,decimal-places,exponent-places,scale,overflowchar,padchar,exptchar***E** — Prints a floating-point number in exponential notation. Prints in *width* positions, with *exponent-places* digits of exponent. If *scale* (default is 1) is positive, prints *scale* digits before point, *decimal-places-scale*+1 after. If *scale* is zero, prints *decimal-places* digits after the point, and a zero before if there's room. If *scale* is negative, prints *decimal-places* digits after the point, of which the first -*scale* are zeros. If *exptchar* is specified, it is used to delimit the exponent (instead of "e" or whatever.) If *overflowchar* is specified, then if the number doesn't fit in the specified *width*, or if the exponent doesn't fit in *exponent-places* positions, the field is filled with *overflowchar* instead.

This directive used to just take one optional prefix control arg, which specified how many mantissa digits to print. This is the same as *decimal-places*+2 for the new **format**. Use **~,***n+2***E** to achieve the same result as the old **~***n***E** directive.

~G (*Generalized floating-point*)

Usage: ~*width*,*decimal-places*,*exponent-places*,*scale*,*overflowchar*,*padchar*,*exptchar***G** — Like **~E**, but if the number fits without an exponent, it is printed without one.

This command used to be used to go to a particular argument. Use ~*argument-number***@*** to achieve the same result as the old ~*argument-number***G** directive.

2.4.13 New Treatment of Square Roots

sqrt *number* Function

Return the square root of *number*, returning a complex number if needed. In Release 1, if *number* was negative, a condition of the flavor **sys:negative-sqrt** would be signalled. However, since this error never occurs in Release 2, the condition flavor has been flushed.

2.5 Pointer Fields Now 25 Bits; Flag Bit Gone

Each typed data word in LISP machine memory used to have one bit called the "flag bit", which was not considered part of the contents of the word. This is no longer so. There is no longer a flag bit; instead, the pointer field of the word is one bit larger, making it 25 bits in all.

This extra bit extends the range of integers that can be represented without allocation of storage, and also extends the precision of small-floats by one bit.

On the LMI Lambda processor, the maximum size of virtual memory is doubled. This is the primary reason for the change. Unfortunately, the CADR mapping hardware is not able to use the extra bit as an address bit, so the maximum virtual memory size on a CADR is unchanged.

The functions **%24-bit-plus**, **%24-bit-difference** and **%24-bit-times** still produce only 24 bits of result. If you wish to have a result the full size of the pointer field, however wide that is, you should use the functions **%pointer-difference** and **%pointer-times** (the last is new), and **%pointer-plus** to do addition. (The new **%pointer-** functions are documented in more detail in the *Lisp Machine Manual.*)

The functions **%float-double**, **%divide-double**, **%remainder-double** and **%multiply-fractions** use the full width of the pointer field.

The values returned by **sxhash** have not changed. They are always positive fixnums less than 2^{23}.

Because of the change in pointer format, short-floats now have 17 bits of mantissa, 7 bits of exponent magnitude, and 1 bit of exponent sign. (Short floats used to have 16 bits of mantissa.)

2.6 EVAL-WHEN Rationalized

The treatment of **(eval-when (load)** *forms...***)** by the compiler is now identical to the treatment of forms encountered with no **eval-when**. They are put into the file to be evaluated on loading, or compiled if they are **defuns**, and any macros defined are made available for expansion during the compilation.

As a consequence, you can no-op an **eval-when** by supplying **(load eval)** as its first argument. It is then equivalent in all cases to no **eval-when** at all.

Nested **eval-whens** now effectively intersect their list of times to evaluate. As a result,

```
(eval-when (compile load eval)
```

```
compile-time-forms...
(eval-when (load eval)
  forms...))
```

treats the *forms* in the ordinary manner, overriding the special treatment given to the *compile-time-forms.*

```
(eval-when (compile) (eval-when (load) ignored-forms...))
```

does not do anything with the *ignored-forms.*

2.7 Change to SI:FULL-GC-INITIALIZATION-LIST

The **si:full-gc-initialization-list** initializations are now run before the garbage collection in **si:full-gc**, rather than after. A new initialization list, **si:after-full-gc-initialization-list**, is run after. The old list which now runs before GC can be requested with the keyword **:full-gc** in **add-initialization**, and the new list which runs after can be requested with **:after-full-gc.**

This change is for greater compatibility with Symbolics systems.

2.8 PROGV With More Variables Than Values

The function **progv** accepts a list of variables and a list of values. In the past, if the list of variables was longer, **nil** was used in place of the missing values. Now, in this case, the extra variables which have no corresponding values will be made "unbound." This is a COMMON LISP change.

2.9 %PAGE-STATUS Change

The subprimitive **%page-status** now returns the entire first word of the page hash table entry for a page, if the page is swapped in; or **nil** for a swapped-out page, or for certain low-numbered areas (which are all wired, so their pages' actual statuses never vary). The argument is an address in the page you are interested in--data type is irrelevant. The **%%pht1-** symbols in **SYS: SYS: QCOM LISP** are byte pointers you can use for decoding the value.

2.10 Character Bits Moved

The Control, Meta, Super, and Hyper bits now occupy a new position in character codes. This is so that they will not overlap the field used by the character's font number.

You can continue to use the byte pointers **%%kbd-control** to examine and set the bits; these byte pointers have different values now but your code will work anyway. No change to the source is needed.

2.11 Time Functions Return Exact Year

The functions **decode-universal-time, time:get-time** and **get-decoded-time** now return the correct year number (a number greater than 1900.) rather than the year number modulo 1900.

2.11.1 Primitive Printer Functions Changed

The functions **si:print-object** and **si:print-list** no longer accept the *slashify-p* argument. Instead, they look at the current value of ***print-escape***.

si:print-object *object prindepth stream* &optional *which-operations* Function
si:print-list *list prindepth stream* &optional *which-operations* Function

These are the primitive printer functions in the system. The recommended way to change the style of printed representation of all objects in the system is to **advise** these functions.

2.12 New List Matching Constructs

The syntax of **select-match** has been changed so as to avoid use of the construct **#?**. This is to avoid defining the construct **#?**, leaving it free for users to define. In addition, new instructions test extremely quickly whether a list has certain elements and then extract the others.

As before, **select-match** takes an expression for an object to be tested followed by any number of clauses to try. Each clause contains a pattern, a conditional form, and more forms that make the body of the clause. The first clause whose pattern matches the object and for which the conditional form produces a non-NIL value is the chosen clause, and its body is executed. The last clause can be an **otherwise** clause.

The change is that the pattern is now an expression made with the ‘ character, with commas indicating a variable in the pattern. For example, in

```
(select-match foo
  (‘(a ,b ,b) (symbolp b) (print b))
  (otherwise (print foo)))
```

the first clause matches if **foo** is a list of three elements, the first being the symbol **a**, and the second and third being the same symbol. The second clause matches anything that slips through the first.

select-match and **list-match-p** also accept logical combinations of patterns, using **and**, **or**, and **not** at top level. Note that matching specifications for patterns actually containing the symbols **and**, **or**, and **not** will not conflict with the use of this feature, since **select-match** and **list-match-p** are special forms which interpret their arguments specially.

```
(defun hack-add-sub (x)
  (select-match x
    ((or ‘(+ ,y 0) ‘(- ,y 0) ‘(+ y)) t
     y)
    ((not (or ‘(+ . ,ignore) ‘(- . ,ignore)))
     (ferror nil "You lose"))
    (t x)))
```

Note that variables used in the patterns (such as **y** in the example above) are bound locally by the **select-match**.

You can get the effect of a single **select-match** pattern with **list-match-p**:

list-match-p *list pattern* Macro

Returns **t** if the value of list matches pattern. Any match variables appearing in pattern will be set in the course of the matching, and some of the variables may be set even if the match fails.

2.13 BREAK Arguments Changed

The function **break** is being changed to accept a format string and format arguments. It used to take an unevaluated first argument, normally a symbol, and simply print it.

To make the changeover easier, **break** evaluates its first argument by hand, unless it is a symbol–then its pname is used as the format string. However, the compiler issues a warning if you use **break** in the old way.

2.14 Macro Expander Functions Take Two Arguments

A macro's expander function used to be passed only one argument, the macro call to be expanded. Now it is passed a second argument as well. It is an "environment" object, and it is used to record the local macro definitions currently in effect.

Since many old macros are still compiled to accept only one argument, **macroexpand-1** is smart and will pass only one argument in such a case. So there is no need to alter or recompile your macro definitions now.

However, if you have anything else that calls macro expander functions directly, it must be changed to do what **macroexpand-1** does. The easiest way is to write

(call *expander-function* **nil** *form* **:optional** *environment***)**

If you define a macro using **macro** (instead of **defmacro**), you should change the arglist yourself to accept a second optional argument, even if it is just **ignore**.

Another change to these functions is that they return a second value which is **t** if any expansion was done.

2.15 SETF and LOCF Definitions Done Differently

You no longer use **setf** and **locf** properties to define how to do **setf** or **locf** on some kind of form. Instead, you use the macro **defsetf** to define how to **setf** it, and you do

(deflocf *function* **...)**

to define how to do **locf** on it. See the section in the *Common Lisp Release Notes* that talks about **setf**.

One exception: **(defprop foo si:unsetfable setf)** still works, by special dispensation. Likewise for **si:unlocfable**. However, it is preferable to say, in the case of a function that should not allow **setf**, to say

(defsetf *function***)** or **(defun** *function* **si::nosetf)**

2.16 Y-OR-N-P And YES-OR-NO-P Arguments Changed

y-or-n-p *format-string* &rest *format-arguments* Function
yes-or-no-p *format-string* &rest *format-arguments* Function

These two functions now take just a format string and format arguments. They no longer accept the stream to use as an argument; they always use the value of ***query-io***.

If you used to pass two arguments, you must now bind ***query-io*** around the call instead.

2.17 COPY-FILE Takes Keyword Arguments

copy-file *filename new-name* &key (*error* **t**) (*copy-creation-date* **t**) (*copy-author* **t**) *report-stream* (*create-directories* **:query**) (*characters* **:default**) (*byte-size* **:default**) Function

Copies the file named *filename* to the file *new-name*.

characters and *byte-size* specify what mode of I/O to use to transfer the data. *characters* can be **t** to specify character input and output, **nil** for binary, **:ask** meaning ask the user which one, **:maybe-ask** meaning ask if it is not possible to tell with certainty which method is best, or **:default** meaning to guess as well as possible automatically.

If binary transfer is done, *byte-size* is the byte size to use. **:default** means to ask the file system for the byte size that the old file is stored in, just as it does in **open**.

The *copy-author* and *copy-creation-date* arguments say whether to set those properties of the new file to be the same as those of the old file. If a property is not copied, it is set to your login name (for the machine on which the target file resides) or the current date and time.

report-stream, if non-**nil**, is a stream on which a message should be printed describing the file copied, where it is copied to, and which mode was used.

create-directories says what to do if the output filename specifies a directory that does not exist. It can be **t** meaning "create the directory", **nil** meaning "treat it as an error", or **:query** meaning ask the user which one to do.

error, if **nil**, means that if an error happens then this function should just return an error indication.

If *filename* contains wildcards, multiple files are copied. The new name for each file is obtained by merging *new-name* (parsed into a pathname) with that file's truename as a default. The mode of copy is determined for each file individually, and each copy is reported on the report-stream if there is one. If error is **nil**, an error in copying one file does not prevent the others from being copied.

The value returned is a list with one element for each file which was to be copied. Each element is either an error object, if an error occurred copying that file (and error was **nil**), or a list (*old-truename new-truename characters*). The two truenames are those of the file copied and the newly created copy. characters is **t** if the file was copied in character mode. The value can also be just an error object, if an error happened in making a directory listing to find out which files to copy (for a wildcard pathname).

2.18 MAKE-PATHNAME Change

The meaning of the defaults argument to **make-pathname** is changed. Now all pathname components that are not specified or specified as **nil** are defaulted from the defaults, if you give defaults. If you do not give defaults, then the host alone defaults from ***default-pathname-defaults***, as it used to.

3. Compatible LISP Programming Changes

3.1 All Objects Except Symbols and Lists Are Constants

All arrays, instances, fefs, characters, closures, etc. now evaluate to themselves. Evaluating such objects used to be an error; this new behavior therefore cannot hurt anything. Keywords (see section 2.4.8, page 8), which are symbols in the **keyword** package, also evaluate to themselves.

The only kinds of objects that currently can evaluate to anything but themselves are symbols and lists. However, it is not guaranteed that no other kind of object will ever be defined to evaluate to other than itself.

3.2 Nonlocal GO and RETURN

You can now **go** or **return** from an internal **lambda** expression to the containing function. Example:

```
(prog ()
   (mapc #'(lambda (x) (if (numberp x) (return T)))
         inputs))
```

returns **t** if any element of **inputs** is a number. So does

```
(prog ()
  (mapc #'(lambda (x) (if (numberp x) (go ret-t)))
        inputs)
  (return nil)
 ret-t
  (return t))
```

3.3 Common Lisp Control Constructs BLOCK and TAGBODY

block takes a block name and a body:

```
(block name body...)
```

and executes the *body*, while allowing a **return-from** *name* to be used within it to exit the **block**. If the *body* completes normally, the values of the last body form are the values of the **block**.

A **block** whose *name* is **nil** can be exited with plain **return**, as well as with **(return-from nil)**.

block can be thought of as the essence of what named **progs** do, isolated and without the other features of **prog** (variable binding and **go** tags).

Every function defined with **defun** whose name is a symbol contains an automatically generated **block** whose name is the same as the function's name, surrounding the entire body of the function.

tagbody, on the other hand, is the essence of **go** tags. A **tagbody** form contains statements and tags, just as a **prog**'s body does. A symbol in the **tagbody** form is a tag, while a list is a statement to be evaluated. The value returned by a **tagbody** is always **nil**. **tagbody** does not have anything to do with **return**.

prog is now equivalent to a macro

```
(macro prog (form)
     (let* ((name (and (symbolp (cadr form)) (cadr form)))
            (vars (if name (caddr form) (cadr form)))
            (body (if name (cdddr form) (cddr form))))
       (if name
           '(block ,name
              (block nil
                (let ,vars
                    (tagbody . ,body))))
         '(block nil
            (let ,vars
              (tagbody . ,body))))))
```

if we ignore the added complication of **progs** named **t** and **return-from-t**.

3.4 LEXPR-FUNCALL And APPLY Now Synonymous.

apply now accepts any number of arguments and behaves like **lexpr-funcall**. **lexpr-funcall** with two arguments now works the way **apply** used to, passing an explicit rest-argument rather than spreading it. This eliminates the old reasons why **lexpr-funcall** was not the best thing to use in certain cases, and paves the way for **apply** to translate into it. **lexpr-funcall** is now considered somewhat obsolete

3.5 :ALLOW-OTHER-KEYS As A Keyword Argument

:allow-other-keys has a special meaning as a keyword when passed to a function that takes **&key** arguments. If followed by a non-**nil** value, it prevents an error if any keyword is not recognized. Thus, given the function

```
(defun foo (&key a b) (list a b))
```

you would get an error if you do `(foo :a 5 :c t)` because **:c** is not recognized. But if you do

```
(foo :a 5 :c t :allow-other-keys t)
```

you get no error. The **:c** and its argument are just ignored.

3.6 GET and GETHASH with Three arguments.

get and **gethash** now take an optional third argument, which is a default value to be returned as the value if no property or hash table entry is found.

3.7 New Macros TYPECASE, PUSHNEW

There is now a **typecase** macro, compatible with COMMON LISP. See the *Lisp Machine Manual* for details.

pushnew pushes an element onto a list only if it was not there (using **cli:member**) before.

(pushnew *elt place***)**

is equivalent to

```
(or (cli:member elt place)
    (setf place (adjoin elt place)))
```

except that *elt* and *place* are evaluated only once. The value returned by **pushnew** is the new list. The keywords **:key**, **:test**, and **:test-not** are accepted by **pushnew**; they get passed to along to **cli:member** to change the test for the "newness" of *elt*.

3.8 Microcoded Functions Interruptible

Many microcoded functions, including **last**, **memq**, **assq** and **get**, are now interruptible. This means in particular that if you pass a circular list to any of them you can now abort successfully.

3.9 Selecting a Returned Value

The function **nth-value** makes it convenient to select one of the values returned by a function. For example, **(nth-value 1 (foo))** returns the second of **foo**'s values. **nth-value** operates without consing in compiled code if the first argument's value is known at compile time.

nth-value *value-number expression* — Special form

Evaluates *expression*, then returns one of its values as specified by *value-number* (with **0** selecting the first value).

3.10 New types NON-COMPLEX-NUMBER and REAL

(typep *x* **(non-complex-number** *low high***))** returns **t** if *x* is a non-complex number (ie a floating-point number, a ratio or an integer) between *low* and *high*, the limits as usual being inclusive normally,or exclusive if they consist of a list of one element. Note that complex-numbers with an imaginary part of 0 are never of the type **non-complex-number**, since they are always of type **complex**. To account for this additional case, there is another new type, **real**, which is defined such that **(typep** *x* **(real** *low high***))** returns **t** if *x* is a either a non-complex number between *low* and *high*, or a complex number with a zero imaginary part and a real part lying between *low* and *high*.

3.11 Remainder, Log Functions Extended

“ *x y* — Function

remainder *x y* — Function

In Release 1, the remainder function only took integer (fixnum & bignum) arguments. In Release 2, it takes any sort of numeric arguments, and returns whatever is necessary to represent the exact result, as per the **Common Lisp** specification.

log *n* &optional (*base* **(exp 1)**) — Function

Return the base *base* logarithm of *n*, where *base* defaults to *e*. Previously, **log** only took one argument, and the base always *e*.

3.12 New Arithmetic Condition

Because COMMON LISP has a such a rich structure of numeric types, there are now cases (especially in the transcendental functions) where raising a number to a power may produce an undefined result.

sys:illegal-expt (sys:arithmetic-error) Condition

The condition **sys:illegal-expt** is signalled whenever an attempt is made to raise a number to a power in some case where the result is not defined. The condition supports the following operations:

:base-number

The base of the exponentiation (the first argument to **expt**, for example).

:power-number

The power of the exponentation (the second argument to **expt**, for example).

3.13 Macro Changes

Because of COMMON LISP, some subtle changes have occurred in the behavior of macros in interpreted code. Macro expander functions now take another argument, the lexical environment, to account for macros which need to be aware of the local macro definitions.

3.13.1 All Macros Are Displacing In Interpreted Code

All macros are displacing when encountered by **eval**. **defmacro-displace**, and so on, are now synonyms for **defmacro**, and so on. This is not exactly a compatible change for the interpreter. It was always made clear in the **Lisp Machine Manual** that part of a *compiled* function's behavior would be affected by the state of the macros it used at *compile*-time, no matter if the macro was displacing or not. On the other hand, it would matter in the interpreter whether the macro *was* displacing or not. If a macro defined with **macro** (not a COMMON LISP construct, by the way) or **defmacro** changed between invocations of a interpreted function that used it, the change would be seen by the function, because the macro would get expanded every time it is encountered by the interpreter. On the other hand, when a macro call uses a displacing macro, it is really expanded only once: the first time it is seen. So, if the macro changes, the changes will not noticed by the interpreter if it encounters a macro call which it has already expanded.

Note that this behavior is closer to being analogous to the compiler, but not exactly so. In order for that to be true, the interpreter would have to expand the macros in function-making forms (the **def** family and **lambda**) immediately. In general, COMMON LISP implementations are free to expand macros whenever they see fit, so users should be wary of depending on the implementation to notice changes in their macros when using interpreter.

3.13.2 MACROEXPAND-ALL

The function **macroexpand-all** is called like **macroexpand**. It expands macro definitions not only at the top level of a form but also in its subexpressions. It is never confused by a macro name, appearing at the start of a list, that is not a subexpression.

macroexpand-all *form* &optional *environment* Function

Expands macro definitions at all levels in *form* and returns the result. *environment* is used for finding local **macrolet** macro definitions; it is like the second argument to **macroexpand** (see previous page).

Only one value is returned.

3.13.3 New Function DEFF-MACRO

deff-macro &quote *function-spec* &eval *definition* Special form

defines *function-spec* as *definition*, just like **deff**. The difference comes in compiling a file, where the compiler assumes that **deff-macro** is defining a macro and makes the definition available for expansion during this compilation. **deff**, on the other hand, is just passed through to be evaluated when the file is loaded. To use **deff-macro** properly, *definition* must be a list starting with **macro** or a suitable subst function (a list starting with **subst** or a compiled function which records an interpreted definition which is a list starting with **subst**).

3.13.4 DEFINE-SYMBOL-MACRO

define-symbol-macro has not been implemented in LMI/MIT ZETALISP.

The effect of **(define-symbol-macro foo (print 'huh))** would be that evaluating the symbol **foo** would execute **(print 'huh)**. "Binding" such a symbol with **let** would probably have undefined or counterintuitive behavior.

If users find this useful or necessary for compatibility with Symbolics systems, it will be implemented.

3.14 Named Structure Operations

You can now **funcall** a named structure to invoke a generic operation on it, just as you would a flavor instance. In fact, you can have code which operates on named structures and flavor instances indiscriminately, if you make sure that the named structures you are using support whichever operations you plan to use.

For example,

```
(send *package* :describe)  ; Use send here to make it clear.
```

invokes the **:describe** operation on the current package, just as

```
(named-structure-invoke :describe *package*)
```

would do.

Invoking a named structure has not been made ultra-fast, but that can bedone in a future microcode release.

3.14.1 DEFSELECT and Named Structures

defselect, by default, defines the function to signal an error if it is called with a first argument not defined in the **defselect** (except for **:which-operations**, which is defined implicitly by **defselect**).

If you use **defselect** to define the handler function for a named structure type, and you use this default behavior, you will get errors at times when the system invokes operations that you may not know or care about, such as **:sxhash** or **:fasload-fixup**.

To avoid this problem, specify **ignore** as the default handler in the **defselect**. **ignore** accepts any arguments and returns **nil**. Also, **defselect-incremental** (see page 23) may be useful when defining a set of operations on a named structure.

3.14.2 Named Structure Operation :FASLOAD-FIXUP

The named structure operation **:fasload-fixup** is invoked by **fasload** whenever a named structure is created according to data in a QFASL file. This operation can do whatever is necessary to make the structure properly valid, in case just reloading it with all its components is not right. For most kinds of structures, this operation need not do anything; it is enough if it does not get an error.

3.15 DEFSUBST Preserves Order of Evaluation

It used to be the case that if a **defsubst**'s body used an argument more than once, or used its arguments out of order, the forms supplied as arguments would be evaluated multiple times or in the wrong order. This has been fixed. The arguments passed to a **defsubst** function will be evaluated exactly once, in the order they are written.

For example, after **(defsubst foo (a b) (cons b a))**, the reference **(foo x (setq x y))** used to turn into **(cons (setq x y) x)**, which is incorrect since it uses the new value of **x** twice. To be correct, the old value of **x** should be used for the second argument to **cons**.

Now, the expansion will be something effectively like

```
(let ((temp x)) (cons (setq x y) temp))
```

3.16 CAR-SAFE, Etc.

car-safe *x* — Function

car-safe is like **car** when operating on a list. If *x* is not a list, **car-safe** returns **nil**. **car-safe** never gets an error.

cdr-safe *x* — Function
cddr-safe *x* — Function
nth-safe *n x* — Function
nthcdr-safe *n x* — Function

These are other functions which are analogous to **car-safe**. If *x* is not a cons, **nil** is returned.

3.17 Global Value Functions

There are now functions to use to examine or set the global binding of a variable, as opposed to the binding currently in effect. The global binding is the one that is in effect in all processes or stack groups that have not rebound the variable.

They work by forking off another process and examining or setting the variable in that process. The bindings of your own process are *not* visible in the other process, and that process establishes no bindings of its own, so references to the symbol there access the global binding.

symeval-globally *symbol* — Function

Returns the global binding of *symbol*.

setq-globally *unevaluated-symbol value unevaluated-symbol value...* — Function

Sets the global binding of each symbol to the corresponding *value*.

set-globally *symbol value* — Function

Sets the global binding of *symbol* to *value*. *symbol* is an evaluated argument.

makunbound-globally *symbol* — Function

Makes the global binding of **symbol** be void.

boundp-globally *symbol* Function

Returns **t** if the global binding of *symbol* is not void.

These functions are used primarily so that init files can set variables that are bound by the **load** function, such as **package** or **base**. If your init file does

```
(setq package (find-package 'foo))
```

this will be nullified as soon as **load** exits and its binding of **package** goes away. If you do

```
(setq-globally package (find-package 'foo))
```

the current binding established by **load** is actually not changed, but when the **load** exits and the global binding is in effect again, **foo** will be the current package.

3.18 LOCATION-MAKUNBOUND Takes Two Arguments

location-makunbound now takes a second, optional argument. This argument supplies a pointer value to use in the void marker that is stored.

A void location actually contains a pointer with data type **dtp-null**. This pointer is supposed to point to the object whose value or function definition is void. In the case of a symbol's value cell or function cell, the object would be the symbol itself.

location-makunbound makes the location point to whatever object you supply as the second value.

3.19 DEFSELECT-INCREMENTAL

defselect-incremental *fspec default-handler* Special form

With **defselect-incremental** you can define a **defselect** that starts out empty and has methods added to it incrementally with individual **defuns**.

You do **(defselect-incremental** *fspec default-handler***)** to define fspec as a select-method function that has no methods except the standard ones (**:which-operations**, **:operation-handled-p**, and **:send-if-handles**).

Then, to define the individual methods, use **defun** on function specs of the form **(:select-method** *fspec operation***)**. Note that the argument list of the **defun** must explicitly provide for the fact that the operation will be the first argument; this is different from what you do in an ordinary **defselect**. Example:

```
(defselect-incremental foo ignore)
; The function ignore is the default handler.
(defun (:select-method foo :lose) (ignore a)
  (1+ A))
```

defines **FOO** just like

```
(defselect (foo ignore)
  (:lose (a) (1+ a)))
```

The difference is that reevaluating the **defselect** gets rid of any methods that used to exist but have been deleted from the **defselect** itself. Reevaluating the **defselect-incremental** has no such effect, and reevaluating an individual **defun** redefines only that method.

3.20 :NO-ERROR Clauses in CONDITION-CALL

The last clause in a **condition-call** or **condition-call-if** may now be a **:no-error** clause. This looks and works about the same as a **:no-error** clause in a **condition-case**: it is executed if the body returns without error. The values returned by the body are stored in the variables that are the elements of the list that is the first argument of the **condition-call**, and the values of the last form in the clause are returned by the **condition-call** form itself.

3.21 Top Level Forms Specially Treated In The Compiler

Following is a partial list of symbols, which, when appearing as the first element of a top-level form, will cause that form to be treated specially by the compiler. Only those whose meanings have changed, or require clarification, are listed here.

progn Treat all following forms as if they also were at top level. Note that in Maclisp and in Release 1 and earlier, it was necessary for the first form of the body to be **'compile** for this to happen. This curious behaviour has been eliminated.

proclaim The arguments are evaluated, and relevant proclamations (such as **special**, **notinline**) are used in the remainder of the compilation. This is as if the form were contained within a **(eval-when (eval compile load) ...)**

export import in-package make-package shadow
shadowing-import unexport unuse-package use-package
These perform their relevant actions as if the form contained within a `(eval-when (eval compile load) ...)`.

require Ditto; this is relevant for COMMON LISP modules.

To cause a form not to be treated specially at top-level by the compiler, enclose it in an **eval-when**. Eg:

```
(eval-when (load) ; don't want this package to be consed up when we're just compiling
  (make-package "lossage" :use nil :size 69))
```

3.22 Compiler Optimization Changed

Many compiler optimizers have been reimplemented, and should often produce better code. The most visible change is that any form is only optimized once, no matter where it appears. (In earlier systems, a form could sometimes be optimized twice, which could produce duplicate compiler warnings) In addition, the order in which optimizations are carried out has changed. All the arguments to a function are optimized before the call to the function on those arguments, unless the "function" is a macro or special form, in which case it is expected to take responsibility for doing its own optimizations.

3.23 TV:BITBLT-CLIPPED

tv:bitblt-clipped is just like **tv:bitblt**, except that if you specify transfers that include points outside the bounds of either the source or destination array, only the part of the transfer that is within the bounds of both arrays will take place.

The height and width you specify must be positive.

3.24 %BLT-TYPED. Proper Use of Pointer Subprimitives

%blt-typed is called just like **%blt** and does about the same thing: it copies any number of consecutive memory words from one place in memory to another. The difference is that **%blt** is only properly used on data that contains no pointers to storage, while **%blt-typed** is only properly used on boxed data.

Both **%blt** and **%blt-typed** can be used validly on data that is formatted with data types (boxed) but whose contents never point to storage. This includes words whose contents are always fixnums or small flonums, and also words that contain array headers, array leader headers, or FEF headers. Whether or not the machine is told to examine the data types of such data makes no difference since, on examining them, it would decide that nothing needed to be done.

For unboxed data (data that is formatted so as not to contain valid data type fields), such as the inside of a numeric array or the instruction words of a FEF, only **%blt** may be used. If **%blt-typed** were used, it would examine the data type fields of the data words and would probably halt due to an invalid data type code.

For boxed data that may contain pointers, only **%blt-typed** may be used. If **%blt** were used, it would appear to work, but problems could appear mysteriously later because nothing would notice the presence of the pointer there. For example, the pointer might point to a bignum in the number consing area; moving it with **%blt** would fail to copy it into a nontemporary area. Then the pointer would become invalidated the next time the number consing area was emptied out. There could also be problems with lexical closures and with garbage collection.

%p-store-tag-and-pointer should be used only for storing into boxed words, for the same reason as **%blt-typed**: the microcode could halt if the data stored is not valid boxed data.

%p-dpb and **%p-dpb-offset** should be used only when the word being modified does not contain a pointer. It may be an unboxed word, or it may be a boxed word containing a fixnum, small-flonum or array header. The same goes for **%p-deposit-field** and **%p-deposit-field-offset**.

Here are some new subprimitives that test values for pointerhood.

%pointerp *object* — Function

returns non-**nil** if *object* points to storage. For example, **(%pointerp "foo")** is **t**, but **(%pointerp 5)** is **nil**.

%p-pointerp *location* — Function

returns non-**nil** if the contents of the word at *location* points to storage. This is similar to **(%pointerp (contents** *location***))**, but the latter may get an error if *location* contains a forwarding pointer, a header type, or an void marker. In such cases, **%p-pointerp** will correctly tell you whether the header or forward points to storage.

%p-pointerp-offset *location offset* — Function

similar to **%p-pointerp** but operates on the word *offset* words beyond *location*.

%p-contents-safe-p *location* — Function

returns non-**nil** if the contents of word *location* are a valid Lisp object, at least as far as data type is concerned. It is **nil** if the word contains a header type, a forwarding pointer, or an unbound marker. If the value of this function is non-**nil**, you will not get an error from **(contents** *location***)**.

%p-contents-safe-p-offset *location offset* — Function

similar to **%p-contents-safe-p** but operates on the word *offset* words beyond *location*.

%p-safe-contents-offset *location offset* — Function

returns the contents of the word *offset* words beyond *location* as accurately as possible without getting an error.

- If the data there are a valid Lisp object, it is returned exactly.
- If the data are not a valid Lisp object but do point to storage, the value returned is a locative which points to the same place in storage.
- If the data are not a valid LIsp object and do not point to storage, the value returned is a fixnum with the same pointer field.

%pointer-type-p *data-type* Function

returns non-**nil** if the specified data type is one which points to storage. For example, **(%pointer-type-p dtp-fix)** returns **nil**.

3.25 Growing the Stack

When the PDL (stack) overflows, a condition is signalled, and the process usually falls in the debugger. If a function is going to use up a lot of stack space, then the function **eh:require-pdl-room** can be used to grow the stack, and thus avoid the debugger.

eh:require-pdl-room *regpdl-space specpdl-space* Function

Makes the current stack group larger if necessary, to make sure that there are at least *regpdl-space* free words in the regular pdl, and at least *specpdl-space* free words in the special pdl, not counting what is currently in use.

3.26 Flavor Changes

3.26.1 Delaying Flavor Recompilation

si:*dont-recompile-flavors* Variable

Normally the system recompiles combined methods automatically when you make a change that requires this. If you plan to make more than one change, you might wish to recompile only once. To do this, set the variable **si:*dont-recompile-flavors*** non-**nil** before you make the changes. Then set it back to **nil**, and use **recompile-flavor** to perform the appropriate recompilations.

3.26.2 Method Combination Improvements

When using method combination types such as **:list**, **:progn**, **:append** and **:pass-on**, which formerly allowed any number of untyped methods and nothing else, you can now use the method combination type keyword as a method type. For example, when using **:or** combination for operation **:doit**, you can now define a method **(myflavor :or :doit)** as well as **(myflavor :doit)**. The method is combined the same way whichever name you use. However, when the operation is invoked, all the typed methods are called first, followed by all the untyped methods.

There is no longer a limit of three values passed back from the primary method when **:after** methods are in use. As many values as the primary method chooses to return will be passed back to the ultimate caller.

3.26.3 Undefinition

undefflavor *flavor-name* Function

Removes the definition of *flavor-name*. Any flavors that depend on it are no longer valid to instantiate.

3.26.4 New DEFFLAVOR options

Several new keyword options for **defflavor** have been added for Release 2.

:instance-area-function *function*

This feature can control in which area flavor instances are consed, on a per-flavor basis, by giving a flavor an instance-area function. This is a function which will be called whenever the flavor is instantiated, and expected to return the area to cons in (or **nil**, if it has no opinion). The function is passed one argument, the init-plist, so if you want to have an init option for the caller to specify the area, the instance-area function can use **get** to get the value the caller specified.

The instance-area function is inherited by flavors which use this one as a component.

:required-init-keywords *init-keywords...*

This option specifies that each of the keywords in *init-keywords* must be provided when trying to make an instance of this flavor. Then, whenever the flavor (or any flavor that depends on it) is instantiated, it will be an error if any of those init keywords fails to be specified. For example, after

```
(defflavor foo (a) :inittable-instance-variables
   (:required-init-keywords :a))
```

it is an error to do **(make-instance 'foo)** since the **:a** keyword is not provided.

:instantiation-flavor-function *function-name*

This allows a flavor to compute what flavor **make-instance** will actually use. When a flavor which uses this option is passed to **make-instace**, it calls a function to decide what flavor it should really instantiate (not necessarily the original flavor).

When **(make-instance 'foo** *keyword-args***)** is done, the function specified is called with two arguments: the flavor name specified (**foo** in this case) and the init plist (the list of keyword args). It should return the name of the flavor that should actually be instantiated.

Note that the instantiation flavor function applies only to the flavor it is specified for. It is not inherited by dependent flavors.

:run-time-alternatives *clauses...*

:mixture *clauses...*

A run-time-alternative flavor is a way to define a collection of similar flavors, all built on the same base flavor but having various mixins as well, and choose which one to instantiate based on init options. (This is implemented using the **:instantiation-flavor-function** feature.)

A simple example would be:

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
     (:big big-foo-mixin))
  (:init-keywords :big))
```

Then, **(make-instance 'foo :big t)** will get you an instance of a flavor whose components are **big-foo-mixin** as well as **foo**. But **(make-instance 'foo)** or **(make-instance 'foo :big nil)** will get you an instance of **foo** itself. The clause **(:big big-foo-mixin)** in the **:run-time-alternatives** says to incorporate **big-foo-mixin** if **:big**'s value is **t**, but not if it is **nil**.

You can have several clauses in the **:run-time-alternatives**. Each one is processed independently. Thus, you could have keywords **:big** and **:wide** independently control two mixins and get four possibilities.

You can test for values of a keyword other than just **t** or **nil**. The clause

```
(:size (:big big-foo-mixin) (:small small-foo-mixin)
       (nil nil))
```

allows the value for the keyword **:size** to be **:big**, **:small**, or **nil** (or omitted). If it is **nil** or omitted, no mixin is used (that's what the second **nil** means). If it is **:big** or **:small**, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values you check for can be anything, but **eq** is used to compare them.

You can also have the value of one keyword control the interpretation of others by inserting clauses within clauses. After the place where you put the mixin name or **nil** for no mixin, you can put other clauses which specify keywords and their interpretation. These other clauses are acted on only if the containing alternative is chosen. For example, the clause

```
(:etherial (t etherial-mixin)
           (nil nil
        (:size (:big big-foo-mixin) (:small small-foo-mixin)
          (nil nil))))
```

says to consider the **:size** keyword only if **:etherial** is **nil**.

3.27 File System Changes

3.27.1 Additional Arguments to FS:PARSE-PATHNAME

fs:parse-pathname *thing* &optional *with-respect-to defaults* (*start* **0**) *end junk-allowed* Function

Parses *thing* into a pathname and returns it. *thing* can be a pathname, a string or symbol, or a Maclisp-style namelist. If it is a pathname, it is returned unchanged, and the other arguments do not matter. *with-respect-to* can be **nil** or a host or a host-name.

- If it is not **nil**, the pathname is parsed for that host and it is an error if the pathname specifies a different host.
- If *with-respect-to* is **nil**, then defaults is used to get the host if none is specified. *defaults* may be a host object in this case.

start and *end* are indices specifying a substring of thing to be parsed. They default to 0 for start and **nil** (meaning end of thing) for end.

- If junk-allowed is non-**nil**, parsing stops without error if the syntax is invalid, and this function returns **nil**. The second value is the index in thing at which parsing stopped, which is the index of the invalid character if there was invalid syntax.
- If junk-allowed is **nil**, invalid syntax signals an error.

3.27.2 Merging Pathname Components

fs:merge-pathname-components *pathname* &optional *defaults* &key *default-name always-merge-name default-type always-merge-type default-version always-merge-version* Function

This function extends the functionality of both the commonlisp function **merge-pathnames** and the old Lisp Machine function **fs:merge-pathname-defaults.**

merge-pathname-components defaults components that are of *pathname* which are **nil**, and returns the defaulted pathname. *defaults* is a pathname or a defaults-list to get defaults from. If non-**nil**, *default-name, default-type* and *default-version* respectively are used as the defaults for the name, type and version components if those components are not supplied by *pathname.* Otherwise, those components are defaulted from *defaults* in the usual manner. *always-merge-name, always-merge-type* and *always-merge-version* respectively mean that the version and type components should always be merged in (from either *default-xxx* or from *defaults*) even if the relevant component is already specified by *pathname.*

(merge-pathnames pathname defaults default-version) is thus equivalent to:

```
(merge-pathnames-components pathname default
                            :default-version default-version
                            :always-merge-version t)
```

since COMMON LISP specifies that the *default-version* argument to *merge-pathnames* is merged into the resulting even if *pathname* already had a version component.

fs:merge-pathname-components differs from **fs:merge-pathname-defaults** in that it performs *only* the merging operation of filling **nil** components of one pathname with (possibly **nil**) components from the defaults, whereas **fs:merge-pathname-defaults** will *never* return a pathname with a **nil** name or type component.

fs:merge-pathname-defaults is thus a function useful for defaulting a pathname that the user has just entered for some purpose, such as to be read. **fs:merge-pathname-componments** will perform a single merging (and *may* return a pathname which is not accceptable for performing file operations upon — such as a pathname with a name of **nil.**) It is useful for programs which need to manipulate filenames in an exact manner (such as the file server) and do not want any user-oriented heuristics happening "behind its back." It ignores such variables as ***always-merge-type-and-version** and ***name-specified-default-type***, which **fs:merge-pathname-defaults** uses. **merge-pathnames** is a simpler version of **fs:merge-pathname-components** which COMMON LISP implementations understand.

A typical use of **fs:merge-pathname-components** is

```
(setq qfasl-file
      (fs:merge-pathname-components qfasl-file lisp-file
                                    :always-default-version t
                                    :default-type :qfasl
                                    :always-default-type t))
```

which will produce a file whose version is the same as that of **lisp-file** and whose type is always **qfasl**, and whose other components are the (perhaps **nil**) results of merging the components of **lisp-file** with **fasl-file**.
Some examples:

```
(setq pn1 (make-pathname :host twenex-host :name "FOO" :version 259))
  => #⊂FS::TOPS20-PATHNAME "TWENEX:FOO.⇌.259"⊃
(setq pn2 (make-pathname :host twenex-host :device "DP" :type :TEXT))
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:⇌.TEXT.⇌"⊃

(fs:merge-pathname-components pn1 pn2)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn1 pn2 :default-version 5)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn1 pn2 :default-version 5
                              :always-merge-version t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.5"⊃
(fs:merge-pathname-components pn1 pn2 :default-version 5
                              :default-type :lisp
                              :always-merge-version t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.LISP.5"⊃

(fs:merge-pathname-components pn2 pn1)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn2 pn1 :always-merge-type t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.⇌.259"⊃ ; merges in null type!
(fs:merge-pathname-components pn2 pn1 :default-type :lisp)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.TEXT.259"⊃
(fs:merge-pathname-components pn2 pn1 :default-type :lisp
                              :always-merge-type t)
  => #⊂FS::TOPS20-PATHNAME "TWENEX:DP:FOO.LISP.259"⊃
```

3.27.3 Logical Hosts

Logical hosts can now have their translations specified by pattern matching, instead of using just literal directory names. A translation now consists of a pair of pathnames or namestrings, typically containing wildcards. Unspecified components in them default to **:wild**. The *from*-pathname of the translation is used to match against the pathname to be translated; if it matches, the corresponding *to*-pathname is used to construct the translation, filling in its wild fields from the pathname being translated as in the **:translate-wild-pathname** operation.

Most commonly the translations contain pathnames that have only directories specified, everything else wild. Then the other components are unchanged by translation.

Each translation is specified as a list of two strings. The strings are parsed into pathnames and any unspecified components are defaulted to **:wild**. The first string of the pair is the source pattern; it is parsed with logical pathname syntax. The second string is the target pattern, and it is parsed with the pathname syntax for the specified physical host.

For example, suppose that logical host **FOO** maps to physical host **BAR**, a Tops-20, and has the following list of translations:

```
(("BACK;" "PS:<FOO.BACK>")
 ("FRONT;* QFASL" "SS:<FOO.QFASL>*.QFASL")
 ("FRONT;" "PS:<FOO.FRONT>"))
```

Then all pathnames with host **FOO** and directory **BACK** translate to host **BAR**, device **PS** and directory **<FOO.BACK>** with name, type and version unchanged. All pathnames with host **FOO**, directory **FRONT** and type **QFASL** translate to host **BAR**, device **SS**, directory **<FOO.QFASL>** and type **QFASL**, with name and version unchanged. All other pathnames with host **FOO** and directory **FRONT** map to host **BAR**, device **PS** and directory **<FOO.FRONT>**, with name, type and version unchanged. Note that the first translation whose pattern matches a given pathname is the one that is used.

Another site might define **FOO**'s to map to a Unix host **QUUX**, with the following translation list:

```
(("BACK;" "//nd//foo//back//")
 ("FRONT;" "//nd//foo//front//"))
```

This site apparently does not see a need to store the **QFASL** files in a separate directory. Note that the slashes are duplicated to quote them for Lisp; the actual namestrings contain single slashes as is usual with Unix.

If the last translation's source pattern is entirely wild, it applies to any pathname not so far handled. Example:

```
(("BACK;" "//nd//foo//back//")
 ("" "//nd//fool//*//"))
```

fs:add-logical-pathname-host *logical-host physical-host translations* Function
fs:set-logical-pathname-host *logical-host* &key *physical-host translations* Function

Both create a new logical host named *logical-host*. Its corresponding physical host (that is, the host to which it should forward most operations) is *physical-host*. *logical-host* and *physical-host* should both be strings. *translations* should be a list of translation specifications, as described above. The two functions differ only in that one accepts positional arguments and the other accepts keyword arguments. Example:

```
(add-logical-pathname-host "MUSIC" "MUSIC-10-A"
      '(("MELODY;" "SS:<MELODY>")
        ("DOC;" "PS:<MUSIC-DOCUMENTATION>")))
```

This creates a new logical host called **MUSIC**. An attempt to open the file

`MUSIC:DOC;MANUAL TEXT 2`
will be re-directed to the file
`MUSIC-10-A:PS:<MUSIC-DOCUMENTATION>MANUAL.TEXT.2`

(assuming that the host **MUSIC-10-A** is a TOPS-20 system).

fs:make-logical-pathname-host *name* Function

Requests that the definition of logical host *name* be loaded from a standard place in the file system: namely, the file **SYS: SITE:** *name* **TRANSLATIONS**. This file is loaded immediately with **load**, in the **fs** package. It should contain code to create the logical host; normally, a call to **fs:set-logical-pathname-host** or **fs:add-logical-pathname-host**, above.

The same file is automatically reloaded, if it has been changed, at appropriate times: by **load-patches**, and whenever site information is updated.

3.27.4 :DEVICE-WILD-P, etc., Pathname Operations

The operation **:device-wild-p** operation on a pathname object is defined to return non-**nil** if the pathname's device component contains a wildcard.

:directory-wild-p, **:name-wild-p**, **:type-wild-p** and **:version-wild-p** are similar, for their respective pathname components.

3.27.5 WITH-OPEN-FILE-SEARCH

with-open-file-search is a new macro for opening a file and trying various pathnames until one of them succeeds. The pathnames tried differ only in their type components.

with-open-file-search (*streamvar* (*operation defaults auto-retry*) *types-and-pathname . options*) &body *body* — Macro

Tries opening various files until one succeeds; then binds streamvar to the stream and executes body, closing the stream on exit.

types-and-pathname should evaluate to two values, the first being a list of types to try and the second being a pathname, called the base pathname. Each pathname to try is made by merging the base pathname with the defaults defaults and one of the types. options should evaluate alternately to keywords and values that are passed to **open**.

If all the names to be tried fail, a **fs:multiple-file-not-found** error is signaled. *operation* is provided just so that the **:operation** operation on the error object can return it. It is usually the user-level function for that the **with-open-file-search** is being done.

If *auto-retry* is non-**nil**, an error causes the user to be prompted for a new base pathname. The entire set of types specified is tried anew with this pathname.

3.27.6 New :PROPERTIES operation on file streams

Sending a **:properties** message to a file stream returns two values: a property list, like the kind which is a element of the list returned by **fs:directory-list**, and a list of settable properties. There is the usual optional *error-p* argument, as well. This operation uses a new **PROPERTIES** command in the Chaosnet file protocol, so it may not work with servers running old software.

3.27.7 Creating Links

fs:create-link *link-name link-to* &key (*error* **t**) — Function

Creates a link named *link-name* that points to a file named *link-to*. An error happens if the host specified in *link-name* does not support links (or because of any of the usual problems that can happen in creating a file).

3.27.8 New :SUBMIT option for opening files

A new **:submit** option available in **open** and other constructs that use **open** (such as **with-open-file** and friends). When this option is **t** and the direction is **:output**, the file

is submitted for batch processing on the host. The **:submit** option is currently effective on VMS and Twenex Chaosnet FILE servers.
An example:

```
(defun retrieve-twenex-file (F)
  "Submit a batch job that will retrieve the file F"
  (setq f (fs:parse-pathname F))
  (with-open-file (s (send (send (send (send f :host)
                                          :sample-pathname)
                                    :homedir)
                              :new-pathname :name "_retrieve"
                              :type "ctl" :version :newest)
                  :direction :output
           :temporary t :submit t)
          (format s "@retrieve ~A~%" (send f :string-for-host))))
```

3.27.9 File-Reading Special Forms

These two special forms are a straightforward aid in writing code that reads Lisp forms from a file, while obeying the attribute list of the file. (The attribute list file's pathname object or generic pathname object is *not* updated with this special form.)

fs:reading-from-file (*form file*) *body...* Macro
fs:reading-from-file-case (*form file error*) *clauses...* Macro

The following form prints out the result of evaluating each form in the file:

```
(fs:reading-from-file (form file)
   (format t "Values from ~S are: " form)
   (format:print-list t "~S" (multiple-value-list (eval form))))
```

The body of the form is executed for every form in the file. **fs:reading-from-file-case** is a cross between **fs:reading-from-file** and **with-open-file-case**, except there's an additional argument *error* argument (which is bound to the error object) for use in the clauses.

```
(fs:reading-from-file-case (form file error)
    ((fs:file-not-found
       (format t "~&Options file ~A not found, using default values."
                 file))
         (fs:file-error
           (format t "~&Error: ~A" (send error :report-string)))
    (:no-error (process-option form))))
```

Here, the **:no-error** clause, which must be present and consists of any number of forms, is executed for every form in the file. (But any error clause would be executed just once.) The value of the error variable is not defined in the **:no-error** clause

The following function does the actual work of getting the attribute list of a stream.

fs:extract-attribute-bindings *stream* Function

returns two values: a list of variables, and a corresponding list of values to bind them to, to set up an environment to read data from *stream* in accordance with *stream*'s attribute list.

3.27.10 VMS Default Device

The "primary device" for VMS hosts now defaults **USRD$** rather than**SYS$SYSDSK**. However, it is also possible specify what the default device using the **:host-default-device-alist** option in the site description; see section 9.3.2.2, page 85 for more details.

3.27.11 Improved File Error Handling

When there is an error accessing a file and the system asks for a new pathname, you now have the option of entering the debugger instead. Simply type **End**.

3.28 String Changes

string-length can give you the length of anything **string** can coerce into a string. In Release 2, it would not accept characters or symbols.

make-string *length* &key *initial-element* &allow-other-keys Function

The "other" keyword which is the most interesting to use here is **:fill-pointer**. (Only the **:initial-element** keyword is supported in COMMON LISP.)

Clarification: note that **string-append** and the related functions do not create strings with fill pointers.

3.29 New Keyword Arguments to MAKE-PLANE

The arguments *initial-dimensions* and *initial-origins* are now accepted. You can use them to specify which part of the infinite plane the initially allocated storage should be for.

make-plane *rank* &key *type default-value extension initial-dimensions initial-origins* Function

Creates and returns a plane. *rank* is the number of dimensions. The keyword arguments are

type The array type symbol (e.g. **art-1b**) specifying the type of the array out of which the plane is made.

default-value The default component value as explained above.

extension The amount by which to extend the plane, as explained above.

initial-dimensions **nil** or a list of integers whose length is *rank*. If not **nil**, each element corresponds to one dimension, specifying the width to allocate the array initially in that dimension.

initial-origins **nil** or a list of integers whose length is *rank*. If not **nil**, each element corresponds to one dimension, specifying the smallest index in that dimension for which storage should initially be allocated.

Example:

```
(make-plane 2 :type 'art-4b :default-value 3)
```

creates a two-dimensional plane of type **art-4b**, with default value **3**.

3.30 New Resource Features

A new option to **defresource** called **:deinitializer** has been added. The value is either a function of one argument, or a form containing a reference to the variable **object**. The deinitializer is called when an object is deallocated. There are two storage-related reasons for specifying a deinitializer:

1. Sometimes, a resource may have pointers to objects that are only valid (with respect to the Lisp Machine storage conventions) when the object is allocated. When the object is deallocated, some objects to which it might point may no longer be around. This situation only arises when using dangerous features such as pointer-making subprimitives or temporary areas.

   Even when an object of a resource is deallocated, the garbage collector can still find it. Thus, "dangerous" pointers should be thrown away by the deinitializer.
2. An object of a resource might the only object to point to another big object that should otherwise be freed by the garbage collector.

In either case, the deinitializer will deference the objects to which it points by setting slots of itself to **nil**.

There are also two new operations on resources:

map-resource *function resource-name* &rest *extra-args* Function

Operates with function on each object created in resource resource-name.

Each time function is called, it receives three fixed args, plus the extra-args. The three fixed args are:

- an object of the resource;
- **t** if the object is currently allocated ("in use")
- the resource data structure itself.

deallocate-whole-resource *resource-name* Function

Deallocates each object in resource *resource-name*. This is equivalent to doing **deallocate-resource** on each one individually. This function is often useful in warm-boot initializations.

3.31 Flushed Processes

A flushed process now has the symbol **si:flushed-process** as its wait function. This function is equivalent to **false** in that it always returns **nil**, but it is distinguishable from **false**. Thus, flushed processes can reliably be distinguished from those that have done **process-wait-forever**.

3.32 Indenting Format Directive

format output within a ~→ ... ~← construct is printed with each line indented to match the indentation that was current when the ~→ was reached.

3.33 Input Read Function Changes

3.33.1 READLINE and Friends

readline and **readline-trim** have been extended to return a second value. This value is **t** if end-of-file was encountered.

Note that end-of-file can still be an error if encountered at the beginning of the line, and this is still controlled by the *eof-option* argument. But if the function does return, the second argument always says whether there was an end-of-file.

The new function **readline-or-nil** is like **readline-trim** except that it returns **nil** rather than "" if the input line is empty or all blank.

3.33.2 New Function READ-DELIMITED-STRING

read-delimited-string &optional *delimiter stream eof rubout-handler-options buffer-size* Function

Reads input from *stream* until a delimiter character is reached, then returns all the input before but not including the *delimiter* as a string. *delimiter* is either a character or a list of characters that all serve as delimiters. It defaults to the character **End**. *stream* defaults to the value of ***standard-input***.

If eof is non-**nil**, then end-of-file on attempting to read the first character is an error. Otherwise it just causes an empty string to be returned. End-of-file once at least one character has been read is never an error but it does cause the function to return all the input so far.

Input is done using rubout handling and echoing if stream supports the **:rubout-handler** operation. In this case, *rubout-handler-options* are passed as the options argument to that operation.

buffer-size specifies the size of the string buffer to allocate initially.

Three values are returned:

- the string of input read;
- a flag which is **t** if input ended due to end of file;
- and the delimiter character which terminated input (or **nil** if end of file was reached).

:run-time-alternatives can also be called **:mixture**, for compatibility with other systems.

3.33.3 :STRING-LINE-IN Stream Operation

:string-line-in is a new standard input stream operation, supported by all the input streams provided by the system. It fills a user-supplied buffer with text from the stream until either the buffer is full, end of file is reached, or a **Return** is found in the input. If input stops due to a **Return**, the **Return** itself is not put in the buffer.

Thus, this operation is nearly the same as **:string-in**, except that **:string-in** always keeps going until the buffer is full or until end of file.

:string-line-in returns three values:

- The index in the buffer at which filling stopped. (If the buffer has a fill pointer, it is set to this value as well.)
- **t** if end of file was reached.
- **t** if the line is not complete; that is, input did not encounter a **Return** character. In that case, there may be more text in the file belonging to the same line.

3.33.4 PROMPT-AND-READ Improvements

There are several new options you can give to **prompt-and-read**, and some existing options now take arguments. Remember that the first argument to **prompt-and-read** is an option that is either a keyword or a list of a keyword followed by arguments (alternating keywords and values). The rest of the arguments are a string and additional args passed to **format** to print the prompt.

Here are the options which have been changed incompatibly:

:eval-form-or-end
: Is changed so that, if the user types just **End**, it returns **:end** as the second value. It used to return **#\end** as the second value in that case. The first value will still be **nil**.

:eval-form-or-end :default *object*

:eval-form :default *object*
: If the user types **Space**, meaning use the default, the second value will now be **:default** rather than **#\Space**. The first value will still be *object*, the default.

Here are the options that now take additional arguments:

:pathname :defaults *default-list* **:version** *default-version*
: A pathname is read, and returned using **fs:merge-pathname-defaults**: *default-list* is passed as the second argument, and *default-version* is passed as the fourth argument.

:number :input-radix *radix* **:or-nil** *nil-ok-flag*
: Reads a string terminated by **Return** or **End**, and parses it into a number using radix *radix* if the number is a rational. The number is returned. If *nil-ok-flag* is non-**nil**, then you may also type just **Return** or **End**, and **nil** is returned.

Here are the new options:

:character Reads a single character and return a fixnum representing it.

:date :never-p *never-ok* **:past-p** *past-required*
: Reads a string terminated by Return or End and parses it as a date/time. The universal time number representing that date/time is returned. If past-required is non-**nil**, the date must be before the present time, or else you get an error and must rub out and use a different date. If never-ok is non-**nil**, then you may also type "never"; **nil** is returned.

:expression
: Is the same as **:read**: read a LISP object using **read** and return it.

:expression-or-end
: Reads a LISP object using **read**, but alternately allows just **End** to be typed and returns the two values **nil** and **:end**.

:pathname-or-nil
: Reads a file name and returns a pathname object, but if the user types just End then returns **nil** instead. The pathname is defaulted with **fs:merge-pathname-defaults**.

:pathname-or-nil :defaults *default-list* **:version** *default-version*
: A pathname is read, and returned using **fs:merge-pathname-defaults**: *default-list* is passed as the second argument, and *default-version* is passed as the fourth argument.

3.33.5 The Rubout Handler

There are some new options for use in controlling the rubout handler; some other options are changed. The new options are **:no-input-save, :activation, :command** and **:preemptable**. The changed options are **:do-not-echo, :pass-through** and **:prompt.**

Recall that the options are the first argument to the **:rubout-handler** stream operation; the remaining arguments being the parsing function and arguments to call it with. The options argument is an alist; each element should look like one of these patterns:

(:no-input-save t)
: Does not save this batch of input in the input history when it is done. **yes-or-no-p** specifies this option.

(:full-rubout *value***)**
: Causes immediate return from the **:rubout-handler** operation if the buffer ever becomes empty due to deletion of text.

  Two values are returned: **nil** and value.

  The debugger uses this option so it can erase **"Eval:"** from the screen if you rub out all the characters of the form to be evaluated.

(:initial-input *string***)**
: Starts the buffer with *string*.

(:initial-input-pointer *n***)**
: Starts by placing cursor *n* chars from the beginning of the buffer. This is used with **:initial-input.**

(:activation *fn args...***)**
: Activates if certain characters are typed in. An activation character causes the buffered input to be read immediately, and moves the editing pointer to the end of the buffer.

  fn is used to test whether characters are activators. It is called with an input character (never a blip) as the first arg and *args* as additional args. If fn returns non-**nil**, the character is an activator.

  The activation character does not go in the buffer itself. However, after the parsing function has read the entire contents of the buffer, it reads a blip (**:activation** *char numeric-arg*) where *char* is the character that activated and *numeric-arg* is the numeric argument that was pending for the next rubout handler command.

(:do-not-echo *chars...***)**
: Poor man's activation characters. Like **:activation** except: the characters that should activate are listed explicitly, and the character itself is returned, rather than a blip, after all the buffered input.

(:command *fn args...***)**
: Makes certain characters preemptive commands. A preemptive command returns instantly to the caller, of the **:rubout-handler** operation, regardless of the input in the buffer. It returns two values: a list (**:command** *char numeric-arg*) and the keyword **:command**. Any buffered input remains in the buffer for the next time input is done. In the meantime, the preemptive command character can be processed by the command loop.

  In testing for whether a character should be a preemptive command, this works just like **:activation.**

(:preemptable *value***)**
: Makes all blips act as preemptive commands. If this option is specified, the rubout handler returns immediately when it reads a blip, leaving buffered input for next time.

Two values are returned: the blip that was read, and value.]

(:pass-through (*char doc*) ...**)**

Defines editing commands to be executed by the parsing function itself. Each *char* is such a command, and *doc* says what it does. *doc* is printed out by the rubout handler's help command. If any of these characters is read by the rubout handler, it is returned immediately to the parsing function regardless of where the input pointer is in the buffer. The parsing function should not regard the character as part of the input.

There are two reasonable things that the parsing function can do:

- print some output
- **:force-kbd-input**

If output is printed, the **:refresh-rubout-handler** operation should be invoked afterward. This causes the rubout handler to redisplay so that the input being edited appears after the output that was done. If input is forced, it will be interpreted as rubout handler commands.

There is no way to act directly on the buffered input because different implementations of the rubout handler store it in different ways.

(:prompt *fn-or-string***)**

Directs prompting for the input being read. If *fn-or-string* is a string, it is printed; otherwise it is called with two args: the stream, and a character that is an editing command that says why the prompt is being printed.

(:reprompt *fn-or-string***)**

Same as **:prompt** except used only if the input is reprinted for some reason after editing has begun. The **:reprompt** option is not used on initial entry. If both **:prompt** and **:reprompt** are specified, **:prompt** is used on initial entry and **:reprompt** thereafter.

A new convenient way to invoke the rubout handler on a stream if the stream supports it is to use **with-input-editing**.

with-input-editing (*stream options*) *body...* Macro

Invokes the rubout handler on stream, if it is supported, and then executes body. body is executed in any case, within the rubout handler if possible. body's values are returned by **with-input-editing**. However, if a preemptive command is read, **with-input-editing** returns immediately with the values being as specified above under **:command** or **:preemptable**.

options are used as the rubout handler options.

sys:parse-error Condition

sys:read-error (parse-error) Condition

All rubout handlers now check for the condition name **sys:parse-error** when they decide whether to handle an error. They used to check for **sys:read-error**. All the errors signaled by the system that have the condition name **sys:read-error** now have **sys:parse-error** as well, so no change in behavior should be apparent. However, you can signal an error that has **sys:parse-error** but not **sys:read-error** if you wish (say, if the error happens in some function other than **read**).

sys:parse-error is also the condition name that the compiler looks for in its efforts to continue from errors that happen while reading text to be compiled.

sys:parse-ferror *format-string* &rest *format-args* Function

The function **sys:parse-ferror** is a convenient way to signal such an error, if you do not want any additional condition names besides **sys:parse-error** and the ones it implies. If **sys:parse-ferror** is called while reading text to be compiled, it will return **nil** automatically.

3.34 Readtables

Because of the adoption of COMMON LISP, some of the Lisp reader syntax internals have been changed or extended. In addition, a mechanism has been added for named readtables, which may be helpful in more easily supporting languages with different syntaxes from ZETALISP

3.34.1 Syntax Descriptions

Remember, that even though the following changes have been documented as a result of COMMON LISP, the syntax descriptions and the way there are modified are not accessed exactly this way in COMMON LISP itself.

si:set-syntax-from-description *char description* &optional *readtable* Function

There are new syntax descriptions that you can pass to this function:

si:escape A quote-one-character character. In the ZETALISP readtable / is such a character. In the COMMON LISP readtable, \ has this syntax description.

si:multiple-escape
A quote-several-characters character. In the ZETALISP readtable | is such a character.

si:character-code-escape
Is followed by a character's octal code. In the ZETALISP readtable ⊗ is such a character.

si:digitscale
A character for shifting an integer by digits. In the ZETALISP readtable ^ is such a character.

si:bitscale A character for shifting an integer by bits. In the ZETALISP readtable _ is such a character.

si:non-terminating-macro
A macro character that is not recognized if a token is already in progress. In the ZETALISP readtable # is such a character. (It is also a dispatching macro, but that is another matter.) The correct way to make a character be a macro is with **set-macro-character**, not with this description.

The syntax descriptions **si:slash** and **si:circlecross** are still implemented but it is preferable to use **si:escape** or **si:character-code-escape**. The syntax **si:verticalbar** is no longer defined; use **si:multiple-escape**. Unfortunately, it is no longer possible to define **si:doublequote**, since Doublequote (") is now just a macro character.

3.34.2 Named Readtables

To aid in the support for COMMON LISP in Zetalisp, readtables were given names so that they could be referred to symbolically. The **readtable** and **syntax** file attributes use this feature to distinguish COMMON LISP files from Zetalisp files. Named readtables may be useful for similar applications. (Note that for a readtable to be accessible from the file attribute list, one of its names must readable as a symbol - so it should have one short name with no whitespace in it.)

There are two ways to get a named readtable:

- The first way is to use the readtable compiler, to make a readtable from scratch. In the section of the readtable definition file where the options (**:opts**) go, use the **:names** option.

```
(:OPT :NAMES '("Lisp Machine COBOL" "COBOL"))
```

- The second way is to copy another readtable, and give it some names. You can actually override another readtable's name by pushing your readtable in front of it on **si:*all-readtables***, so be careful about this feature, which may or may not always be the right thing for an application.

```
(defvar *strange-table* :unbound
  "For slightly modified Common Lisp syntax")

(defun set-up-strangeness ()
  (let ((rt (copy-readtable nil)))
    (setf (si:rdtbl-names rt) '("strange Common Lisp" "STRANGE"))
    (push rt si:*all-readtables*)
    (setq *strange-table* rt)))
```

The COMMON LISP readtable has, among other names, **CL** and **Common-Lisp** for nicknames. The standard Zetalisp readtable can be found with the names **T**, **Traditional**, **ZL**, and **Zetalisp**.

si:find-readtable-named *name create-p* Function

Find or possibly create a readtable named *name* If there is a readtable which has a name **string-equal** to it, we return that readtable. Otherwise, we may create such a readtable, depending on *create-p*

nil

:error Get an error.

:find Return **nil**

:ask Ask whether to create a readtable named *name* which is a copy of the current readtable (***readtable***), and returns it if so.

t Create the readtable (a copy of ***readtable***) and return it.

si:*all-readtables* Variable

This is a list of all readtables *except* those created with **copy-readtable**, which does not automatically put new readtables on this list.

si:rdtbl-names *readtable* Function

The accessor for the names (strings) of the readtable, the first name being the one printed out at the beginning of Lisp interaction loops. The rather constrained name of the function is due to historical reasons.

3.35 Fasdumping Functions Record Package

These functions:

- **dump-forms-to-file**
- **compiler:fasd-symbol-value**
- **compiler:fasd-font**
- and **compiler:fasd-file-symbols-properties**

now always record, in the QFASL file created, the name of the package in which the file was written. This makes sure that the symbols used when the file is loaded will be the same as when it was dumped.

In **dump-forms-to-file**, you can specify the package to use by including a **:package** attribute in the **attribute-list** argument. For example, if that argument is the list **(:package :si)** then the file is dumped and loaded in the **si** package. If you do not specify a package, the file is dumped and loaded in **user**. With the other three functions, the file is always dumped and loaded in **user**.

3.36 Process Queues

A process queue is a kind of lock, that can record several processes that are waiting for the lock and grant them the lock in the order that they requested it. The queue has a fixed size. If the number of processes waiting remains less than that size, then they will all get the lock in the order of requests. If too many processes are waiting, then the order of requesting is not remembered for the extra ones.

si:make-process-queue *name size* Function

Makes a process queue object named name, able to record size processes. size includes the process that owns the lock.

si:process-enqueue *process-queue* &optional *lock-value who-state* Function

Attempts to lock *process-queue* on behalf of *lock-value*. If *lock-value* is **nil** then the locking is done on behalf of **current-process**.

If the queue is locked, then *lock-value* or the current process is put on the queue. Then this function waits for that lock value to reach the front of the queue. When it does so, the lock has been granted, and this function returns.

who-state appears in the who line during the wait. It defaults to **"Lock"**.

si:process-deqeueue *process-queue* &optional *lock-value* Function

Unlocks process-queue. *lock-value* (that defaults to the current process) must be the value that now owns the lock on the queue, or an error occurs. The next process or other object on the queue is granted the lock and its call to **si:process-enqueue** will therefore return.

si:reset-process-queue *process-queue* Function

Unlocks the queue and clears out the list of things waiting to lock it.

si:process-queue-locker *process-queue* Function

Returns the object in whose name the queue is currently locked, or **nil** if it is not now locked.

3.37 New Function SI:PATCH-LOADED-P

si:patch-loaded-p *major-version minor-version* &optional (*system-name* **"SYSTEM"**) Function

Returns **t** if the changes in patch number *major-version.minor-version* of system *system-name* are loaded. If major-version is the major version of the system currently loaded, then the changes in that patch are loaded if the current minor version is greater than or equal to *minor-version*. If the currently loaded major version is greater than major-version, then it is assumed that the newer system version contains all the improvements patched into earlier versions, so the value is **t**.

3.38 Date Formats

time:*default-date-print-mode* Variable

This defines the default way to print the date for functions in the **time** package that accept a print-mode argument, which currently include:

* **time:print-time**
* **time:print-universal-time**
* **time:print-brief-universal-time**
* **time:print-date**
* **time:print-universal-date**
* **time:print-current-time**
* **time:print-current-date**

Following is a description of the possible values, using ZETALISP syntax.

:dd//mm//yy
Prints out as 27/10{/66}

:dd//mm//yyyy
27/10{/1966}

:mm//dd//yy
10/27{/66}

:mm//dd//yyyy
10/27{/1966}

:dd-mm-yy 27-10{-66}

:dd-mm-yyyy
27-10{-1966}

:dd-mmm-yy
27-Oct{-66}

:dd-mmm-yyyy
27-Oct{-1966}

:dd/ mmm/ yy
27 Oct{ 66} – Note that the print name of this symbol really does contain a space; backslash would be used to enter the symbol in COMMON LISP syntax.

:dd/ mmm/ yyyy
27 Oct{ 1966}

:ddmmmyy
27Oct{66}

:ddmmmyyyy
27Oct{1966}

:yymmdd 661027

:yyyymmdd
19661027

:yymmmdd
{66}Oct27

:yyyymmmdd
{1966}Oct27

:yy-mmm-dd
{66-}Oct-27

:yyyy-mmm-dd
{1966-}Oct-27

:yy-mm-dd {66-}10-27

:yyyy-mm-dd
{1966-}10-27

These last four, and all the **yyyy** ones are new since the manual.
The default value is **:mm//dd//yy**. If one wishes to customize this for a site (usually, a site not in the United States), simply put a **setq** of **time:*default-date-print-mode*** to the appropriate value something in the **SYS: SITE: SITE LISP** file.

The time parser now accepts ISO format dates. **1980-3-15** means 15 March, 1980; **1980-MAR-15** means 15 March, 1980.

3.39 Network Changes

Some of the site changes (section 9.3.2.2, page 85) are also network-related.

3.39.1 Host Network Operations

si:parse-host *string error-p* (*unknown-ok t*) Function

si:parse-host's third argument, *unknown-ok*, now defaults to **t**. That means that if it can't find the host on **si:host-alist**, it tries contacting a host table server to see if *it* knows about the host. If the server contact does not, an error is signalled (or **nil** is returned) as usual. The change was made to minimise the penalty for not loading the latest site files. (Maintaining up-to-date site information can be a problem at large installations.)

The list of hosts that may be contacted on the Chaosnet for this service are listed in the site option **:chaos-host-table-server-hosts**.

:network-addresses Operation on **si:host**

The operation **:network-addresses**, on a host object, returns an alternating list of network names and lists of addresses, such as

```
(:chaos (3104) :arpa (106357002))
```

You can therefore find out all networks a host is known to be on, using **getf**.

:network-address *network* &optional *smart-p* Operation on **si:host**

Returns a network address, if possible, for the host on *network*. The network address returned is the primary one (determined ultimately by the order found in the host table source) unless *smart-p* is non-**nil**; then, some optimal address as defined by *network* is returned.

The actual format of the network address is left unspecified; it is usually the "unparsed" form which is passed to the network entry point functions.

:unparsed-network-address *network* &optional *smart-p* Operation on **si:host**

Like **:network-address**, but returns an unparsed network address (a string), where the string representation is defined by the network.

:internet-connect *socket protocol* &key *timeout* (*ascii-translation* **t**) (*direction* **:bidirectional**) Operation on **si:host**

This is the current interface for using Internet in LMI ZETALISP, to connect to the host at *socket* using *protocol*, a keyword. Currently, the only legal value is **:tcp**. *timeout*, in sixtieths of a second, currently defaults to some reasonable value. The remaining keyword arguments are only applicable when the Internet protocol requested is **:tcp**. *direction* can be one of symbols acceptable to **chaos:open-stream**: **:input**, **:output**, or **:bidirectional**. Currently, *ascii-translation* defaults to **t**, since most TCP servers and protocols are oriented to the ASCII character set.

3.39.2 New Error Condition SYS:NO-SERVER-UP

sys:no-server-up (**sys:connection-error**) Condition

The error condition **sys:no-server-up** is signalled by certain requests for a service from any available network host, when no suitable host is currently available.

3.39.3 Some Chaosnet Functions Renamed

Some functions in the **chaos** package have had their names changed. This is so we can avoid having two advertised system functions with the same name in different packages. The old names still work.

Old Name	New Name
chaos:finish	chaos:finish-conn
chaos:close	chaos:close-conn
chaos:finished-p	chaos:conn-finished-p

3.39.4 Chaosnet Listening Streams

Now you can listen for a Chaosnet connection and open a stream at the same time. To do this, call **chaos:open-stream** with **nil** as the host argument. You must still pass a non-**nil** contact-name argument. The function will return a stream to you as soon as someone attempts to connect to that contact name.

At this time, you must accept or reject the connection by invoking the stream operation **:accept** or **:reject**. **:reject** takes one argument, a string to send back as the reason for rejection. Before you decide that to do, you can use the **:foreign-host** operation to find out where the connection came from.

3.39.5 New Chaos Routing Inspector Functions

These two functions make use of the **DUMP-ROUTING-TABLE** protocol, documented in the new edition of the *Lisp Machine Manual*. They are primarily for inspecting the operation of the network and the localisation of bridging and routing problems.

chaos:show-routing-table *host* &optional (*stream* ***standard-output***) Function

Prints out the routing table of *host* onto *stream*.

chaos:show-routing-path &key (*from* **si:local-host**) *to* (*stream* ***standard-output***) Function

Shows how packets will flow from *from* to *to*, using the routing information supplied by *from* and any intervening bridges to figure out the path.

For example, **(chaos:show-routing-path :from "charon" :to "nu-1")** may produce the following output:

```
MIT-CHARON will bounce the packet off MIT-SIPB-11 at cost 81.
MIT-SIPB-11 will bounce the packet off MIT-INFINITE at cost 63.
MIT-INFINITE will bounce the packet off MIT-BYPASS at cost 51.
MIT-BYPASS will bounce the packet off MIT-OZ-11 at cost 37.
MIT-OZ-11 will bounce the packet off XI (XX-Network-11) at cost 23.
Direct path from XI (XX-Network-11) to host MIT-NU-1 on subnet 32 at
interface 1.
```

3.40 Infix Expressions.

You can now include infix expressions in your Lisp code. For example,

```
#◇X:Y+CAR(A1[I,J])◇
```

The ◇ character is **Altmode.**

is equivalent to

```
(setq x (+ y (car (aref a1 i j))))
```

#◇ begins an infix expression, and ◇ ends it.

The atomic terms of infix expressions include

- symbols: use " to quote special characters.
- numbers: any valid Lisp real or imaginary number is accepted. Complex numbers can be constructed by addition or subtraction.
- strings: the same as in ordinary Lisp syntax.
- raw Lisp data: ! followed by any Lisp expression, as in

```
#◇ FOO . !(CAR BAR) ◇ => (list* foo (car bar))
```

Combining operations:

```
Highest precedence
   a[i]          (AREF a i)
   a[i,j]        (AREF a i j)   and so on
  examples
    X[I,J+3]    =>    (AREF X (+ J 3))
    (GET-MY-ARRAY(FOO))[I]    =>    (AREF (GET-MY-ARRAY FOO) I)

    f(a)          (f a)
    f(a,b)        (f a b)     and so on
  examples
    CAR(X)    =>    (CAR X)
```

```
    (exp)        exp     parentheses control order of evaluation
  examples
    (X+1)*Y   =>   (* (+ X 1) Y)

    (e1, e2)         (PROGN e1 e2)      and so on
  examples
    (X:5, X*X)   =>   (PROGN (SETQ X 5) (* X X))

    [elt]          (LIST elt)
    [e1,e2]        (LIST e1 e2)     and so on
  examples
    [!'X,Y,Z]    =>    (LIST 'X Y Z)

Precedence 180 on left, 20 on right
    a : b               (SETF a b)
  examples
    X: 1 + Y: Z+5    =>    (SETQ X (+ 1 (SETQ Y (+ Z 5))))

Precedence 140
    a^b                (EXPT a b)     right associative
  examples
    X ^ N ^ 2    =>    (EXPT X (EXPT N 2))

Precedence 120
    a* *b                 (* a b)
    a* *b * *c         (* a b c)     and so on
    a/ *b                 (// a b)
    a/ *b / *c         (// a b c)     and so on

Precedence 100
    - a                   (- a)
    a+ *b                 (+ a b)
    a+ *b + *c         (+ a b c)    and so on
    a- *b                 (- a b)
    a- *b - c          (- a b c)       and so on

Precedence 95
    a. b                  (LIST* a b)
    a. b . c            (LIST* a b c)      and so on
    a@ b                  (APPEND a b)
    a@ b  c            (APPEND a b c)    and so on

Precedence 80
    a ∊ b                 (MEMQ a b)
    a= b                  (= a b)
    a= b = c            (= a b c)          and so on
    <, >, ≠, ≥, ≤ are like =.

Precedence 70
```

```
    NOT a                  (NOT a)

Precedence 60
    a AND b                 (AND a b)
    a AND b AND c           (AND a b c)      and so on

Precedence 50
    a OR b                    (OR a b)
    a OR b OR c               (OR a b c)     and so on

Precedence 45 for c, 25 for a and b.
    IF c THEN a                (IF c a)
    IF c THEN a ELSE b        (IF c a b)
```

It is easy to define new operators. See **SYS: IO1: INFIX LISP.**

3.41 Bug Reports for User Systems

To make it easier to collect bug reports about a system, there is now a **:bug-reports** option to **defsystem.** Two values are supplied: the name of the topic, and a documentation string. The topic name is usually the name of the system. The documentation string appears in the mouse documentation line when the user sends a bug report from ZMail. For example:

```
(defsystem foo
  ...
  (:bug-reports "FOO" "Tell about a bug in the FOO system")
  ...
  )
```

For this to really work, there must be a mailing address named **bug-foo** on the bug report host (that is, the host named by the site option **:host-for-bug-reports**).

This feature does not work with the **Control-M** debugger command, because the error handler presets the address according to the value of the string returned by sending the **:bug-report-recipient-system** message to the error instance.

4. DEFSTRUCT

This describes changes to the **defstruct** feature as implemented in Release 2.

The compatible changes to **defstruct** as discussed in this section of the manual are:

- New Options
- Documentation for Structures
- Slot Options
- Changes to the **:include** option
- **defstruct** Tries to Determine an Appropriate Array Type
- New Predefined Structure Types
- COMMON LISP Support

One change is that **defstruct** no longer generates any sort of **eval-when**. If you want the expansion of a **defstruct** to be inside an **eval-when**, simply write an **eval-when** around the **defstruct**.

4.1 New Options

The following are now accepted by **defstruct** in addition to the options described in the *Lisp Machine Manual.*

:callable-constructors

Giving this option a value of **t** (i.e. by writing **(:callable-constructors t)**) causes constructors for this structure to be functions, rather than macros, as they used to be. This, however, means that code like the following, which works with a macro-defined constructor, will usually cause an error if it is a function:

```
(make-foo a 1 b 'bee)
```

The syntax to use for callable constructors is like that for **&key** functions (which is actually how they are defined):

```
(make-foo :a 1 :b 'bee).
```

Macro-defined constructors now accept keywords for slot-names also. Just to facilitate changing the kind of constructor you use, it is probably best to always use this syntax. However, an irresolvable incompatibility exists in the way the two types of constructors handle the constructor options such as *:times* and **:make-array**. When **:callable-constructors** is **nil**, they *should not* be quoted, and when it is **t**, they *must* be quoted. For example, in the first case we would say:

```
(make-frobboz :slot-1 'foo :make-array (:leader-length 2))
```

With callable constructors the **:make-array** argument must be quoted:

```
(make-frobboz :slot-1 'foo :make-array '(:leader-length 2))
```

:subtype

This option is valid only when used with structure-types that include **:subtype** among their **:defstruct-keyword** keywords (see below). Such types include things like **:array** and **:array-leader**, for which a subtype of the primary array-type is a meaningful concept. In the case of arrays, this could be used to make a structure of this type use a specific array-type, rather than the default **art-q**. The subtype can also be implicitly specified through the **:type** option. Types such as **:list** or **:fixnum-array** do not have any any meaningful subtypes, and hence do not support the **:subtype** option. It is an error to use **:subtype** with such types.

:type This is by no means a new option, but its syntax has been extended. Previously, this option could be used only in the form (**:type** *defstruct-type*). It is now possible to write (**:type** (*defstruct-type subtype*)), the effect being like specifying both (**:type** *defstruct-type*) and (**:subtype** *subtype*). For example:

```
(defstruct (foo (:type (:array ART-4B))) A B)
  or
(defstruct (foo (:type (:vector (mod 16)))) a b)
```

using a COMMON LISP type defines a structure with two slots, each of which can contain only fixnums in the range [0,15]. This is a COMMON LISP change, but is worthwhile to use in any case as this syntax is more transparent and cleaner than the present technique of writing:

```
(defstruct (foo (:type :array) (:make-array (:type art-4b))) a b)
```

:print-function

The argument to this option is a function of three arguments, which will print an object of the type being defined. This function will be called with three arguments – the structure to be printed, the stream to print it on, and the current printing depth (which should be compared with ***print-level***). The function is expected to observe the values of the various printer-control variables. Example:

```
(defstruct (bar :named
  (:print-function
    (lambda (struct stream depth)
      (format stream "#<This is a BAR, with ring-ding index ~S>"
      (zap struct)))))
    "The famous bar structure with no known use."
    (zap 'yow) random-slot)

(MAKE-BAR) => #<This is a BAR, with ring-ding index YOW>
```

This option is similar in application to the existing option **:print**. Its introduction is a COMMON LISP change.

4.2 Documentation for Structures

defstruct now interprets a string occurring after the structure name and options as documentation for this structure. The documentation can be accessed by:

`(documentation` *structure-name* `'structure)`

and changed by **setf**ing such a form.

4.3 Slot Options

Slots within a structure may now include one or more slot options. The extended syntax for defining slots is either:

```
slot-name

or

(slot-name (default-init
                (slot-option-1 option-value-1
                 slot-option-2 option-value-2 ...)))

or

((slot-name-1 byte-spec-1 (default-init-1
                              (slot-option-1-1 option-value-1-1 ...)))
 (slot-name-2 byte-spec-2 (default-init-1
                              (slot-option-2-1 option-value-2-1 ...)))
 ...)
```

Here are the currently defined *slot-options*:

:read-only *flag*
: Specifies that this slot mat not be **setfed** if *flag* is non-**nil**. The contents of this slot are not supposed to be changed after you construct the structure.

:type *type*
: Declares that this slot is expected to be of a given type. The LISP machine compiler does not use this for any assumptions, but sometimes the information enables **defstruct** to deduce that it can pack the structure into less space by using a numeric array type.

:documentation *documentation-string*
: Makes *documentation-string* the documentation for the slot's accessor function. It also goes in the **defstruct-slot-description-documentation** for this slot. Example:

```
(defstruct (eggsample :named :conc-name
             (:print-function #'(lambda (s stream ignore)
       (format stream "#<Eggsample ~S ~S ~s>"
(eggsample-yolk s)
(eggsample-grade s)
(eggsample-albumen s)))))
     (yolk 'a :type symbol :documentation "First thing you need in
     an eggsample.")
     (grade 3 :type (mod 4))
     (albumen nil :read-only t))
=> eggsample
(setq egg (make-eggsample :albumen 'white))
=> #<Eggsample A 3 WHITE>
(setf (eggsample-yolk <c-sh-d>
   EGGSAMPLE-YOLK: (EGGSAMPLE)
   "First things you need in an eggsample."
(setf (eggsample-yolk egg) 19.5)
=> 19.5      ; no type checking !
egg
=> #<Eggsample 19.5 3 WHITE>
(setf (eggsample-albumen egg) 'eggsistential)
=> >>ERROR: SETF is explicitly forbidden on
```

```
   (EGGSAMPLE-ALBUMEN EGG)
While in the function SI::UNSETFABLE ← SI::LOCF-APPLY
   ← SI::SETF-1
...
```

4.4 Changes to the :INCLUDE Option

4.4.1 DEFSTRUCT

defstruct now accepts slot-options in the specification for included slots. This extended syntax is illustrated here:

```
(defstruct one :conc-name :named
  (slot-1 0 :type fixnum :documentation "The very first" :read-only t)
  (slot-2 'bar)
  slot-3)

(defstruct (two :conc-name :named
           (:include one (slot-1 6 :documentation "The second first")
                  (slot-3 '(a b) :type cons :read-only t)))
  (slot-3 5))
```

two will be a structure whose first slot has default value 6, has the documenation, and is read-only and of type **fixnum**, these last two attributes being inherited from the included structure. The third slot will have a default value of **'(a b)**, should be a cons, and is read-only.
The following example will cause an error:

```
(defstruct (loser :named :conc-name
                  (:include one (slot-1 0 :read-only nil :type symbol))))
```

This is because (i) the slot is specified to be not read-only, when the included slot was, and (ii) the slot was given a type that is not a subtype of the included slot type.

4.4.2 New Slot-Accessor Functions Generated

Previously no accessor called **two-slot-1** was generated in the example above, and you had to access that slot using the function **one-slot-1**. Now such accessors are generated for all the included slots, using the conc-name of the including structure. Note that the accessors need not necessarily be the same as the accessors used in the included structure. That is, they may have different documentation, or be read-only.

4.5 DEFSTRUCT Tries to Determine an Appropriate Array Type

If all the slots to **defstruct** are given **:type** slot-options and the structure is based on an array that can be of a specialised type (such as **:array**, **:typed-array**, **:grouped-array** or **:vector**) and no **:subtype** is explicitly given, then **defstruct** will attempt to find the most storage-efficient array-type (subtype) for the structure. Example:

```
(defstruct (foo)
   (eh 3 :type (mod 7))
   (be 0 :type (mod 1)))
```

will define a structure that makes arrays of type **art-4b**. This feature can be overridden by explicitly giving a **:subtype**, or by just not giving all the slot-types.

4.6 New Predefined Structure Types

The system now has a number of new predefined structure types:

:typed-array
: This is the same as **:array**, for use with **:named-typed-array**.

:named-typed-array
: This is an named array type with which you can specify a subtype restricting the type of elements. The named structure symbol is always put in leader slot 1.

:named-fixnum-array
: Named **:fixnum-array**; the named-structure-symbol is stored in the leader.

:named-flonum-array
: Named **:flonum-array**; the named-structure-symbol is stored in the leader.

:vector Same as **:typed-array**. This is used for COMMON LISP.

:named-vector
: Same as :named-typed-array. This is the default for COMMON LISP structures.

:phony-named-vector
: This is what you get in COMMON LISP if you say **(:type :vector)** and **:named**.

Examples:

```
(defstruct (foo (:type (:vector (mod 4)))) a)
(defstruct (foo (:type (:vector art-fat-string))) a)
(defstruct (bar (:type :fixnum-array) :named) x y z)
```

4.7 Common Lisp Support

There now exists a macro **cli:defstruct** to support the COMMON LISP **defstruct** feature. The only difference between **cli:defstruct** and regular **defstruct** is that the COMMON LISP version has different defaults for certain options:

:conc-name
: Defaults to **name-**, where *name* is the defstruct being defined. (Normally, it is **nil** by default.)

:predicate Defaults to **t**, producing a predicate called *name*-**p**, if no predicate name is requested by the user. (Default is normally **nil**.)

:callable-constructors
: Defaults to **t** (normally **nil**).

:alterant Defaults to **nil**, i.e. no alterant macro is defined (traditionally a macro called **alter-***name* is defined).

If you do not specify **:type**, you get **:named-vector**, which makes a named structure. You get a predicate by default. You may specify how to print the structure.

If you do specify **:type**, you never get a named structure. You either get a plain list or a plain vector. You do not get a predicate by default, and you may not request one. You may not specify how to print.

If you specify **:named** along with **:type**, you do not get a named structure. You get either type **:named-list** or type **:phony-named-vector**. Both of these types store the structure type in the structure somewhere, and both of them allow you to define a predicate that looks there to see whether an object appears to be a structure of the sort you defined. Neither type is recognizable by **typep**, and anyone randomly creating a list or vector with the right thing in it at the right place will find that it satisfies the predicate.

4.8 Changes to DEFSTRUCT-DEFINE-TYPE Options

4.8.1 New Per-Type Method of Declaring DEFSTRUCT Options

defstruct used to check whether a keyword appearing as an option was valid by checking whether the keyword had a non-**nil** **si:defstruct-description** property. The problem with this technique is that keywords that are appropriate to only one type of structure are accepted by **defstruct** as options for other structures for which they are meaningless. (For example, the **:times** option for grouped arrays has no meaning for other currently-defined structure types.) The new way to achieve this functionality is via the **:defstruct-keywords** option to **defstruct-define-type**, which has the same syntax as the old **:keywords** option, for example, (**:defstruct-keywords** *keyword-1 keyword-2* ...). A typical use is the following, which is the actual definition of the **:grouped-array** type:

```
(defstruct-define-type :grouped-array
  (:cons-keywords :make-array :times :subtype)
  (:defstruct-keywords :make-array :times :subtype)
  (:defstruct (description)
    (defstruct-hack-array-supertype description))
  (:cons (arg description etc) :alist
    (lispm-array-for-defstruct
      arg
      #'(lambda (v a i) '(aset ,v ,a ,i))
      description etc nil nil nil
      (or (cdr (or (assq :times etc)
                   (assq :times
                         (defstruct-description-property-alist))))
     1)
      nil))
  (:ref (n description index arg)
    description ; ignored
    (cond ((numberp index)
      '(aref ,arg ,(+ n index)))
          ((zerop n)
      '(aref ,arg ,index))
  (t '(aref ,arg (+ ,n ,index))))))
```

The **:cons-keywords** specifies the valid keywords that can be supplied to a constructor for this type. **:defstruct-keywords** (which happens to be the same in this case) specifies valid keywords to appear in the structure definition of a grouped array, making

```
(defstruct (foo (:type :grouped-array) :times 7) a b)
```

a valid defstruct, while

```
(defstruct (foo (:type :grouped-array) :typo 7) a b)
```

and

```
(defstruct (foo (:type :array) :times 7) a b)
```

signal an error.

The old type-independent method of saying

```
(defprop :make-array t :defstruct-option)
```

is obsolete, although still supported so that programs using this continue to work.

4.8.2 :KEYWORDS Option to Renamed :CONS-KEYWORDS

This has been done because **defstruct-define-type** now knows about more than one type of keyword relevant to the structure, namely **:cons-keywords** and **:defstruct-keywords**, which are relevant to the construction and definition respectively of structures of a given type. Previously, there were no **:defstruct-keywords**, and so there was no ambiguity in calling this option plain **:keyword**. As this is largely a change for consistency's sake, the old syntax continues to be supported.

5. The New Package System

A new package system has been created; it is essentially that of COMMON LISP with some added compatibility features. Its highlights are:

- Symbols in a package are now marked as either internal or external. Only the external symbols are inherited by other packages.
- Packages are no longer arranged in a hierarchy; inheritance is no longer required to be transitive. Now you can specify exactly which other packages' external symbols are to be inherited by a new package.
- **keyword** and **user** are now two distinct packages. No symbol is shared between **keyword** and **global**, so that **compile** and **:compile** are two distinct symbols, and so are **nil** and **:nil**. You must now be careful to use the correct symbol (keyword or global) in your code, whereas it used to make no difference.
- All package names are now global in scope; they mean the same thing regardless of which package is current. It is also possible to define local nicknames, in effect in only one package, but this is usually not done.
- Package prefixes can now contain **#**: in place of just :. They also sometimes contain two colons in a row.

These things have not changed in the new package system:

- A package is still an object used by **intern** to map names into symbols. At any time one package is current; it is the value of ***package***, and is used by default in **intern** and **read**. Packages can still have their own ("local") symbols while inheriting additional symbols from other packages.
- **read** still looks up symbols in the current package by default. It still allows you to specify another package with a package prefix, a package name followed by a colon, as in **si:full-gc**.
- There is still a package called **global**, which contains the fundamental function and variable symbols of LISP, such as **eval**, **cond**, **setq**, **t** and **package**. By default, new packages inherit from this package alone.
- There is still a keyword package whose symbols are normally referred to with a package prefix that is just a colon, as in **:noselective**.
- Nearly all the old documented functions for operating on packages still work, though not always exactly in the same way.

5.1 Specific Incompatibilities

Here are the specific incompatibilities between the old and new package systems.

- **list** and **:list** are now two distinct symbols. No symbol is now shared between the **global** package and the **keyword** package. This means that in many cases where a colon prefix used to make no difference, it is now significant. You must be careful to use package prefixes when you want the keyword symbol. The documentation has made the distinction even when it did not matter. If you are lucky, you followed the documentation as if you did not realize that **list** and **:list** were the same symbol, and your old code will still work.
- Files loaded into the **user** package will not work if they omit the colon on keyword symbols, as they were formerly allowed to do. See the section "The USER Package", below, for more information. With luck, these problems will be infrequent.

- **pkg-subpackages** no longer exists. There is no way to simulate the old meaning of this function, since there is no equivalent of "subpackages" close enough to the old concept.
- **pkg-super-package** does still exist, but it uses a heuristic. Its new definition manages to satisfy most aspects of this function's old contract, but not quite all. If you define a package with **package-declare, pkg-super-package** will still return the same package that it used to return. But for packages defined in other, newly available ways, there may be no unique way of defining the "superpackage". The **global** package will probably be returned as the "superpackage" in this case.
- **pkg-refname-alist** still exists and its value is used in roughly the same way. However, it is no longer the case that most package names are found there. In fact, these lists will normally be **nil.**
- Some hairy undocumented features of **package-declare** are no longer supported.
- **apropos, who-calls** and **what-files-call** take different keyword arguments. They used to accept keywords **:superiors** and **:inferiors** to specify whether to look in the superpackage and subpackages of the specified package. Now that packages do not have superpackages and subpackages, the keywords have been changed to **:inherited** and **:inheritors.**
- Package names are now treated much like symbol names with regard to case. In package prefixes, letters are converted to upper case unless quoted with a slash or vertical bar, so it does not matter what case you use. In functions that accept a package name to look up a package, the string or symbol you specify is compared, with case being significant. Thus, if you use a string, the string must contain upper-case letters if the package name does. If you supply a symbol, you can type the symbol in upper or lower-case because **read** converts the characters of the symbol to upper case anyway.

5.2 The Current Package

package Variable
package Variable

These are now synonymous names for a variable whose value is the current package. ***package*** is the COMMON LISP name for **package.**

packagep *anything* Function

Returns **t** if *anything* is a package.

pkg-bind (*package*) *body...* Macro

Executes body with ***package*** bound to *package.*

pkg-goto *package* Function

Sets ***package*** to *package*, but only if *package* is suitable. A package that automatically exports new symbols is not suitable and causes an error without setting ***package.** This is because typing expressions with such a package current would create new external symbols and interfere with other packages that **use** this one.

pkg-goto-globally *package* Function

Sets the global binding of ***package*** (in effect in all processes that do not bind ***package***) to *package*. An error occurs if *package* automatically exports new symbols. Note that the LISP read-eval-print loop binds ***package***, so such loops are not affected by the global binding. Conversely, doing **pkg-goto** inside a LISP read-eval-print loop would not change the global binding. **load** also binds the current package, so in order to change the global binding from your init file, you must use this function.

5.3 Finding All Packages

all-packages Variable

all-packages is a new variable whose value is a list of all packages.

list-all-packages Function

The function **list-all-packages**, with no arguments, returns the same list. This is a standard COMMON LISP construct. Strangely, ***all-packages*** is not.

5.3.1 Package Naming

A package has one name, also called the primary name, and can have, in addition, any number of nicknames. All of these names are defined globally, and all must be unique. An attempt to define a package with a name or nickname that is already in use is an error.

Either the primary name of a package or one of its nicknames counts as a name for the package. All of the functions described below that accept a package as an argument will also accept a name for a package (either a string or a symbol whose pname is used). Arguments that are lists of packages may also contain names among the elements. However, for transportable COMMON LISP, one must not use this feature.

When the package object is printed, its primary name is used. The name is also used by default when printing package prefixes of symbols. However, when you create the package you can specify that one of the nicknames should be used instead for this purpose. The name to be used for this is called the prefix name.

Case is significant in package name lookup. Usually package names should be all upper case. **read** converts package prefixes to upper case except for quoted characters, just as it does to symbol names, so the package prefix will match the package name no matter what case you type it in, as long as the actual name is upper case: **TV:FOO** and **tv:foo** refer to the same symbol. In the functions **find-package** and **pkg-find-package**, and others that accept package names in place of packages, if you specify the name as a string you must give it in the correct case:

(find-package "TV") finds the **tv** package
(find-package "tv") finds nothing

You can alternatively specify the name as a symbol; then the symbol's pname is used. Since **read** converts the symbol's name to upper case, you can type the symbol in either upper or lower case: **(find-package 'TV)** and **(find-package 'tv)** both find the **tv** package since both use the symbol whose pname is **"TV"**.

Each package has a list of local nicknames, which are mapped into packages. These local nicknames serve as additional names for those other packages, but only when this package is current, and only for the sake of package prefixes in **read**. It is permissible to define a local nickname that is the same as the name of some existing package; this is useful for "redirecting" symbol references with package prefixes to packages other than the ones named in the code.

Relevant functions:

package-name *package* Function

Returns the name of *package* (as a string).

package-nicknames *package* Function

Returns the list of nicknames (strings) of *package*.

package-prefix-print-name *package* Function

(Not in COMMON LISP) Returns the name to be used for printing package prefixes that refer to *package*. Note that COMMON LISP does not have such a feature.

rename-package *package new-name* &optional *new-nicknames* Function

Makes *new-name* be the name for *package*, and makes *new-nicknames* (a list of strings, possibly **nil**) be its nicknames. An error is signalled if the name or any of the nicknames is already in use.

find-package *name* &optional *use-local-names-package* Function

Returns the package that *name* is a name for, or **nil** if there is none. If *use-local-names-package* is non-**nil**, the local nicknames of that package are checked first. Otherwise only actual names and nicknames are accepted. *use-local-names-package* should be supplied only when interpreting package prefixes. The use of the second argument is not transportable COMMON LISP.

If a package is supplied as *name*, it is returned.

If a list is supplied as *name*, it is interpreted as a specification of a package name and how to create it. The list should look like (*name super-or-use size*). If *name* names a package, it is returned. Otherwise a package with name *name* is created with **make-package** (see page 60) and then returned. *size* is specified as the size. *super-or-use* should be either the name of a single package, to be specified as the *super* argument to **make-package**, or a list of package names, to be specified as the *use* argument to **make-package**.

pkg-find-package *name* &optional *create-p* Function

(Not in COMMON LISP) *name* and *use-local-nickname-pkg* are passed to **find-package** (see page 60). If that returns a package, **pkg-find-package** returns the same package. Otherwise, a package may be created, according to the value of **create-p**. These values are allowed:

nil	An error is signaled if an existing package is not found.
t	A package is always created.
:find	**nil** is returned.
:ask	The user is asked whether to create a package.

If a package is created, it is done by calling **make-package** with *name* as the only argument. This function is not quite for compatibility only, since certain values of *create-p* provide useful features.

5.4 Creation and Destruction of Packages

While **package-declare** still works, the standard way to create a package now is the new function **make-package** or the defining construct **defpackage**. To eliminate one, use **kill-package** (see page 63).

make-package *name* &key *nicknames use super shadow export prefix-name size invisible import shadowing-import import-from relative-names relative-names-for-me* Function

Creates a new package with name *name* (a string) and nicknames *nicknames* (a list of strings). It is initially made large enough to hold at least *size* symbols before needing expansion. The package is returned as the value.

The following keyword arguments are accepted:

:use	A list of packages or names for packages from which the new package should inherit or a single name or package. It defaults to just the **global** package.
:super	If this is non-**nil**, it should be a package or name to be the superpackage of the new package. The new package will inherit from the superpackage and from all

the other packages from which the superpackage inherits. The superpackage itself is marked as autoexporting; see the section "External and Internal Symbols" for more information. Superpackages are implemented for compatibility only; they are not recommended for use in any new package definitions.

:prefix-name
: This specifies the name to use for printing package prefixes that refer to this package. It must be either the name or one of the nicknames. The default is to use the *name*.

:shadow A list of strings that are names for symbols that should be shadowed in the newly created package. This argument is passed directly to the function **shadow** (see page 68).

:export A list of symbols or names to export in the new package. This is handled by the function **export** (see page 64).

:nicknames and **:use** are the only arguments allowed in transportable COMMON LISP. All of keyword arguments are for ZETALISP only.

:invisible If non-**nil**, means that this package should not be put on the list ***all-packages*** (see page 59). As a result, **find-package** will not find this package, either by its name or by its nicknames. You can make normal use of the package in all other respects (passing it to **intern**, passing it to **use-package** to make other packages inherit from it or it from others, and so on).

:import If non-**nil**, is a symbol or a list of symbols to be imported into this package. You could accomplish as much by calling **import** after you have created the package.

:shadowing-import
: If non-**nil**, is a symbol or a list of symbols to be imported into this package with shadowing. You could accomplish as much by calling **shadowing-import** after you have created the package.

:import-from
: If non-**nil**, is a list containing a package (or package name) followed by names of symbols to import from that package. Specifying *import-from* as **(chaos "open" "close")** is nearly the same as specifying *import* as **(chaos:open chaos:close)**, the difference being that with *import-from* the symbols *open* and *close* are not looked up in the **chaos** package until it is time to import them.

:relative-names
: An alist specifying the local nicknames to have in this package for other packages. Each element looks like (*localname* . *package*), where *package* is a package or a name for one, and *localname* is the desired local nickname.

:relative-names-for-me
: An alist specifying local nicknames by which this package can be referred to from other packages. Each element looks like (*package localname*), where *package* is a package name and *localname* is the name to refer to this package by from package. You will note that the elements of this list are not dotted while those of **:relative-names** are.

pkg-create-package *name* &optional (*super* ***package***) (*size* **200**) Function
(Not in COMMON LISP) Creates a new package named name of size size with superpackage super. This is for compatibility only.

defpackage &quote *name keywords-and-values...* Special form
(Not in COMMON LISP) This is the preferred way to create a package in ZETALISP. (It is compatible with the **defpackage** introduced in Symbolics Release 5.) All the arguments

are simply passed to **make-package** (see page 60). The differences between this function (actually, macro) and **make-package** are:

- **defpackage** does not evaluate any arguments.
- Re-evaluating a **defpackage** for an existing package is allowed; it modifies the existing package in accordance with changes in the definition.
- The editor notices **defpackage** and records it as the "definition" of the package.

IMPORTANT: The latest edition of the *Lisp Machine Manual* documented this function to take arguments in **&key** (property-list) style. However, the keywords and values are actually supposed to be passed in an association list form. *Ignore the version in the manual.* For example, here is the correct version of the example given the bottom of page 653 in the *Lisp Machine Manual*:

```
(defpackage "EH"
  (:size 1200)
  (:use "GLOBAL" "SYS")
  (:nicknames "DBG" "DEBUGGER")
  (:shadow "ARG"))
```

Package attributes in a file's **-*-** line can now have this format

Package: (*name keyword value keyword value...*) ;

which means that the package to be used is name and, if that package does not exist, it should be created by passing name and the keywords and values to **make-package.**

sys:package-not-found Condition

This error condition is signalled whenever you do **pkg-find-package** with second argument **:error, nil** or omitted, and the package you were looking for does not exist.

The condition instance supports the operations **:name** and **:relative-to;** these return whatever was passed as the first and third arguments to **pkg-find-package** (the package name, and the package whose local nicknames should be searched).

The proceed types **:retry, :no-action, :new-name** and **:create-package** may be available.

:retry Says to search again for the specified name in case it has become defined; if it is still undefined, the error occurs again.

:create-package
Says to search again for the specified name, and create a package with that name if none exists yet.

:new-name Is accompanied by a name (a string) as an argument. That name is used instead, ignoring any local nicknames. If that name too is not found, another error occurs.

:no-action (Available on errors from within **read.**) Says to continue with the entire **read** as well as is possible without having a valid package.

package-declare &quote *name super size unused body...* Special form

(Not in COMMON LISP) Is one old-fashioned equivalent of **defpackage** (see page 61). It is no longer recommended for use. It creates a package named *name* with superpackage *super* (another name) and initial size *size*. The unused argument must be **nil**. *body* is now allowed to contain only these types of elements:

shadow *names*
Passes the names to the function **shadow** (see page 68).

external *names*
Does nothing. This controlled an old feature that no longer exists.

intern *names*
Converts each name to a string and interns it in the package.

refname *refname packagename*
Makes *refname* a local nickname in this package for the package named *packagename.*

myrefname *packagename refname*
Makes *refname* a local nickname in the package named *packagename* for this package. If *packagename* is **global**, makes *refname* a global nickname for this package.

pkg-add-relative-name *in-pkg name for-pkg* Function
(Not in COMMON LISP) Defines *name* as a local nickname in *in-pkg* for *for-pkg. in-pkg* and *for-pkg* may be packages, symbols or strings.

pkg-delete-relative-name *in-pkg name* Function
(Not in COMMON LISP) Eliminates *name* as a local nickname in *in-pkg*.

kill-package *name-or-package* Function
(Not in COMMON LISP) Kills the package specified or named. The name **pkg-kill** is also allowed for compatibility.

5.5 Package Inheritance

You now may completely control which packages are inherited by which other packages. Inheritance no longer has to be transitive. **x** can inherit from **y** and **y** from **z** but without **x** inheriting from **z** also. Inheritance can also be multiple. **x** can inherit from two unrelated packages **y** and **w**. However, in any case, only external symbols are inherited. More information on internal vs external symbols is in the following section.

In the past, a package would inherit only from its superior, its superior's superior, and so on. Thus, if **bar** and **quux** were two subpackages of **global**, a new package **foo** could inherit from **bar** and **global**, or from **quux** and **global**, or just from **global**; but **foo** could not inherit from **bar** alone, or **quux** alone, or from **bar** and **quux**, or from **bar** and **quux** and **global**. Now any of these possibilities is possible.

The functions **use-package** and **unuse-package** are used to control the inheritance possibilities of an existing package. The **:use** argument to **make-package** can be used to specify them when a package is created. If **foo** inherits from **bar**, we also say that **foo uses bar**.

use-package *packages* &optional (*in-package* ***package***) Function
Makes *in-package* inherit symbols from *packages*, which should be either a single package or name for a package, or a list of packages and/or names for *packages*.

unuse-package *packages* &optional (*in-package* ***package***) Function
Makes in-package cease to inherit symbols from packages.

package-use-list *package* Function
Returns the list of packages **used** by *package*.

package-used-by-list *package* Function
Returns the list of packages that **use** *package*.

You can add or remove **used** packages at any time.

If one package **uses** several others, the **used** packages are not supposed to have any two distinct symbols with the same pname among them all. An attempt to create such a situation causes an error, which you can override by shadowing. (See below.)

5.6 External and Internal Symbols

Each symbol in a package is marked as external or internal in that package. Symbols created in the package by **intern** are initially internal. You must mark symbols as external if you want them to be so.

The plan is that all the symbols in a package that are intended to be used from other packages will be marked as external.

The internal versus external distinction makes a difference at two times:

> Only external symbols are inherited from other packages
>
> In COMMON LISP, only external symbols can be referred to with ordinary colon prefixes. :: prefixes must be used for internals.

intern (see page 65) works by first checking the current or specified package for any symbol, whether external or not, and then checking all the inherited packages for external symbols only. All the symbols in **global** and **system** are external to start with, so that they can still be inherited, and all new symbols made in them are made external. All symbols in the **keyword** package are also automatically external.

Some other packages also automatically export all symbols put in them. This happens, for compatibility, in any package that has been specified as the "superpackage" in the old-fashioned **package-declare** and **pkg-create-package** functions. You are not allowed to **pkg-goto** one of these packages, and **read** makes a special check to prevent you from creating symbols in them with package prefixes.

Relevant functions:

export *symbols* &optional (*package* ***package***) Function

Makes *symbols* external in *package*. *symbols* should be a symbol or string or a list of symbols and/or strings. The specified symbols or strings are interned in package, and the symbols found are marked external in package.

If one of the specified symbols is found by inheritance from a **used** package, it is interned locally in *package* and then marked external.

unexport *symbols* &optional (*package* ***package***) Function

Makes *symbols* not be external in *package*. It is an error if any of the symbols to be marked not external are not directly present in package.

globalize *name-or-symbol* &optional (*into-package* **"GLOBAL"**) Function

If *name-or-symbol* is a name (a string), interns the name in into-package and then forwards together all symbols with the same name in all the packages that **use** into-package as well as in **into-package** itself.

If *name-or-symbol* is a symbol, interns that symbol in into-package, and then forwards together all symbols with the same name.

The symbol ultimately present in *into-package* is also exported.

pkg-external-symbols *package* Function

Returns a list of all the external symbols of package. *package* can be a package or a package name.

5.7 Looking Up Symbols

The four old functions for looking up symbols work with minor changes. There are also two new ones.

intern *symbol-or-string* &optional *package* Function

Looks up the specified name in the specified package and inherited packages. If *package* is omitted or **nil**, the current package is used.

If a string is specified, a symbol of that name is looked for first in the specified package and then in each of the packages it inherits from. If a symbol is found, it is returned. Otherwise, a new symbol with that name is created and inserted in the specified package, and returned.

If a symbol is specified, lookup proceeds using the symbol's pname as the string to look for. But if no existing symbol is found, the specified symbol itself is inserted in the package. No new symbol is made. Use of a symbol as argument is not defined in COMMON LISP.

intern actually returns three values. The first is the symbol found or created. The second is a flag that says whether an existing symbol was found, and how. The third is the package in which the symbol was actually found or inserted. It will be the specified package or a package from which the specified package inherits.

The possible second values are:

nil	Nothing was found. The symbol returned was just inserted.
:internal	The symbol was found as an internal symbol in the specified package.
:external	The symbol was found as an external symbol in the specified package.
:inherited	The symbol was inherited from some other package (where it was necessarily an external symbol).

intern-soft *symbol-or-string* &optional *package* Function
find-symbol *symbol-or-string* &optional *package* Function

(**find-symbol** is the COMMON LISP name.) Looks for an existing symbol like **intern**, but never creates a symbol or inserts one into package. If no existing symbol is found, all three values are **nil**.

package defaults to the current package if it is omitted or given as **nil**.

intern-local *symbol-or-string* &optional *package* Function

(Not a COMMON LISP function) Like **intern** but looks only in *package*, ignoring the packages *package* normally inherits from. If no existing symbol is found in *package* itself, the specified symbol or a newly created symbol is inserted in *package*, where it permanently shadows any symbol that previously would have been inherited from another package.

The third value is always *package*, and the second one is never **:inherited**.

package defaults to the current package if it is omitted or given as **nil**.

intern-local-soft *symbol-or-string* &optional *package* Function

(Not a COMMON LISP function) Like **intern-soft** but looks only in *package*, ignoring the packages it normally inherits from. If no symbol with the specified name is found in *package*, all three values are **nil**.

package defaults to the current package if it is omitted or given as **nil**.

remob *symbol* &optional *package* Function
unintern *symbol* &optional (*package* ***package***) Function

(**unintern** is the COMMON LISP name) Removes *symbol* from being present in *package*. In **remob**, *package* defaults to *symbol*'s package. In **unintern**, it defaults to the current package.

If a shadowing symbol is removed, a previously-hidden name conflict between distinct symbols with the same name in two USEd packages can suddenly be exposed, like a discovered check in chess. This signals an error.

import *symbols* &optional (*package* ***package***) Function

Is the standard COMMON LISP way to insert a specific symbol or symbols into a package. *symbols* is a symbol or a list of symbols. Each of the specified symbols will be inserted into package, just as **intern** (see page 65) would do.

If a symbol with the same name is already present (directly or by inheritance) in package, an error is signaled. On proceeding, you can say whether to leave the old symbol there or replace it with the one specified in **import**.

5.8 Looping Over Symbols

Several new macros are available for writing loops that run over all the symbols in a package.

do-symbols (*var package result-form*) *body...* Macro

Executes *body* once for each symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once if it is present directly in two or more of package and the inherited packages.

do-local-symbols (*var package result-form*) *body...* Macro

(Not a COMMON LISP form) Executes *body* once for each symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally *result-form* is executed and its values are returned.

do-external-symbols (*var package result-form*) *body...* Macro

Executes *body* once for each external symbol findable in package either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once if it is present directly in two or more of package and the inherited packages.

do-local-external-symbols (*var package result-form*) *body...* Macro

(Not a COMMON LISP form.) Executes *body* once for each external symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-all-symbols (*var result-form*) *body...* Macro

Executes *body* once for each symbol present in any package. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once.

These old functions still work:

mapatoms *function* &optional (*package* "**GLOBAL**") (*inherited-p* **t**) Function
Calls *function* successively on each of the symbols in *package*. Symbols inherited from other packages are included if *inherited-p* is non-**nil**.

mapatoms-all *function* &optional (*package* "**GLOBAL**") Function
Calls *function* successively on each of the symbols in *package* and all the packages that inherit from package. When *package* has its default value, this will include just about all packages.

5.9 The USER Package

In Release 2, the **user** package is an ordinary package that inherits from **global**.

The **user** package used to be the same as the **keyword** package, so in files read into **user** it was not necessary to put a colon on any keyword. This is no longer the case. You must use colons in the **user** package just as in any other package.

5.10 Package Prefixes

In COMMON LISP, a package prefix is used before a symbol to refer to a symbol that is not present or inherited in the current package. (In ZETALISP and NIL, one can also put prefixes in front of any form, and the package prefix will be pervasive during the reading of that form.) **tv:tem** is an example; it refers to the symbol with the print-name **tem** that is visible in the package named **tv**. (**tv** can be the primary name or a nickname.)

Internal symbols that print with package prefixes will print with :: prefixes, as in **tv::tem**, rather than as **tv:tem**. This is because in COMMON LISP a simple colon prefix can be used only for external symbols; a :: prefix must be used if the symbol is internal.

This restriction has *not* been implemented for ZETALISP programs. The colon prefixes in your programs will still work ! But :: prefixes are being printed for informational purposes, and will be accepted by the reader.

A prefix consisting of just **#:** indicates an uninterned symbol. Uninterned symbols are printed with such prefixes, and **#:** can also be used in input to create an uninterned symbol.

Package prefixes are normally decoded when read by checking the local nicknames, if any, of the current package and its superpackages before looking at the actual names and nicknames of packages. You can use a **#** before the colon in the prefix to prevent the use of the local nicknames. Suppose that the current package has **tv** as a local nickname for the **xtv** package. Then **tv:sheet** will get the **sheet** in the **xtv** package, but **tv#:sheet** will get the one in the **tv** package. That symbol will print out as **tv#:sheet** as well, if the printer sees that **tv:sheet** would be misinterpreted by the reader.

The package name in a package prefix is read just like a symbol name. This means that slash and vertical bars can be used to include special characters in the package name. Thus, **foo/:bar:test** refers to the symbol **test** in the **foo:bar** package, and so does **|foo:bar|:test**. Also, letters are converted to upper case unless they are quoted with a slash or vertical bar. For this reason, package names should normally be all upper case.

5.11 Shadowing and Name Conflicts

If multiple symbols with the same name are available in a single package, counting both symbols interned in that package and external symbols inherited from other packages, we say that a name conflict exists.

Name conflicts are not permitted to exist unless a resolution for the conflict has been stated in advance by specifying explicitly which symbol is actually to be seen in package. This is done by shadowing. If no resolution has been specified, any command that would create a name conflict signals an error instead.

For example, a name conflict can be created by **use-package** if it adds a new inherited package with its own symbol **foo** to a package which already has or inherits a different symbol with the same name **foo**. **export** can cause a name conflict if the symbol becoming external is now supposed to be inherited by another package that already has a conflicting symbol. On either occasion, if shadowing has not already been used to control the outcome, an error is signaled and the **use** or exportation does not occur.

Shadowing means marking the symbol actually interned in a package as a shadowing symbol, which means that any conflicting symbols are to be ignored.

package-shadowing-symbols *package* Function

Returns the list of shadowing symbols of *package*. Each of these is a symbol interned in package. When a symbol is interned in more than one package, it can be a shadowing symbol in one and not in another.

Once a package has a shadowing symbol named FOO in it, any other potentially conflicting external symbols with name FOO can come and go in the inherited packages with no effect.

There are two ways to request shadowing: **shadow** and **shadow-import**.

shadow *names* &optional (*package* ***package***) Function

Makes sure that shadowing symbols with the specified *names* exist in *package*. *names* is either a string or symbol or a list of such; any symbols present in *names* are coerced into their print-name strings. Each name specified is handled independently as follows:

- If there is a symbol of that name interned in *package*, it is marked as a shadowing symbol.
- Otherwise, a new symbol of that name is created and interned in *package*, and marked as a shadowing symbol.

In any case, *package* will have a symbol with the specified name interned directly in it and marked as a shadowing symbol.

The primary application of **shadow** is for causing certain symbols not to be inherited from any of the **used** packages. To avoid problems, the **shadow** should be done right after the package is created. The **:shadow** keyword to **make-package** (see page 60) or **defpackage** (see page 61) lets you specify names to be shadowed in this way when you create a package.

shadowing-import *symbols* &optional (*package* ***package***) Function

Interns the specified symbols in *package* and marks them as shadowing symbols. *symbols* must be a list of symbols or a single symbol; strings are not allowed.

Each symbol specified is placed directly into *package*, after first removing any symbol with the same name already interned in package. This is rather drastic, so it is best to use **shadowing-import** right after creating a package.

shadowing-import is useful primarily for choosing one of several conflicting external symbols present in packages to be **used**.

6. Window System Changes

6.1 The FONTS Package No Longer Uses Global

This means that any fonts created in earlier systems will have to be redumped in order to work with Release 2. This has been done for all the system's fonts appearing in the **SYS: FONTS:** directory. There are two ways to do update the fonts to run in Release 2. The first is to write out (using **fed**) a **kst** format file of the font, load that into a Release 2 world and then write out a **qfasl** font file. The other technique is to do the following (in Release 2):

```
(use-package "GLOBAL" "FONTS")
(load file-containing-font)
(unuse-package "GLOBAL" "FONTS")
(compiler:fasd-symbol-value file-to-contain-font 'fonts:name-of-font)
```

6.2 New way of initializing process of TV:PROCESS-MIXIN

Normally, if the *process* keyword argument to **make-instance** of some window flavor incorporating **tv:process-mixin** is a symbol, it is used as the top level function and **make-process** is called with no keyword arguments. But, as an exception, if **process** is **t**, the top level function is to send the window a **:process-top-level** message with no arguments. So, for example, one could write:

```
(defflavor crock-window ()
                        (tv:process-mixin tv:window)
  (:default-init-plist :process t)
  (:documentation "A window which displays a crock."))

(defmethod (crock-window :process-top-level) ()
  (draw-crock self)
  (do-forever
    (update-hands)
    (process-sleep 60.)))
```

6.3 TV:SHEET-FORCE-ACCESS Does Not Prepare the Sheet

The macro **tv:sheet-force-access** (documented in the *Window System Manual*) used to put a **tv:prepare-sheet** into its expansion unless an optional argument was supplied to inhibit doing so.

It turned out that most uses of the macro had no need to prepare the sheet but were neglecting to supply the optional argument. Since combining the two facilities is unmodular, the **prepare-sheet** has simply been flushed from **tv:sheet-force-access**. If you really want to do one, simply write a **tv:prepare-sheet** explicitly in the body of the **tv:sheet-force-access**.

The old optional *dont-prepare-flag* argument is still accepted but has no effect now.

6.4 TV:MAKE-WINDOW Now Identical to MAKE-INSTANCE

Windows can now be created with **make-instance** just like any other flavor instances. The function **tv:make-window** will be supported indefinitely since it is so widely used.

6.5 TV:MOUSE-WAKEUP and TV:MOUSE-RECONSIDER

The window manual says that you should call the function **tv:mouse-wakeup** to report a change in screen configuration. This is not exactly true.

The function **tv:mouse-wakeup** causes the mouse process to look again at the position of the mouse. It is called by the function **tv:mouse-warp**, so that the mouse will be tracked to its specified new position. It is also the thing to use if you redisplay a menu-like window with a new set of menu items, for example, so that the mouse process will notice whether the mouse position is now inside a different menu item.

However, actual changes in the window configuration may make it necessary to force recomputation of which window owns the mouse. This is done by setting the variable **tv:mouse-reconsider** non-**nil**. Calling **tv:mouse-wakeup** may not be enough, since the current mouse position may still be inside the old screen area of a no-longer-eligible window.

6.6 Mouse Clicks Are Blips By Default

If a window has an input buffer and does not define a handler for mouse clicks, they are handled by putting **:mouse-click** blips into the input buffer. It used to be necessary to mix in **tv:list-mouse-buttons-mixin** to get this behavior. Now that flavor is a no-op.

Refer to section 10.1 of the *Window System Manual* for more information.

6.7 :PREEMPTABLE-READ for TV:STREAM-MIXIN

Now all windows that handle **:rubout-handler** also handle the **:preemptable-read** operation. It used to be necessary to mix in **tv:preemptable-read-any-tyi-mixin** to have this operation available. That flavor is now a no-op.

Refer to page 55 of the *Window System Manual* for information on using this operation.

You can also do preemptable input using the **:rubout-handler** operation with the **:preemptable** option. This is a new feature documented in this file.

6.8 Menu Item Types

The value of a **:menu** menu item can now be any form that evaluates to a suitable menu. A menu itself is a special case of such a form, now that menus and other unusual objects evaluate to themselves.

:funcall-with-self is a new type of menu item. The value associated with it is a function of one argument. If the menu item is executed, the function will be called, with the menu (that is the value **self**, in the menu's **:execute** method) as its argument. The value that the function returns is the value of executing the menu item.

6.9 TV:MOUSE-WAIT Takes Who-state as Argument

tv:mouse-wait takes an additional optional argument that, if specified, is displayed as the run state in the who line while the function waits for mouse input.

6.10 Mouse Characters

window

You should no longer use the byte pointer **%%kbd-mouse** in making mouse characters or testing whether a character is a mouse character. It still works at the moment, but may stop working in the future. To avoid problems, convert code as soon as you have switched over to Release 2.

To test, use **tv:char-mouse-p**. To construct, use **tv:make-mouse-char**.

tv:char-mouse-p *char* Function

t if *char* is a mouse character. This function was incorrectly documented as **tv:kbd-mouse-p** in the *Lisp Machine Manual.*

tv:make-mouse-char *button n-clicks* Function

Returns the mouse character for clicking on button *button*, *n-clicks* times. Both *button* and *n-clicks* range from 0 to 2; *n-clicks* is actually one less than the number of clicks. The left button is button 0; the right one is 2.

Continue to use **%%kbd-mouse-button** and **%%kbd-mouse-n-clicks** as byte pointers to extract from a mouse character which button was clicked and how many times.

6.11 TV:MARGIN-SPACE-MIXIN

The mixin **tv:margin-space-mixin** defines a blank margin item. You can leave blank space next to any of the window's edges. The blank space can go between two margin items at that edge, or between the inside of the window and the margin items. For example, it can be used to separate the scroll bar from the inside of the window, or separate the scroll bar from the border, depending on where in the ordering you mix the mixin in.

The mixin defines an init keyword called **:space** whose value specifies how much blank space to leave at each edge. The values you can use are:

t Leaves one pixel of space at all four edges.

nil Leaves no blank space. This turns off the effect of the mixin.

n Leaves *n* pixels of space at each edge.

left top right bottom

Leaves top pixels of space at the top edge, left pixels at the left edge, etc.

Two operations are also defined by the mixin: **:space** and **:set-space**. **:set-space** takes an argument just like the **:space** init keyword and alters the amount of space the mixin is generating. **:space** as an operation returns a list of four values (left top right bottom) describing how much space is currently being taken up by the mixin.

6.12 TV:ADD-SYSTEM-KEY Improvement

If a system key is already defined and you use **tv:add-system-key** to redefine it, the previous definition is restored when you do **tv:remove-system-key** to remove the new definition.

6.13 New String Drawing Primitive

The following function has been added to help speed up string drawing. It is compatible with the Symbolics function of the same name.

tv:%draw-string *sheet alu xpos ypos string font start stop xlim* Function

Draw *string* on *sheet* starting with the character at index *start* and stopping after drawing the character at index *stop*, presuming it all fits. Output starts at *xpos*, *ypos* on the sheet and continues until all appropriate characters are drawn, or until the next character to be drawn would extend past *xlim*. The index of the next character to be drawn, and the xpos where it would go are returned. If a **newline** is encountered, **tv:%draw-string** returns its index and xpos immediately. The sheet's cursor position is ignored and left unchanged.

This function also handles fonted (**art-fat-string** or 16-bit) strings. Therefore, the function **tv:sheet-fat-string-out** is now obsolete; use **tv:sheet-string-out**. The message **:fat-string-out** is also obsolete; use the message **:string-out**.

7. User Interface Changes

The section on the Yank system in the Editor chapter (section 8.2, page 79) is also very relevant to the user interface.

7.1 New function COMMON-LISP

When this function is called in a Lisp Listener, it changes whether COMMON LISP or traditional ZETALISP (actually, their syntax and incompatible functions) are to be used for reading and printing lisp objects. It works by **setqing *readtable***.

It takes one argument, which should be either **t** or **nil**.

See chapter 1, page 3 for basic information on COMMON LISP support. See section 8.11.1, page 82 for COMMON LISP support in ZMacs.

7.2 New Run Bar

A new run bar can occasionally be seen at the bottom of the screen, to the left of the older run bars. This bar goes on whenever the machine would take a sequence break (see the *Lisp Machine Manual* chapter on processes), but cannot because **inhibit-scheduling-flag** is non-**nil**.

7.3 Arguments to APROPOS and WHERE-IS changed

apropos *substring* &optional (*package* ***all-packages***) &key (*inheritors* **nil**) (*inherited* **t**) *dont-print predicate boundp fboundp* — Function

The *package* argument is now the always the second argument (it used to be a keyword argument) The value of this argument may be **nil**, meaning to search all packages, a single package or package name, or a list of packages and/or package names.

where-is now accepts a package or a package name or a list of packages and/or package names as it second argument.

7.4 Beep Types

The system now supplies a non-**nil** beep-type to the function **beep** on certain occasions. These are the types defined so far:

zwei:converse-problem
Used for the beep that is done when Converse is unable to send a message.

zwei:converse-message-received
Used for the beeps done when a Converse message is received.

zwei:no-completion
Used when you ask for completion in the editor and the string does not complete.

tv:notify
Used for the beep done when you get a notification that cannot be printed on the selected window.

supdup:terminal-bell
Used when the remote host sends a "bell" character over while using SUPDUP.

fquery
Used when the **fquery** function beeps for attention.

Those of you who redefine **beep** can use the beep type (the first argument) to produce different sounds for different occasions. More standard beep types will be defined in the future, if users suggest occasions that deserve beep types.

7.5 *VALUES* for Evaluator Loops

LISP Listeners, **break** loops, and the debugger now record all the values of each evaluated form in the variable ***values***. Each process has its own ***values***. The value of ***values*** is a list, and each element is a list of the values of one evaluated form. The most recent forms' values come first.

If a form is aborted for any reason, **nil** is pushed on ***values*** for it.

(caar *values*) is therefore equivalent to the value of the variable * if and only if the last form was not aborted.

7.6 Variable Ratio Mouse Motion

The ratio of mouse motion on the table to mouse cursor motion on the screen now depends on the speed of motion. If you move the mouse slowly, the cursor moves only a little as the mouse moves. As you move the mouse faster, the same amount of mouse motion moves the cursor a long distance.

To control this feature, use this function:

tv:mouse-speed-hack &rest *specs* Function

specs consists of an odd number of elements: alternating scale factors and speeds, followed by one more scale factor. Each scale factor applies up to the speed that follows it. The last scale factor applies to all higher speeds. The standard settings are made with specs of (.6 120 1 200 1.5 400 2.2 700 3.3) so you can see that a speed of 120 is fairly slow, while 700 is moderately fast. A scale factor of 1 corresponds to the mouse motion ratio previously in use. So, **(tv:mouse-speed-hack 1)** would restore the old fixed-ratio behavior.

7.7 Evaluating/Compiling Multi-Font Files.

It now works to evaluate or compile files that contain multiple fonts as specified with the **Fonts** attribute in the -*- line. The old kludge that some users used for doing this should no longer be used.

To make this work in all cases, user-defined readmacro characters should do all input using the function **si:xr-xrtyi** (see its on-line documentation). You may wish to specify arguments of *stream* **nil t**.

Note that if a reader macro detects a syntax error and wants to report this by signaling an Lisp error, it should always make **sys:read-error** one of the condition names and provide the proceed-type **:no-action**, which should be handled by skipping over the invalid data and returning *something* (**nil** is a reasonable thing to return).

7.8 Debugging Changes

7.8.1 Evaluation in the Debugger

When you evaluate an expression in the debugger, it is evaluated in the binding environment of the frame that is current in the debugger.

Initially, the debugger starts out with its current frame being the one in which the error happened. Therefore, your expressions are evaluated in the environment of the error. However, you now have the option of evaluating them in other environments instead.

The debugger command **Meta-S** is no longer necessary in most cases, since simply evaluating the special variable will get the same result. But it is still useful with a few variables such as ***standard-input*** and **eh:condition-handlers** which are rebound by the debugger for your protection when you evaluate anything.

7.8.2 UNADVISE

The function **unadvise** has been generalized in that all arguments now act independently to restrict which pieces of advice should be removed. Thus, if all three arguments are **nil**, all advice is removed. If the first argument is non-**nil**, it is a function spec, and only advice on that function spec is removed. If the second argument is non-**nil**, it is an advice class (**:before**, **:after** or **:around**), and only advice of that class is removed. If the third argument is non-**nil**, it is a position (if it is a number) or a name (if it is a symbol), and only advice with that position or that number is removed.

unadvise-within has been improved in a similar fashion.

7.8.3 :STEPCOND Argument to TRACE

The **:stepcond** argument to TRACE generalizes the :STEP argument. It allows you to specify that STEP should be invoked on the execution of the traced function only if a certain condition is met. The value you provide for the **:stepcond** argument should be a form to be evaluated when the traced function is called; if the form evaluates non-**nil**, the function will be stepped.

7.8.4 MONITOR-VARIABLE No Longer Exists

One consequence of the fact that boxed data words no longer have a flag bit is that **monitor-variable** is no longer possible to implement. This function has been removed.

7.8.5 Describing Condition Handlers

The debugger command **Control-Meta-H** prints a description of the condition handlers established by the stack frame you are looking at.

7.8.6 Overriding *DEBUG-IO*

eh:*debug-io-override* Variable

If **eh:*debug-io-override*** is non-**nil**, the debugger will now use it for its input and output, rather than using the value of ***debug-io***.

7.9 Choose Variable Values Windows

Clicking the right mouse button on a variable's value now puts you in the rubout handler with the old value of the variable there for you to edit. You can use parts of the text of the old value to make up the text of the new value.

Clicking left still puts you in the rubout handler with a blank slate; then you must type the new value from scratch.

7.10 Output of Character Names

The **format** directive `~:C` and the function **format:ochar** now never output a character name for graphic characters other than **Space** and **Altmode.** All other graphic characters are output as themselves, whether or not they have names, since they appear on the keyboard as themselves.

7.11 Terminal T Change

Terminal T now controls just the deexposed Typeout action of the selected window. A new command **Terminal I** controls the deexposed type-In action. (Sadly, **Terminal O** is already in use).

Terminal 0 T
: Just wait for exposure on output when deexposed.

Terminal 1 T
: Notify user on attempt to do output when deexposed

Terminal 2 T
: Permit output when deexposed.

Terminal 0 I
: Just wait for exposure on input when deexposed.

Terminal 1 I
: Notify user on attempt to do input when deexposed

Terminal 2 I
: There is no **Terminal 2 I.** It doesn't make sense.

7.12 Terminal c-Clear-Input is now Terminal c-M-Clear-Input

This keyboard sequence is used to try to unhang some window-system problems. It has been changed so that **c-clear-input** is typeable (by having it quoted with **terminal**, which causes it to lose its special meaning of "flush keyboard typeahead" and be simply passed on to the program which is reading from the keyboard.)

7.13 DRIBBLE-START, DRIBBLE-END gone

Use **(dribble** *filename***)** or **(dribble-all** *filename***)** to start wallpapering output to a file, and **dribble** with no arguments to terminate output and close the file.

7.14 Compiler Behavior

The areas in which compiled code lives are now read-only. This is to catch bugs such as **nconcing** onto a constant list. The print names of interned symbols are also read-only now.

Compilation no longer uses fixed data structures that exist in only one copy. You will no longer get the message "Compiler in process FOO waiting for resources."

7.15 MAKE-SYSTEM Improvements

If **make-system** is done on a system that is not known, the file **SYS: SITE:** *system* **SYSTEM** is now loaded without any query if the file exists.

When **make-system** asks you about a list of files to be compiled or loaded, you now have the option of saying you would like to be asked again about each individual file. Do this by typing **S** instead of **Y** or **N**.

After you type **S**, you will be asked about each file in the bunch just as you would have been earlier if you had specified **:selective** as an argument to **make-system.** Finally you will be asked once again to approve of the entire bunch of files, before processing actually begins.

7.16 APROPOS and SUB-APROPOS Extended

In **apropos**, specifying a non-**nil** value for the keyword argument boundp restricts the search to symbols that have values. A non-**nil :fboundp** argument restricts it to symbols with function definitions. **sub-apropos** accepts the same new arguments.

7.17 LOAD Defaults Are the Default Defaults

COMMON LISP wants **load** to use the default defaults. It seems that if **load** should do so then everything else that used the **load** defaults should do likewise. So the two variables (**cli:load-pathname-defaults** and **fs:load-pathname-defaults**) have been forwarded together.

7.18 Hardcopy Options

Now, options to the system-defined hardcopy functions can be defaulted on a per-printer-type basis.

set-printer-default-option *printer-type option value* Function

This function allows the user to set a default option for a printer type, which the hardcopy functions look at. A common use at MIT may be **(set-printer-default-option :dover :spool t)**, which will cause Dover output to be spooled unless the **:spool** option to a hardcopy function is supplied. Currently defaultable options are **:font**, **:font-list**, **:heading-font**, **:page-headings**, **:vsp**, **:copies**, and **:spool**.

7.19 ZMail Changes

On the LAMBDA, Zmail is now a supported system. There will be a new manual, and introductory documentation as well.

7.19.1 Message-ID Fields.

If you want, ZMail can put a Message-ID field in your outgoing messages. Go into the Profile editor to get this behavior, because the default is not to generate Message-ID fields.

7.19.2 New Command M-X Undigestify Message

This command takes the current message and splits it into its submitted messages so that you can act on them individually. You can set aspects of what the command does by using the Profile editor:

1. Should the original message be deleted ? (Default: *Yes*)
2. Should everything but the header and "table of contents" be clipped out of the original message ? (Default: *No*)
3. Should the name of the digest be append to the subject field of all the new messages so that you can tell from which digest they came ? (Default: *Yes*)

7.19.3 Usual mail file directory option for ZMail

You can set this option in the Profile editor in ZMail. It simply informs ZMail to use a short name for a mail file in a menu, if that file is found in the directory. (The full name of the file is displayed if it has not been read into a buffer yet.)

8. Editor Changes

This chapter covers changes in command names, the yank system, and the rubout handler, among other things.

8.1 Selective Undo

You can now undo an editing change that is not the most recent change you made. If you give the Undo command **C-Shift-U** while there is a region, it undoes the most recent batch of changes that falls within the region. The region does not go away, so you can repeat the command to undo successive changes within the same region. For example, you can undo your changes to a specific Lisp function by using **C-M-H** to create a region around it and then using C-Shift-U.

8.2 Yank Command Improvements

What used to be called the *kill ring* is now called the *kill history* because it is no longer a ring buffer. It now records all the kills you have ever done, in strict chronological order.

Meta-Y still brings older kills into the region, and any particular sequence of **Meta-Y** commands works just as it used to. But the history is not permanently rotated; as soon as a new kill is done, it snaps back to chronological order. We say that **Meta-Y** rotates the history's yank pointer around the history list. **Control-Y** with no argument yanks what the yank pointer points at.

So far, the yank pointer corresponds entirely to what used to be the front of the kill ring, but here are the differences.

- Killing anything moves the yank pointer up to the front of the list.
- Numeric arguments to **Control-Y** count from the most recent kill, not from the yank pointer.

You can think of this as meaning that either killing or using **c-Y** with an argument "un-rotates" any rotation you have done, before it does its work.

Control-Y with an argument of zero prints a list of the first 20 elements of the kill history. Click on one of them to yank it. Click on the message saying that there are more elements, if you want to see the rest of them.

There are several other histories as well as the kill history. They all work just like the kill history, except that you use some other command instead of **Control-Y** to yank from them. **Meta-Y** is used for rotating the yank pointer no matter which history you are yanking from; it simply works on whatever history your last yank used.

For example, the previous inputs in the rubout handler are now stored in a history. The command **Control-C** yanks from it, much as before, except that it now takes arguments exactly like **Control-Y**. **Control-Meta-Y** is a new alias for **Control-C**; it has the advantage of not being a debugger command, so you can use it in the debugger with no extra complications.

All the pathnames you have typed in minibuffers now go in a history. The command **Meta-Shift-Y**, which used to yank the last pathname input, has been generalized to yanks from this history. It takes args just like **Control-Y**, now. Use **Meta-Y** immediately after a **Meta-Shift-Y** to rotate the yank pointer to other pathnames in the history.

All buffer names given as arguments in the minibuffer also have a history. (Actually, each ZMACS window has its own history of these.) **Meta-Shift-Y** is the command for this history as well.

All function specs and other definition names you give as arguments in the minibuffer also have their own history, which is accessible through **Meta-Shift-Y**.

There is no ambiguity in **Meta-Shift-Y**: when the minibuffer wants a pathname, **Meta-Shift-Y** uses the pathname ring. When the minibuffer wants a buffer name, **Meta-Shift-Y** uses the buffer name ring. When the minibuffer wants a definition name, **Meta-Shift-Y** uses that ring. Other rings of minibuffer arguments of particular kinds may be created in the future; **Meta-Shift-Y** will be the way to access all of them.

Note that the command **c-X Altmode**, which repeats previous minibuffer commands, takes arguments just like **c-Y**, and also stores its data in a history. However, this command does not really work by yanking text. There has been no change in the way **c-m-Y** is used to go back to previous minibuffer arguments or to a previous command.

To summarize, here are how the histories are accessed:

Control-Y Kill history; everywhere (including the rubout handler).

Control-Meta-Y
Input history; rubout handler.

Control-C
Control-Meta-Y
Input history; Editor and Ztop.

Meta-Shift-Y
Arg history; minibuffer.

Meta-Y Rotate yank pointer of any history.

The LISP (Edit) window or editor top level, and Ztop mode, now provide infinitely long input histories just like the one that the usual rubout handler provides. Formerly each batch of input read in a LISP (Edit) window or in Ztop mode was pushed on the kill history. Now it goes on the window's or Ztop buffer's input history instead. Use **c-m-Y** to yank the most recent element of the input history, just as you would in the rubout handler, and then use **Meta-Y** to rotate to earlier inputs if you wish.

8.3 More Rubout Handler Commands

The rubout handler now has a mark, and supports the commands **c-Space**, **c->**, **c-<**, **c-W** and **m-W**. They work about the same as the editor commands of the same names.

The rubout handler also now supports **Meta-T**.

Typing **Meta-Status** is now a way to print the rest of the input history beyond the part that Status shows you. A numeric argument specifies how many elements at the front of the input history to skip mentioning. **Control-Meta-Status** does a similar thing for the kill history, to complement the **Control-Status** command.

8.4 Sectionization Improvements

Now each form in a buffer gets its own section. This has several beneficial results.

m-X Compile Buffer Changed Sections will no longer recompile any random forms that are adjacent to functions you have edited. In fact, this command recompiles only sections containing `def...` forms.

Evaluating a random form in the buffer will no longer mark any definition as "already recompiled". Even evaluating a form that is part of a definition will no longer mark the entire definition as "already recompiled."

c-sh-C can now print the name of the function being compiled very quickly, based on the sectionization.

The section nodes for non-definition forms have names that are strings containing the file or buffer name, the function that the form invokes, and a numeric suffix to make the name unique: for example, **QFCTNS-DEFPROP-182**. You will see these section names mentioned in the output of **m-X List Sections** and other commands for listing or visiting sets of sections.

8.5 Buffer Selection History Now Per Window

Each **Zmacs** window now keeps its own history of all buffers. The **c-m-L** command, and defaulting when reading a buffer name argument, both use the selected window's history. (This is the same history that you can yank from using the **m-sh-Y** command when giving a buffer name argument.) The history's "most recent" elements are buffers that have been selected in this window, most recent first. The least recent elements are other **Zmacs** buffers that have not yet been selected by this window. The histories of different **Zmacs** windows all contain the same elements, but they may be in different orders. **c-X c-B** now displays the per-window history.

8.6 Per-Buffer Local Variables

Now you can make any special variable's value local in a specific editor or, in the case of ZMACS, in a specific buffer. For editor user option variables this can be done with a **Meta-X** comand.

zwei:make-local-variable *variable* &optional *value xvcell* Function

Makes *variable* local in the current editor, or the current buffer if this is an editor that can select various buffers (that is, ZMACS). If *value* is specified (whether **nil** or not) then *variable* is set to *value* after it is made local; otherwise it keeps its global value.

The argument *xvcell* is used in ZMACS buffer switching. If non-**nil**, it should be a closure value cell, which is used as the value cell for the local binding. value is ignored when xvcell is given.

zwei:kill-local-variable *variable* Function

Makes *variable* no longer be local in the current editor or buffer. It reverts to its global value.

The easy way to make a ZWEI user option variable (such as ***comment-column***) local is with the command **m-X Make Local Variable.** It reads a variable's pretty name (such as "Comment Column") with completion and makes that variable local. The complementary command **m-X Kill Local Variable** also exists.

m-X List Local Variables prints the names and values of all the local variables in the current editor or current buffer.

8.7 Shifted Mouse Clicks

If is now possible to use "shifted" mouse clicks to give ZMACS commands. ("shifted" means modified by one or more of the CTRL-, META-, SUPER- or HYPER- keys.) Thus it is now possible to give the **m-X Set Key** a "shifted" mouse click (like **control-Mouse-Left-1**) as the key, and to set "shifted" mouse keys in init files using **zwei:set-comtab**.

8.8 Close Parenthesis Displayed for Open Parentheses

When point is before an open parenthesis, the matching close parenthesis now blinks. If point is both before an open parenthesis and after a close parenthesis, the matching open of the preceding close parenthesis is the one that blinks.

8.9 Editor Aids for Common Lisp

There are now two new commands (available from Lisp mode) that allow easy modification of the current readtable for an editor buffer, which controls the particular syntax used for that buffer.

m-X Set Readtable

Changes the **Readtable** attribute of the current buffer, prompting for a readtable name (with completion available). A short description of the names of the standard readtables is available on 41.

To specify a readtable that doesn't already exist, you must exit with **Control-Return**, or type **Return** twice. Then you must confirm with "Yes."

You will also be asked whether to change the attribute list in the text. If you answer yes, the buffer's first line is modified to say that it should be read using the new readtable. This will affect all operations on the file, once you save the buffer.

m-X Set Common-Lisp

This commands changes whether the contents of this buffer are to be regarded as having COMMON LISP syntax, which is done by changing the readtable in effect for this buffer. The command then queries you for whether to change the attribute list in the text as well.

Besides binding the readtable for the editor buffer and the break loop, the readtable attribute also sets the quoting character (one of the two slash characters) as appropriate.

8.10 Lisp Case Changing Commands Renamed

The extended (m-X) commands for changing the alphabetic case of Lisp code have been renamed:

Old name	New name
Lisp Lowercase Region	Lowercase Lisp Code In Region
Lisp Uppercase Region	Uppercase Lisp Code In Region

As a result, typing **m-X lisp** now completes to **Lisp Mode**.

8.11 Font Handling Changes

8.11.1 Yanking and Fonts

When text is moved between buffers and files in which fonts are specified, the precise font of each character is now preserved. If you yank a character that was in font **medfnt** in the buffer where it used to be, it will be in **medfnt** after it is yanked. This may necessitate adding fonts to the font list of the buffer you are editing; if so, you will be asked whether to modify the attribute list (the -*- line) in the text as well.

This new feature applies to the commands **c-Y**, Insert File, Insert Buffer, and **c-X G**. But it only applies when fonts have been specified in the buffer you are editing (with Set Fonts or with a Fonts: attribute). Otherwise, all the yanked text gets put into the default font along with everything else in the buffer. Also, if fonts were not specified in the file or buffer that the text came from, it simply goes into font zero of the current buffer.

8.11.2 New Font Change Commands

The following **Zmacs** keys now have font change commands bound to them:

Control-Shift-J
: This is now Change Font Region, just like Control-X Control-J.

Meta-Shift-J
: This is the command Change One Font Region, which operates on all characters in the region that have a particular font, changing them to another font. It asks for the font to look for first, and then the font to change to. For example, you could specify to change all font A characters into font C.

8.12 New Meta-X Commands

Tags Search List Sections
: This command searches all the files in the currently selected tag table for a string that you specify. It does not itself move point or select a different buffer. Instead it records which sections the string is found in and then prints a list of the sections' names. You can then begin visiting the sections one by one with CONTROL-SHIFT-P.

Start Private Patch
: A private patch is one which is not installed in the system. It is not associated with any specific patchable system, and it does not get a patch version number. It is simply a file of redefinitions that you can load explicitly if you like. **load-patches** does not know about private patches.

: **m-X Start Private Patch** starts editing a private patch. You are asked to specify the pathname of the patch file. Once you have done this, you can put text into the patch with **m-X Add Patch**, just as you can for installed patches. You finish with **m-X Finish Patch**, as usual. This saves and compiles the patch file.

: You can use **m-X Start Private Patch** to resume editing a private patch you created previously. This works whether or not you had finished the patch earlier.

Add Patch Changed Sections
: The command **m-X Add Patch Changed Sections** finds all sections you have changed the text of, in all buffers, and asks you for each one whether to do **m-X Add Patch** of its text. But sections that have been **Add Patch**ed already since their last modification are excluded. If you answer the question **P**, then all the rest of the changed sections in the same buffer are patched without further question.

Add Patch Buffer Changed Sections
: This is similar but considers only the current buffer's sections.

8.13 CONTROL-X 4 J Jumps to Saved Point in Other Window

If you save a location in a register with **c-X S** *register*, you can jump to it again with **c-X J** *register*. Now you can also select the other window (in or entering two-window mode) and jump to the saved location in that window, by using **c-X 4 J** *register*.

8.14 Minor Command Changes

- The command **Control-Shift-D** now prints the full documentation of the function which point is inside a call to. **Control-Shift-D** is thus analogous to **Control-Shift-A**. **Meta-Shift-D** is still available if you wish to specify the function to be documented.
- The default version for the command **m-X Source Compare** is now **:newest** for both the first and the second input file.
- The **m-X View File** command now displays the file with its correct fonts if the file specifies fonts in its attribute list.
- The **Meta-X** commands Copy File, Rename File, Delete File, and Undelete File have been changed to do prompting and querying in a new way.

  If the pathname specified has no wildcards, no prompting or querying is done. The operation is just performed.

  If there is a wildcard, then a list of the files that match it is printed all at once. You are then asked to confirm with **Y** or **N**. If you say **Y**, then all the files are operated on forthwith.

8.15 Commenting a Region

This command puts a comment starter (the value of **zwei:*comment-begin***) in front of each line starting in the region, except for blank lines. With numeric argument, it removes precisely the value of **zwei:*comment-begin*** (a single semicolon in LISP mode) from each line in the region that starts with one.

You can use **c-X c-;** to comment out the lines of the region before recompiling a function. Later, use **c-U c-X c-;** to remove the commenting thus made. Only a single semicolon is removed, so any lines that were comments before commenting out the region remain comments after un-commenting the region.

8.16 Dired

Dired now displays files that are really deleted on disk with a lower case **d** in the first column. Files whose deletion has been requested but not done are displayed with a capital **D**. If you request undeletion of an actually deleted file, the file is displayed with a capital **U**, but such operations as printing, editing, or applying a function to the file are not allowed since the file is really still deleted.

When the current buffer is a **Dired** or **BDired** buffer and you issue a command that reads a filename, the default filename is now the file whose line you are pointing at.

A new command to edit the superior directory of the current buffer's directory can be found on the **<** key in **Dired**.

9. Site File Changes

9.1 Logical Host Definitions Kept in the SITE directory

The definition (and translations) for logical pathname hosts can now be kept in the site directory. Refer to page 31 for a discussion of this new feature of logical hosts. The **SYS** host, by convention, is now defined in the file **SYS: SITE: SYS TRANSLATIONS**. Use of **:sys-host-translation-alist** and variables to hold express the **SYS** host translations is considered obsolete.

9.2 Specification of File Servers

Due to a change in the internals of the pathname system, the name of the site option that lists file server hosts is now called **:file-server-hosts**. This is just like **:chaos-file-server-hosts**, except that It contains hosts that the machine knows about by default. If someone tries to reference a host that is a file server, but which did not appear on the list, he will still get the desired behavior, the host object will dynamically be added to the pathname host list, if need be. Thus, **SYS: SITE: SITE LISP >** no longer needs to be changed merely to add new file servers - as long as they appear in the host table, that should be sufficient.

The hosts do not have to be Chaosnet hosts. Currently, Chaosnet access is the only kind of remote access. However, the name change is anticipation of access by other protocols, such as TCP FTP. Other access flavors are for Local File access and local LMFILE access.

It is now possible is specify the default device of a host by using the site option **:host-default-device-alist**, an alist of host names and device names (with the colon). This option is effective for Twenex and VMS hosts. An example of the use of the option:

```
(:host-default-device-alist '(("OZ" . "OZ")))
```

Here, we are overriding the default name **PS**. This option is especially useful for VMS hosts, since the "default" device is some logical name that can differ from system to system.

If, for example, a Twenex host is configured for a non-**PS** primary structure name, this option should be used, to eliminate some strange interactions that can happen when the truenames of files are compared against supplied names.

One user-level change has occured because of this. Suppose one supplies the file name

```
SRC:<L.IO.FILE>ACCESS.LISP
```

which is intended to name a file on the Twenex host OZ, to a program, and the default host is OZ. The pathname parsing system will try to see if SRC: is a file host, and that may entail going over the network to contact a host table server to see if SRC is a host. Still, when it determines that SRC is not a host, it will recognise it as a device instead, and the pathname will become **OZ:SRC:<L.IO.FILE>ACCESS.LISP**. All of this checking is a result of the unfortunate choice, made for historical reasons, of colon being the delimiter for both host and devices. The rule has now been changed so that the first colon delimits the device. Therefore, when supplying a pathname with an explicit device, but defaulting the host, a colon must also be supplied before the device, like

```
:SRC:<L.IO.FILE>ACCESS.LISP
```

9.3 New site option :STANDALONE

If the Lisp Machine is just by itself, the option should be supplied with value **t**. This will cause the Lisp Machine to not to try to use the Chaosnet for getting the time, for one thing. On the Lambda, the time will obtained from the SDU's clock. On the CADR, the time will be obtained from the user.

Concept Index

Lisp Index

Release 2 Conversion Guide

October 1984

1000 Mass. Ave., Cambridge, Massachusetts 02139
Formatted with LaTeX on November 6, 1984.

Chapter 1.

Introduction

This documentation covers the incompatibilities for programmers between Release 1 and Release 2. The first section discusses in general what one can expect when converting code that ran in Release 1 for Release 2. The second section covers more specific incompatibilities between the releases. Note that two system facilities, packages and **defstruct**, have changed enough to warrant their own chapters in the Release Notes. The specifics of the changes and new features are discussed in those chapters, but the general guidelines for those facilities are discussed here in brief as well.

First, note that Release 1 binaries (QFASL files) cannot be successfully loaded in a Release 2 system. Because of new instructions, Release 2 binaries cannot be loaded into a Release 1 system. If you need to conditionalize code that depends on Common Lisp between Release 1 and Release 2, you may be advised that the symbol **common** appears on the features list in Release 2 but not in Release 1. (All implementations of Common Lisp have **common** on the features list.) You can use the read-time conditionalization feature of the reader to write code that will run in both Common and non-Common Lisp systems. Here is an example of its use: the Common Lisp form will be used in LMI Release 2, the newer versions of NIL, and other Common Lisp implementations, while the second form will be used in LMI Release 1, older versions of NIL, and Maclisp.

```
(defun print-with-radix (n the-base stream)
  #+common
  (write n :stream stream :radix t :base the-base :escape t)
  #-common
  (let ((*nopoint t))
    (format stream "#~D~VR" the-base the-base n)))
```

Here, the interpreter and the compiler never even get to see the forms meant for the "other" implementation of Lisp.

Chapter 2.

Common Lisp Issues

2.1. File Attribute Lists

Because of the adoption of Common Lisp, the default base in the Lisp Machine is now ten. If you have files with no base attribute, you should add one to the file while you are editing it in Zmacs. Another attribute that should be added to file attribute lists is a new one, the Readtable attribute. Readtables implement ability to run Common Lisp in the Lisp Machine without having to reference the new incompatible Common Lisp functions with the CLI: package prefix. Therefore, the readtable attribute indicates the syntax (for the editor) as well as what functions are to be used. No readtable attribute means to use the default (initially, the value of the readtable); a value of T means ZetaLisp; CL or Common-Lisp selects the Common Lisp mode. [1] Therefore, Release 1 files should be have an attribute to ensure that the correct mode is selected.

For changing the parts of the attribute list in an editor buffer, the extended (m-X) commands Set Readtable, Set Common Lisp, and Set Base will change the attribute list, as well as the values of any variables that may need to be changed in the editor.

2.2. Lexical Scoping

In Release 2, with the adoption of Common Lisp, lexical scoping becomes the rule in the interpreter. The compiler also supports full lexical scoping, including upward lexical closures. Lexical scoping is actually compatible with Lisp code that would compile without any warnings in Release 1, but because of some internal changes, the following must be noted:

[1] This scheme is also compatible with NIL.

1. The scope of local special declarations (usually heralded by **local-declare** or **declare** within a block-type special form like **defun**) has changed (but the old way still works for now – see the specific section for details). However, many of the uses of local special declarations can now go away because of lexical scoping. One of the most common instances of using a local **declare** in Release 1 is to make a variable special so that a **lambda**-expression being passed to a function (such as **mem**) can make a free reference to that variable.
2. By default, an error is signalled when a free reference is made to an undeclared variable. Therefore, any forgotten **defvars** or special declarations quickly manifest themselves in Release 2.

Here is an example of a function that will work in both releases, but one that can also take advantage of lexical scoping. Here is a function that is like **apropos**, except that only variables of a given type are printed:

```
(defun variable-type-apropos (string pkg type)
  (declare (special type))
  (apropos string pkg
                 ':predicate
                 #'(lambda (s) (and (boundp s)
                                    (typep (symeval s) type)))))
```

The local **declare** is needed here, in Release 1, because the lambda-expression that is being passed as the **:predicate** makes a reference to the variable **type**, which does not appear in the argument list of the lambda-expression. So, **type** variable is made special so that the passed function can refer to it.

This can be modernized to the following, after taking out the local declaration and using the new Common Lisp name for **symeval**:

```
(defun variable-type-apropos (string pkg type)
  (apropos string pkg :predicate
                   #'(lambda (s) (and (boundp s)
                                      (typep (symbol-value s) type)))))
```

This is more intuitive; we do not have to make **type** special because the predicate function is textually within the scope where **type** appears. Some other advantages to avoiding special variables are:

- Using the faster method of lexical variable lookup. (It is a simple extension of ordinary, old-style local variable lookup.)
- Avoiding name conflicts. If **type** is declared special by some other program, it may be possible for something wrong to happen if type were declared special in this function.

2.3. All Keyword Arguments Are Optional

In Release 2, **&optional** makes no difference after **&key** in a definition's lambda-list. Keyword arguments that are not supplied will not cause an error in Release 2, but will simply default to **nil**.

Chapter 3.

Other Changes

This part of the documentations covers some areas affected by incompatible changes in Release 2 that are more specific in nature – some, but not all of these changes are related to Common Lisp.

3.1. Array Order Has Changed in Release 2

The order of storage for array elements has been switched to be compatible with Maclisp and Common Lisp. Arrays are now stored in *row-major* order, which means that the last subscript varies the fastest through memory. The Lisp Machine used to store array elements in *column-major* order. The value of the constant **sys:array-index-order** is now **t**.

The change in order should be transparent for most programs, but there are some situations where code may have to be changed:

- Loops that go through large, multidimensional arrays (arrays where (**array-number-dimensions** *array*) > 1) will have to be rewritten if they took advantage of the older order to decrease paging.
- Programs that deal with pixel arrays should have been using **ar-2-reverse** and **as-2-reverse** to correctly reference the correct dimensions when given horizontal and vertical "coordinate" arguments.
- Use of multidimensional indirect arrays. If you have been exploiting the order in which elements appear when the displaced and source arrays are of different rank, you will have to rewrite the code. See section 8.2.1, especially the second paragraph on page 167, which discusses indirect arrays.

This is all explained in more detail in section 8.11 (pages 182-3) of the *Lisp Machine Manual.*

3.2. The Package System

The package system has been completely reimplemented. It is now a superset of the Common Lisp specification. Programs that used the hierarchical nature of the package system will have to be changed carefully; the structure of the package system is now an inheritance graph. The user and keyword packages are separate now; while this will not affect users who have always followed documentation about keywords, it will cause problems when programs running in Release 2 try to read data from Release 1, where some of that data was meant to be in the keyword package. Here is a quick rundown of the implications; read the chapter on packages for the details.

- package-declare is obsolete. Use defpackage, which is easier to use.
- The hierarchical structure that made symbol references like foo:bar:baz possible is gone.
- user and keyword are different packages keyword symbols always print out with colons in Release 2, but not in Release 1. This creates a problem with interchange of printed Lisp forms between releases, if keywords are included. (This could happen with data in files, or with a network protocol.)
- Many of the old package functions are still around, but they might not do exactly what was documented under Release 1. This is due to packages being arranged by inheritance, rather than by a strict hierarchy.
- Packages are now named like symbols.[1] In a clean Release 2 system, (pkg-find-package "tv") will get an error. A correct specification for the TV package would be 'TV or "TV". This kind of package specification (using lower case letters in a string) appears to be relatively common in Release 1 programs.
- In addition to changes in the package system, there has been symbol movement since System 94 (Release 1). Some symbols that were local to a package have now been globalized. Many of these are names of Common Lisp functions that were previously in the FS package. Also, note that since the keyword package is really a different package from the user package, keywords always print out with a colon even when they have the same name as a symbol in the global package.

3.3. Incompatible Calling Sequences

The following is a list of incompatible functions and macros that take a different set of arguments than they did in Release 1. Constructs marked with an asterisk (*) will still accept the old calling sequence, but are planned to change incompatibly in the future.

[1]More precisely, they are named like the *print names* of symbols.

```
break *
y-or-n-p
yes-or-no-p
si:print-object
si:print-list
select-match
```

Other classes of functions also have new incompatible calling sequences:

- **Evalhook** functions.
- **Applyhook** functions.
- **Macroexpand-hook** ????
- Macro expanders. [These can still take just argument for now.]

3.4. Function Warnings

You will get warnings, when compiling code with calls to obsolete functions, to use the preferred Common Lisp functions that are upward compatible. For example, **array-dimension-n** should be replaced by **array-dimension**. Such obsolete functions still work, but will go away in the future. Relatively few of these warnings should occur in well-written Release 1 programs, since the green (fifth) edition of the manual had also documented many of these functions as obsolete. But now that the newer functions have become part of the Common Lisp specification, the obsolete functions should not be used anymore.

3.5. Input and Output

3.5.1. Character Objects

Character objects are here, but it is not necessary for Zetalisp programs to always use character-oriented functions if they handle characters; most character functions will work on integers, and *vice versa*. The normal **aref** on strings (arrays of type **art-string** and **art-fat-string**) still returns integers; there is a Common Lisp version, **cli:aref**, which is exactly the same as **aref** except it returns character objects out of strings. (Using **vref** on strings will also return character objects, because it is a synonym for **cli:aref**.) Integer and character comparison and coercion actually work on both types of data, though this should not be relied upon in future releases. In particular, if you are using **aref** (in a character-oriented user-interface) to reference a command in a command table, for instance, the "character" should be coerced to an integer with **char-int**, even if the character is not actually a character object in your program. Besides, it is better to use the functions that operate on characters to make the code easier to understand, and to port to other Common Lisp systems.

When the standard Zetalisp readtable is in effect, #\character reads in as an integer; in Common Lisp syntax, such input will be read in as a true character object. In Zetalisp syntax, character objects print out as #≠/character; #\character currently is equivalent to #/character, which reads in as an integer.

3.5.2. Variable Issues

Though not predominantly an incompatible programming change, Common Lisp has given new names to many of the standard Lisp input and output "control variables" that control slashification (now called *escaping*), input and output radices, and the verbosity of the printout. The old names are still accepted, but programmers are encouraged to change the variable names in their code at their convenience.

Common Lisp has also introduced new variables that control other aspects of the reader and printer. Programmers should especially be interested in ***print-array***, which prints out arrays readably, and ***print-circle***, which prints out circular structure (which can be built out of all kinds of structure, including conses, arrays, and instances) readably. The variable ***print-gensym***, when T, prints out uninterned symbols so that they can be read back and still be uninterned. A complete description of the variables that affect printing can be found in section 23.2 (page 514) of the Lisp Machine Manual.

There is a new Common Lisp variable called ***print-radix***; if T, numbers will be printed out with some indication of what radix they should be read in with. If ***nopoint** is T, then ***print-radix*** really does take effect; otherwise, the old behavior with ***nopoint** takes effect.

3.6. FORMAT

The **format** function has changed; there are some new and incompatible control-string ("tilde") directives. An incompatible directive that is quite common in typical **format** control strings is ~X, which used to perform a tabbing function – it now prints out its argument in hexadecimal. Occurrences of ~X should be replaced to ~@T. The old ~G has been moved to ~@; ~E and ~F have been extended to take more arguments but they have not become incompatible. For more information, read the documentation of the **format** function in the Lisp Machine Manual. (The Common Lisp release notes also briefly describe the new extensions to **format**). Remember that the error signalling functions and some other utilities pass their control-string arguments to **format**.

Since there is no "edit callers" command in Zmacs that can help you to track down obsolete **format** control strings, if you want to track them down, you can now get acquainted with the extended string search capabilities of Zmacs if you have not used them before. The functionality, which includes pattern matching, is available through the Zmacs extended commands **String Search** and **Tags Search**. [2] When typing a search string, use ⟨c-H ⟩ ⟨Help ⟩ to describe the various patterns that can be matched. For example, typing in the search string ~⟨c-H ⟩⟨c-X ⟩ X will search for an occurrence of the string ~, **any character**, X.

3.7. Syntax changes

Because of the incompatibilites of Release 1 and Release 2 syntax, the exchange of objects using printed representation will not always work correctly between releases. (The situation arises when reading in a data file produced by the **print** function, or using Lisp forms in a network protocol: one must be careful if one release is doing the reading and the other is doing the printing.) In general, Release 2 can read anything that can be printed readably in Release 1, but not the other way around. Besides the problem with syntax, there are certain situations that would cause what was meant to be a keyword (at that time, a symbol in the **user** package) to be printed out without a colon prefix in Release 1. Names of flavors of most hosts and pathnames have changed in Release 2, so hosts and pathnames printed out by Release 1 will not read back in, either. (If this is a serious problem in your application, notify us and LMI will supply a fix.) Most of these problems can be avoided by simply switching all machines at a site to Release 2 at the same time.

3.8. Logical Pathnames and Hosts

Logical hosts have been changed in an upward compatible manner; now, the translations are not directories, but mappings from one (usually wildcarded) specification to target file name. Also, the directories of a logical pathname can be structured: IO: FILE: can be considered a "subdirectory" of IO:. The newer features of logical pathnames (and the definition of logical pathname hosts) is documented in more detail in the Release 2 notes. It is now possible to specify, for example, on a typical Twenex host's SYS host translations, that all directory names translate to directories under the <L> directory: one translation for one-level directories, one translation for two-level directories (like IO: FILE:), and one translation that will specially translate CHAOS: HOSTS TEXT in PS:<SYSTEM>HOSTS2.TXT, which is where host table usually resides on Twenex host. Here is the *translation* needed: it would get passed to **fs:add-logical-pathname-host**:

[2] Try "Apropos" in Zmacs on the string "**tags table**" to learn how to select various kinds of groups of files (loaded buffers, systems defined with defsystem) as tags tables.

```
(("CHAOS; HOSTS TEXT" "PS:<SYSTEM>HOSTS2.TXT")
 ("*.*;" "PS:<L.*.*>")
 ("*;" "PS:<L.*>"))
```

Site translations can now be updated automatically: for example, SYS translations (the definition of the SYS host) are actually kept in the site directory. The function **fs:make-logical-pathname-host** requests that a logical host get its translations from the SITE: directory. For Release 2, at least the SYS translations file must be present. If you have defined your own logical hosts for your own systems, you can use this feature, but it is not necessary. (In that case, you can still use **fs:add-logical-pathname-host**.)

3.9. Host Device Specification In Filenames

Because of changes in the internals of the file access system, a change has been made in the rules for determining the device of a partially-specified pathname.[3] If you wish to specify a different device in a pathname that is still going to have the same host as the default pathname, you must put a colon in front of the device name.[4] This can be considered mostly a user-interface change, but users are advised to keep this in mind if they suspect that they have incompletely specified physical pathnames wired into their programs. The change is really important only for users of VMS, Twenex, and ITS file servers.

---

[3] This restriction is compatible with the Symbolics system, in which the same change had taken place a while ago.

[4] Actually, omitting the colon may still work, but if there is a file server host on the network with the same name as the device, something will probably go wrong. Even if there is no such host, your machine may also try to contact host table servers to check to see if the name is valid. The time to do this can be noticeable.

Interprocessor Communication
The Extended STREAMS Interface

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Extended STREAMS Interface^{tm} is a trademark of LISP Machine Inc.
LMI Lambda^{tm} is a trademark of LISP Machine Inc.
NuBus^{tm} is a trademark of Texas Instruments Corp.
UNIX^{tm} is a trademark of American Telephone & Telegraph.

Comments on this document and other documentation should be addressed to:
LMI
1000 Mass. Ave.
Cambridge MA 02139

Phone: (617) 876-6819

Principal writer of this document is Meryl R. Cohen.
Formatted with BoTeX.

Table of Contents

1. Introduction

The Extended STREAMS Interface facilitates interprocessor communication. This software allows you to share memory and program control between the Lambda LISP and UNIX (68010) processors and also between the two LISP processors on a Lambda 2x2. Communication between processors can be accomplished in several different ways:

- by sending information through the Chaosnet
- with the Share TTY feature
- using the shared memory area

Each of these three methods has its advantages and drawbacks. However, for applications involving a lot of data transfer, using the shared memory capabilities will be considerably faster.

This document covers all three methods of LISP/UNIX communication with emphasis on the shared memory techniques. The examples in the body of the text are for illustration only; there is an extended example included with your software in **SYS:EXAMPLES;STREAMS** that can be run online. You can find the text of this example in Appendix B.

Familiarity with the streams concept in general, Chaosnet, and UNIX and **C** programming will be necessary in understanding the following material. The *LISP Machine Manual*, as usual, is the best alternate source of information; it contains the definitive documentation on the Zeta-LISP implementation of Chaosnet and the machine subprimitives which are used in LISP/UNIX communication.

Changes in Release 2

The major changes to the Extended STREAMS software in Release 2 are additional UNIX functions that allow access to shared memory, and a new approach to devices in ZetaLISP. The UNIX material is covered in Section 4.2, "Shared Memory from UNIX"; the new device information is in Section 5.1, Devices from LISP.

2. Chaosnet

Chaosnet is the least direct connection between LISP and UNIX. Passing data and programs over Chaosnet involves the greatest amount of processor overhead, but allows you to communicate between processors that reside on separate busses. For instance, you can use Chaosnet to communicate between a Lambda LISP processor and another computer running UNIX. The various protocols that are available and user definable are useful tools for solving many problems and for initiating and coordinating high performance intra-bus transactions.

All functions are transparent with respect to bus configuration. Many symbolic Chaosnet contact names, such as **FILE** and **TIME**, are also transparent with respect to the operating system of the particular processor.

The following example shows how you would set up a call from the LISP processor over Chaosnet to the UNIX processor and request UNIX to run a program for you.

The **chaos:open-stream** function is useful for connecting to the UNIX **eval** server.

```
(defun simple-unix-eval (host command)
  (with-open-stream (s (chaos:open-stream host
                         (string-append "EVAL " command)))
    (format t "~&@c ~A~%" command)
    (do ((c))
        ((null (setq c (send s ':tyi))))
      (send standard-output ':tyo
            (selectq c
             ((#o12 #o15) #\return)   ;LISP to UNIX character
             (#o11 #\tab)             ; set translation.
             (t c))))))
```

The command to **eval** is given to the UNIX shell, for example:

```
(simple-unix-eval "unix-test-a" "ls //etc")  ;this produces the
                                             ; effect of ls /etc.
myhostname
passwd
hosts
...
```

Here is an example using both file-system protocol and **eval** protocol:

```
(with-open-file (s "unix-test-a://tmp//foo.c" ':out)
   (format s "#include <stdio.h>~
             ~%main()~
             ~%{ printf(/"Hello world.\n/");}~%")

(simple-unix-eval "unix-test-a" "cc //tmp//foo.c")  ;this produces the
                                                    ; effect of cc /tmp/foo.c.
```

```
(simple-unix-eval "unix-test-a" "//tmp//a.out")    ;this produces the
                                                   ; effect of /tmp/a.out.
Hello world.
```

3. Share TTY

Share TTY actually uses the shared memory area set aside in Extended STREAMS, but you can take advantage of it using operations that look like regular stream operations, without worrying about any issues of "safe" versus "dangerous" memory addresses.

The simplest way to take advantage of this feature is with (SYSTEM) **U**. This will give you direct access to the UNIX system as another window reachable through the Lambda window system. You can use this window as you would any other UNIX connected terminal, yet still quickly switch back to your LISP and editor windows.

The UNIX window feature is implemented using a low-overhead character stream between LISP and UNIX. Under UNIX it is a device such as **/dev/ttyl0**, **/dev/ttyl1**, etc.; under LISP it is a stream of type **unix:unix-stream** which is built on **si:buffered-stream**. There are eight such devices.

The default configuration has two "login enabled" ports, and six disabled ones. This means that you can have two **SYSTEM** U-created UNIX windows at once. You can use the other lines for applications that want a unix-stream for communication between LISP and a UNIX process; for instance, **PICON/RTIME** uses this interface. You can change the relative number of enabled and disabled ports at any time by editing the UNIX file **/etc/ttys**. The file will contain entries that look like this:

```
...
  ...
17ttyl0
17ttyl1
07ttyl2
07ttyl3
...
  ...
```

Entries that begin with a **1** are login enabled; those that begin with a **0** are disabled. To login-enable an additional port, change the first character of the entry from zero to one. Conversely, to disable a port, change the first character from one to zero.

For example:

```
(defconst *s* (open "unix-stream-3:"))
#<UNIX-PORT 3>
```

The corresponding device under UNIX would be **/dev/ttyl3**, and the usual stream operations are supported. One caveat: since ZetaLISP does not use the traditional ASCII character codes, the integers returned by the **:tyi** message to a **unix-stream** need to be given the UNIX interpretation of the ASCII character set. Hence it is usually useful to build a stream around a **unix-stream** to do the LISP/UNIX character set translations.

There are two kinds of translation you typically want: one like the **FILE** protocol uses, to handle all characters normally stored in strings and files (8-bit representation), and another, to handle those characters and operations associated with terminals.

The **simple-unix-window-mixin**, as used by the **simple-unix-window** that implements the SYSTEM **U** command, provides both simple character set translation and display operation translation. Conventional terminal escape codes are defined to generate the **:clear-screen**, **:insert-line**, **:clear-eol**, and other operations including the handling of inverse video.

Here is the flavor used to implement the share TTY feature.

unix:unix-stream (si:buffered-stream) Flavor

Use this flavor stream for simple LISP/UNIX interprocessor communication.

Note that you should not use **make-instance** with this flavor; rather, **open** a device of type "**unix-stream-*n***" as in the example above. Alternatively, you can use the following function.

unix:find-unix-stream *with-login-p* Function

Returns the device pathname of the first free **unix-stream**. You can then operate on this. ***with-login-p*** determines whether or not to return a login-enabled stream.

See Section 5.1, "Devices from LISP", for more information on the device/pathname relationship, and Section 5.2, "Devices from UNIX", for more information and functions to use from the UNIX side.

4. Shared Memory

The shared memory area set aside by the Extended STREAMS software provides the most direct connection between different processors. You can use it to communicate between LISP and UNIX on a Lambda Plus, and between the two LISP processors on a Lambda 2x2. This method involves the least processor overhead, but requires the most care on the part of the programmer. The shared memory area is by default, 20K bytes; you can change this from the **config** program.

Facilities that relate directly to shared memory are covered below; the general Multibus and NuBus functions are described in Appendix A.

4.1 Shared Memory from LISP

These are the facilities for accessing shared memory from LISP.

si:*global-shared-memory-size* Variable

Number of bytes in the shared memory area. **Never set this variable**; if you want to change the size of the shared memory area, do so from **config**.

si:*global-shared-memory-8* Variable

An **art-8b** array indirected to the shared memory area.

si:*global-shared-memory-16* Variable

An **art-16b** array indirected to the shared memory area.

si:*global-shared-memory-32* Variable

An **art-32b** array indirected to the shared memory area. Currently, the high seven bits of a number get overwritten with a data type when you read from a location in this array. To get true 32-bit access, use adjacent locations in the **si:*global-shared-memory-16*** array or use the functions described below.

si:share-mem-read *address* Function

Reads a 32-bit value from a given *address*. Since this function always reads 32 bits of data aligned on the word boundary, the lower two bits of the address are ignored.

si:share-mem-write *address data* Function

Writes a 32-bit value to *address*. Again, it will write only along word boundaries, so the lower two bits of the address are ignored.

4.2 Shared Memory from UNIX

In order to use the UNIX shared memory functions you need to let **C** know about them. They are defined in **/usr/lib/libshare.a**. To do this, add the following line to the top of your **C** files.

```
#include <share.h>
```

You also need to specify the **-lshare** option to cc, the compile/link command.

```
cc vision.c -lshare
```

These are the UNIX functions and variables that will allow you to take advantage of Extended STREAMS from the UNIX processor. They fall into several catagories.

- Shared memory setup functions.
- Byte swap primitives.
- Multibus and NuBus read and write functions.
- Functions that access the system configuration structure.

4.2.1 Shared Memory Setup

char ***sharebase** — Variable

Pointer to shared memory area.

sharesize — Variable

The size in bytes of the shared physical memory area.

share_setup () — Function

Sets the variables **sharebase** and **sharesize**. Returns -1 on failure.

4.2.2 Byte Swap Primitives

The 68010 and the Lambda LISP Processor use different conventions for how numbers are stored. The 68010 stores the low order byte on the left; this is the so-called "big-endian" convention, that IBM uses. The Lambda LISP Processor (and the SDU) store the low order byte on the right; this is the "little-endian" convention also used by DEC. To reconcile these notational differences, several functions swap these bytes and allow you to convert from one format to the other.

swapn (*destptr, srcptr, nwords*) — Function

long *destptr, *srcptr;
int nwords;

Byte reverses and copies ***nwords*** words from *srcptr* to ***destptr***. The source and destination pointers can have the same value.

SWAB32 (*x*) — Macro

Expands into an expression that byte reverses *x*. Since this is a macro it is fast because it expands into inline code. However, it evaluates its argument four times and so should not be used with large expressions or expressions with side effects; in such cases, use the function defined below.

long **swab32** (*x*) Function
long x;
Returns a byte-reversed copy of *x*.

4.2.3 Multibus and NuBus Functions

The file **/dev/nubus** is an extension of the UNIX **mem** device. The following functions access the NuBus at the address specified, and bypass the UNIX memory mapping used in **/dev/mem** and **/dev/kmem**. All reads and writes are done through the **ioctls** defined in **/usr/include/mem.h**. **Do not read from or write to /dev/nubus directly.**

This file is accessible only to the superuser. **CAUTION:** There is no protection from bus timeouts, and an access that causes a timeout will crash UNIX. This problem will be fixed in the release of System V UNIX.

mread8 (*addr*) Function
long addr;
Reads an 8-bit value from the Multibus at the 20-bit address ***addr***.

mwrite8 (*addr, data*) Function
long addr, data;
Writes an 8-bit value, ***data***, to the Multibus at the 20-bit address ***addr***.

mread16 (*addr*) Function
long addr;
Reads a 16-bit value from the Multibus at the 20-bit address ***addr***.

mwrite16 (*addr, data*) Function
long addr, data;
Writes a 16-bit value, ***data***, to the Multibus at the 20-bit address ***addr***.

In the following functions ***slot*** must be a number between 0 and 31, inclusive.

nread8 (*slot, addr*) Function
long slot, addr;
Reads an 8-bit value at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nwrite8 (*slot, addr, data*) Function
long slot, addr, data;
Writes an 8-bit value, ***data***, to the NuBus at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nread16 (*slot, addr*) Function
long slot, addr;
Reads a 16-bit value at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nwrite16 (*slot, addr, data*) Function
long slot, addr, data;
Writes a 16-bit value, ***data***, to the NuBus at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nread32 (*slot, addr*) Function
long slot, addr;
Reads a 32-bit value at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nwrite32 (*slot, addr, data*) Function
long slot, addr, data;
Writes a 32-bit value, ***data***, to the NuBus at the address specified by the 8-bit ***slot*** and the 24-bit ***addr***.

nread (*addr*) Function
long addr;
Reads a 32-bit value from the 32-bit address, ***addr***.

nwrite (*addr, data*) Function
long addr, data;
Writes a 32-bit value, ***data***, to the 32-bit address, ***addr***.

4.2.4 System Configuration Structure Access

The file **/dev/sysconf** is a special extension to **/dev/nubus**. The file is readable by anyone, but writable only by the superuser. It allows access to just the **sysconf** and **procconf** structures. A file offset of **0** corresponds to the first byte of the **sysconf** structure. When accessed directly the **/dev/sysconf**, the structures will be in Lambda Processor byte order. Access through **/dev/sysconf** is restricted to memory that is guaranteed to exist, so it is safe to read beyond the defined area.

getprocconf (*pp, pn, max*) Function
struct procconf *pp;
int pn, max;
Reads up to ***max*** bytes of the ***pn***th procconf structure into memory at ***pp***. Byte-reverses all the words in the structure.

getsysconf (*sp, max*) Function
struct sysconf *sp;
int max;
Reads up to ***max*** bytes of the sysconf structure into memory at ***sp***. Byte-reverses all the words in the structure.

5. Devices and Allocation

The following functions deal with shared devices: devices available from both LISP and UNIX. In order for the system to know about the devices, the SDU's (System Diagnostic Unit's) **config** program must be run to correctly configure the system. (For details see the *Release 2.0 Installation Packet.*)

5.1 Devices from LISP

ZetaLISP devices are coming to be thought of as analogous to files, just as UNIX devices are. This means that you can operate on devices through their pathnames, and macros like **with-open-file** will work appropriately.

```
(with-open-file (str "sdu-serial-b:")
  (format str "This is a test. ~C#o215 ~C#o212))
```

You can allocate and deallocate devices, and perform other operations by sending messages to the objects gotten by parsing the device pathname; however, since devices in ZetaLISP are not (unlike UNIX) really identical to files from a software point of view, you need to go down an extra level to find the device object. To do this, send the parsed pathname the **:host** message. You can then send this object the messages for operations that you want carried out.

For example:

```
(send (send (fs:parse-pathname "medium-resolution-color:") :host)
:allocate-if-easy)
```

To find out the available devices you can either look at the **Devices** option of **PEEK** or evaluate the variable **si:all-shared-devices.** Looking at **PEEK** is the method of choice, because the **si:all-shared-devices** only contains shared devices, and is not guaranteed to remain a stable part of the system. What **si:all-shared-devices** returns is a list of all the shared device *objects*; not the device *pathnames.* This means that you can send messages directly to these objects without first sending them a **:host** message.

5.1.1 Device Messages

These are some of the user callable methods for the flavor **si:shared-device.** They are actually inherited from **si:basic-shared-device.** For more information see the file SYS:SYS;SHARED-DEVICE.LISP.

:owner Operation on **basic-shared-device**

Returns either **nil** which means that the device is free; a number from 0 to 31, which is the slot number of the processor that currently owns the device; or **:not-on-bus**, which means that you don't physically own such a device.

:quad-slot Operation on **basic-shared-device**
For a NuBus device, this returns the quad-slot the device occupies; else **nil**.

:device-still-owned-by-me-p Operation on **basic-shared-device**
Returns **t** if the same processor still owns the device; **nil** if its either **:not-on-bus**, free, or owned by someone else.

:error-if-i-dont-own-device Operation on **basic-shared-device**
If **:device-still-owned-by-me** would return **nil** this signals the appropriate error.

:allocate-if-easy Operation on **basic-shared-device**
Allocates the device and returns **t** if it is currently free; returns **nil** if the device is not on the bus, or is owned by someone else.

:allocate Operation on **basic-shared-device**
Allocates the device and returns **t** if it is currently free; signals the appropriate error if not.

:deallocate Operation on **basic-shared-device**
Deallocates the device if owned by the current processor, else just returns.

5.1.2 Configuration Variables

These are some variables used by the shared device function.

si:*sys-conf* Variable
A representation of the system configuration structure. Use **describe** on the variable to get a "human readable" response.

si:*my-proc-conf* Variable
A representation of the processor configuration structure for this particular processor.

5.2 Devices from UNIX

The following standard **C** system calls allow you to access devices from the UNIX side. This is a brief review of these functions, in LISP-style documentation. For complete documentation see the *UNIX Reference Manual.*

open (*device, mode*) Function
char *device;
int mode;
The returned value is **int**, a *file descriptor*. (A file descriptor (*fd*), is an identification number that you use when referring to that device in functions.) A negative return value indicates failure. A *mode* of 0 indicates reading; *mode* of 1 is writing; *mode* of 2 is for both reading and writing.

read (*fd, buffer, length*) Function
char *buffer;
int fd, length;
The returned value is an **int**, the number of characters actually read. A negative value indicates failure; zero indicates logical end of file.

write (*fd, buffer, length*) Function
char *buffer;
int fd, length;
The returned value is an **int**, the number of characters actually written.

To find out what devices are defined for UNIX look at the **/dev** directory under UNIX. Currently many more devices are defined for LISP than for UNIX.

Here is an example that measures the cycle frequency of a LISP/UNIX/LISP communication channel.

```
main()
 {int f; char c[1];
  f = open("/dev/tty14",2);
  if (f<0) {printf("error"); exit(0);}
  while (1)
  { read(f,c,1);
    write(f,c,1);}}

;; from LISP

(defconst *p* (make-instance 'unix:unix-stream ':port-number 4))

(defun test (&optional (n 10000.) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j)))
      ((= j n))
   (send *p* ':tyo 5)
   (send *p* ':tyi))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
        "cycles per second"))
```

Appendix A. Memory Functions From LISP

There are two ways to view shared physical memory, other than at the device/stream level which uses shared physical memory in its device buffers. One way is through system calls accessing a particular memory location. In LISP this would be a call to a function such as **%nubus-read**. To accomplish this kind of memory access under UNIX, you need to use the **share** library functions discussed earlier, in Section 4.2, "Shared Memory From UNIX".

The second way to access shared physical memory is through virtual-memory/physical-memory mapping, as normal language-specific references to data structures which are previously arranged to have some fixed relationship with the virtual memory subsystems of the processors under consideration. This second way is more powerful but inherently more difficult, because it can bring to the forefront the problems of finite memory and disk resource allocation which were previously handled by the system.

The functions described below are both powerful and dangerous; work carefully to ensure that these functions are used in ways that don't hurt your programming environment. **CAUTION:** The Lambda does not handle non-existing memory exceptions. Reference to non-existing memory with any of the **%bus** functions will result in the machine halting with a bus timeout. The SDU can recognize this condition and restart the processor. In other words, if you use this function to read information from a memory location that does not exist you will crash the Lambda and may need to warm or cold boot it.

NuBus Functions

%nubus-read *slot byte-address* Function

Returns the contents of a word read from the NuBus. Addresses on the NuBus are divided into an 8-bit slot number which identifies the physical board being referenced and a 24-bit address within the slot. (Slot numbers on the bus go from F0 through FF.) The address is measured in bytes and therefore should be a multiple of four. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%nubus-write *slot byte-address word* Function

Writes the contents of a word to the NuBus. **Caution:** This function can crash the Lambda if you access nonexistent memory.

Multibus Functions

%multibus-read-8 *address* Function

Reads an 8-bit byte from the Multibus byte address. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-write-8 *address value* Function

Writes an 8-bit byte to the Multibus byte address. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-read-16 *address* Function

Reads a 16-bit halfword from the Multibus byte address. NOTE: To use this function you need to have the latest version of the SDU hardware, which is revision **K**. To find out whether you have this revision, look at the assembly number on the SDU board (slot 15). The last character of the assembly number is the revision letter. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-write-16 *address value* Function

Writes a 16-bit halfword to the Multibus byte address. NOTE: This function requires revision **K** of the SDU hardware. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-read-32 *address* Function

Reads a 32-bit word from the Multibus byte address. **Caution:** This function can crash the Lambda if you access nonexistent memory.

%multibus-write-32 *address value* Function

Writes a 32-bit word to the Multibus byte address. NOTE: This function requires revision **K** of the SDU hardware. **Caution:** This function can crash the Lambda if you access nonexistent memory.

The **%multibus** functions can be effectively used for writing simple device drivers for heavily buffered Multibus devices which can be efficiently handled by a busy-wait.

The following example illustrates one way to write a function that writes to an array processor on the MultiBus.

```
(defconst *opcode-reg* #x0A00C)
(defconst *status-reg* #x0A00D)
(defconst *data-start* #x0A00E)
(defconst *op-clear* 0)
(defconst *op-fft* 1)

(defun fft-data-array (x)
  (%multibus-write-8 *opcode-reg* *op-clear*) ; reset machine clear
  (%multibus-write-8 *status-reg* 0)
  (dotimes (j (array-length x))
    (%multibus-write-8 (+ *data-start* j) (aref x j)))
  (%multibus-write-8 *opcode-reg* *op-fft*)
  (process-wait "Array Processor"
    #'(lambda (reg) (not (zerop (%multibus-read-8 reg))))
    *status-reg*))
```

Certain areas of virtual memory are by default mapped to Multibus and NuBus memory. Some functions for dealing with this are defined in the file `SYS:MULTIBUS;MAP`. This file must be loaded if you want to use the following function and variables.

si:describe-multibus-address-space Function

Provides a listing of what address space is free, what is used, and what is mapped to the NuBus.

Below are three arrays mapped to the Multibus. All three do a 32-bit access; then, for the **art-8b** and the **art-16b**, all but the relevant 8, or 16 bits are stripped off. Therefore, you can use them for accessing device buffers, but not in a controller situation, because they may confuse device registers.

si:*multibus-bytes* Variable

An **art-8b** array mapped to the Multibus.

si:*multibus-halfwords* Variable

An **art-16b** array mapped to the Multibus

si:*multibus-words* Variable

An **art-32b** array mapped to the Multibus.

Safe Address Space

Address space available for use by applications programmmers occasionally changes. The best way to make sure that address space can be used is to call LMI and ask. Outside of Massachusets call 1-800-872-LISP. Within the state call 1-800-325-6115.

Appendix B. Online Example

This file can be found online in sys:examples;streams.

```
;;; -*- Mode:LISP; Package:(STE global); Fonts:(cptfont); Base:8 -*-

;; Copyright LISP Machine, Inc. 1984
;;   See filename "Copyright" for
;; licensing and release information.

;; A self-contained example of streams software usage for
;; testing the performance of and documenting the LAMBDA<->UNIX interface.
;; This code runs in system version 1.120, unix-interface version 12.
;; 10/13/84 00:10:24 -George Carrette.
;; modified for release II beta-test 2/26/85 13:23:09 -George Carrette.

;; To run the tests:
;; (1) Create the C programs by running (CREATE-C-PROGRAMS)
;;     These functions illustrate some of the higher level protocals.
;; (2) Create a split-screen with two lisp listeners.
;;     Use RUN-C-PROGRAM in the top window, switch to the bottom and
;;     use the corresponding lisp function.

(defun attached-unix-host ()
  "Returns host object for attached unix-host if it exits otherwise NIL"
;; Relevant variable:
;; si:*other-processors* list of structures of type SI:OTHER-PROCESSOR
  (dolist (op si:*other-processors*)
    (let ((host (SI:GET-HOST-FROM-ADDRESS
  (si:%processor-conf-chaos-address (si:op-proc-conf op))
  ':CHAOS)))
      (if (typep host 'fs:unix-host)
  (return host)))))

(defun temp-unix-path (name type)
  (fs:make-pathname ':host (attached-unix-host)
    ':directory "TMP"
    ':name (string-append (string-upcase si:user-id)
  "-"
  name)
    ':type type))

;; a simple "null-device" for testing.

(defconst *p* (open "unix-stream-4:"))

(defun null-device (message &rest ignored)
  (selectq message
```

```
    (:tyi 0)
    (:tyipeek 0)
    (:which-operations '(:tyo :tyi :tyipeek :untyi :string-out))))

(defconst *null* (closure () #'null-device))

;; This uses the FILE protocal and EVAL protocal.

(defun share-compile-string (name string)
  "writes out the string as name.c and C compiles it to name"
  (with-open-file (stream (temp-unix-path name "C") ':out)
    (princ string stream))
  (simple-unix-eval (attached-unix-host)
    (format nil "cc ~A -o ~A -lshare"
    (send (temp-unix-path name "C") ':string-for-host)
    (send (temp-unix-path name ':unspecific)
  ':string-for-host))))

(defun simple-unix-eval (host command)
  (with-open-stream (s (chaos:open-stream host
  (format nil "EVAL ~a" command)))
    (format t "~&% ~A~%" command)
    (do ((c (send s ':tyi) (send s ':tyi)))
((null c))
      (send standard-output ':tyo
    (selectq c
      ((12 15) #\return)
      (11 #\tab)
      (t c))))))


(defvar *c-programs* ())

(defun enter-c-program (name string)
  (setq *c-programs* (delq (ass #'string-equal name *c-programs*)
   *c-programs*))
  (push (list name string) *c-programs*)
  name)

(defun create-c-programs ()
  (dolist (p *c-programs*)
    (create-c-program (car p))))

(defun create-c-program (x)
  (let ((p (ass #'string-equal x *c-programs*)))
    (format t "~&;Writing and compiling ~A.C" (car p))
    (apply #'share-compile-string p)))

(defun run-c-program (name)
  (simple-unix-eval (attached-unix-host)
```

```
    (send (temp-unix-path name ':unspecific)
  ':string-for-host)))


;;; The tests

;;; open loop frequency

(defun test-olf (&optional (n 1000.) (stream *p*) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j)))
      ((= j n)
       (send stream ':tyo #/S))
    (send stream ':tyo #/?))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
"cycles per second"))


(enter-c-program "OLFT" '
|//*  program for open-loop sink response *//
#include <stdio.h>
main()
 {int f,n; char c[1];
  f = open("//dev//tty14",2);
  if (f < 0) {printf(/"open lost\n/"); exit(0);}
   while(1)
   { n = read(f,c,1);
     if (n == 0) {printf("got end of file\n"); exit(1);}
     if (n < 0) {printf("read lost\n"); exit(0);}
     if (*c == 'S') {printf("Been told to stop\n"); exit(1);}}}
|)

;; the closed loop frequencey is the basic "remote-function-call"
;; overhead time. With this implementation it is highly dependant on,
;; and usually limited by the lisp scheduler timing because
;; of the process-wait which encumbers the ':tyi to the unix share tty.
;; As things stand, without adding an interrupt driven process wakeup
;; feature to the lispmachine system, the unix processor can
;; affect a process on the lispmachine in no less than 1/60'th of
;; a second. realtime programming applications needing faster response
;; times should consider more low-level clock-break and scheduler
;; modifications. A faster speciallized remote function call mechanism
;; itself calls for a special microcoded function. However, the
;; following is more than reasonable for any job that takes more
;; than half a second in the unix processor.

(defun test-clf (&optional (n 100.) (stream *p*) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j)))
      ((= j n)
```

```
        (send stream ':tyo #/S)
        (print (if (eq (send stream ':tyi) #/0)
  "Unix process stopped ok"
"Unix process failed to reply to stop")))
    (send stream ':tyo #/?)
    (send stream ':tyi))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
"cycles per second"))

(enter-c-program "CLFT" '
|//* program freq.c for closed-loop. *//
#include <stdio.h>
main()
 {int f,n; char c[1];
  f = open("//dev//tty14",2);
  if (f < 0) {printf("open lost\n"); exit(0);}
  while(1)
  { n = read(f,c,1);
    if (n == 0) {printf("got end of file\n"); exit(1);}
    if (n < 0) {printf("read lost\n");exit(0);}
    if (*c == 'S') {printf("Been told to stop\n");
                    c[0] = '0';
                    write(f,c,1);
                    exit(1);}
    n = write(f,c,1);
    if (n < 0) {printf("write lost\n"); exit(0);}}}
|)


(defsetf si:share-mem-read si:share-mem-write)

(defun share-mem-read-single-float (addr)
  (float-68000-32b (si:share-mem-read addr)))


(defmacro share-mem-read-bit (j)
  `(ldb (byte 1 (remainder ,j 32))
(si:share-mem-read (quotient ,j 32))))

(defun float-68000-32b (x)
  "Take 32bits, a 68000 float, and return a lisp float object"
  ;; note: This takes byte reversal into account. It doesnt try to be
  ;; efficient in its use of lispmachine arithmetic.
  (// (* (expt -1 (ldb #o3701 x))
 (expt 2.0 (- (ldb #o3007 x) #o100))
 (+ (ldb #o2010 x)
    (ash (+ (ldb #o1010 x)
    (ash (ldb #o0010 x)
 8.))
 8.)))
```

```
     #o100000000))

(defun 68000-32b-float (x &aux sign exp frac)
  "Take a lispmachine floating point number and return 32 bits suitable for
the 68000"
  (cond ((zerop x)
 0)
('else
 (cond ((small-floatp x)
(cond ((< x 0.0)
       (setq sign 1)
       (setq x (- x)))
      (t
       (setq sign 0)))
(setq exp (+ (- (si:%short-float-exponent x) #o101) #o100))
(setq frac (ash (- (si:%short-float-mantissa x) (expt 2 16))
(- 23 16))))
       ((floatp x)
(cond ((< x 0.0)
       (setq sign 1)
       (setq x (- x)))
      (t
       (setq sign 0)))
(setq exp (+ (- (si:%single-float-exponent x) #o2001) 127))
(setq frac (ash (- (si:%single-float-mantissa x) (expt 2 30))
(- 23 30))))
       (t
(ferror nil "Not a floating point number: ~S" x)))
 (ferror nil "foo, work on this tommorow"))))


(defun test-inc-loop  (&optional (n 100.) &aux time)
  (setq time (time))
  (do ((j 0 (1+ j))(value))
      ((= j n)
       (send *p* ':tyo #/S)
       (print (if (eq (send *p* ':tyi) #/0)
  "Unix process stopped ok"
"Unix process failed to reply to stop")))
    (setq value (test-inc-1 j))
    (or (= value (1+ j))
(format t "~&;Error, expecting ~D, got ~D"
(1+ j) value)))
  (list (quotient n (quotient (time-difference (time) time) 60.0))
"cycles per second"))

(defun test-inc-1 (integer)
  "When the program test-inc is compiled on a unix system with
cc test-inc.c -lshare
```

```
and then executed, you can call (test-inc n) and it will
write the integer n into the shared-array area, signal the
unix process to do the computation, wait for the computation
to complete, then return the result, which is N+1 in this case."
  (si:share-mem-write 0 integer)
  (send *p* ':tyo #/?)
  (send *p* ':tyi)
  (si:share-mem-read 0))


(enter-c-program "INCT" '

|//*  program test-inc *//

#include <stdio.h>
#include <share.h>

main()
 {int f,n,*p,val;
  char c[1];
  f = open("//dev//tty14",2);
  if (f < 0) {printf("open lost\n"); exit(0);}
  if (share_setup() < 0) {printf("share setup lost\n"); exit(0);}
  p = (int *) sharebase;
  while(1)
  { n = read(f,c,1);
    if (n == 0) {printf("got end of file\n"); exit(1);}
    if (n < 0) {printf("read lost\n");exit(0);}
    if (*c == 'S') {printf("Been told to stop\n");
                    c[0] = '0';
                    write(f,c,1);
                    exit(1);}
    val = p[0];
    val = SWAB32(val)+1;
    p[0] = SWAB32(val);
    n = write(f,c,1);
    if (n < 0) {printf("write lost\n"); exit(0);}}}
|)


;; these test values are probably wrong (i.e. TOO LOW) for Release II.
;; 2/26/85 13:33:07 -gjc

;; (test-inc-loop)  20 Hz.


;; results:
;; (test-clf 100. *p*)              52.2 Hz. using freq.c
;; (test-clf 10000. *null*)         6.5 KHz.
;; (test-olf 1000. *p*)  303.3 Hz. Using cat /dev/tty14 > /dev/null
```

```
;; (test-olf 1000. *p*)  220.0 Hz  Using freqc.c
;; (test-olf 10000. *null*)  9.8 Khz.
```

Lisp Index

Lambda Tape Software

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

This document corresponds to LMI Release 2.

Any customers having specific suggestions for the next magnetic tape software release, or comments on this documentation, should address them to:

Dr. Sarah Smith
LMI
1000 Massachusetts Avenue
Cambridge MA 02138
(617) 876-6819

Table of Contents

Introduction

This manual documents the tape software for the LMI Lambda series of LISP machines. This software is a workable magnetic tape system with all the needed functionality for dumping and retrieving files and disk partitions.

This document is intended for both users of the tape software and programmers who want to incorporate tape functions into their programs. To use the high-level facilities provided, you will need to know basic LISP syntax and how to use the LISP Listener. Documentation conventions are those of the *LISP Machine Manual.*

The first part of this document describes the tape software functions for copying files onto tape, getting them off tape, and generally providing the services that any operating system tape utility program would provide. (The major exception to this is a true file backup system, which is really a separate facility and which is still being implemented.)

The second part of this document describes the programmer's view of the tape software. Most of the interface to the tape software is at the stream level; in fact, no hardware command-oriented documentation is supplied. The ''LMFL'' header format is documented (in section 2.6, page 13) in case a need arises for a tape to be read or written by a non-LMI machine. (For example, there is the **lmtar** for LMI-supplied UNIX systems.) Except for obtaining streams to tapes, the programming interface follows LISP Machine stream conventions as closely as possible.

General Information

Lambda and CADR tape software can each read and write tapes for the other; however, since the formats are not completely compatible differences are noted in this document when appropriate.

All keyword arguments (those arguments that appear after the **&key** symbol in the argument list) are optional. *unit* arguments, which are for selecting one of multiple drives on a controller, need not be specified. The term "generic tape function" refers to functions prefixed by **mt-**; these are the only ones that should be called directly by a user, as they select the correct special operations according to the type of tape drive being used.

When a directory is said to be specified in "internal format," this means the same format as that found in pathname objects: single-level directories are strings, while multi-level directories are lists of strings:

```
RPK;  => "RPK"        RPK.DOC;  => ("RPK" "DOC")
```

This internal format is exactly the same as the directory element of a pathname object. This scheme is used for labeling the files on the tape so that they can be easily copied from or copied to (restored to) any of the supported file system types (LISP Machine, UNIX, Twenex, Tenex, ITS, and VMS) without having to actually translate the file name syntax of the system from which the file was copied.

In Release 2, keywords (such as the symbol **:byte-size**) do *not* have to be preceded by a quote, because keyword symbols are self-evaluating. Don't worry about changing old software, the quotes will not affect its ability to run under Release 2. The colons, however, are always needed. This document follows the more modern convention; any examples that do not quote keywords in a context in which they previously would have to have been quoted must be read more carefully by users of earlier releases.

1. User Tape Functions

This chapter documents functions that *can* be used from programs, but are usually used as commands to LISP through a LISP Listener window.

Functions for putting a tape drive offline, rewinding the tape, and positioning the tape to append more files to it are documented in section 2.2, page 11.

1.1 Selecting and Initializing the Tape Device

Before you use a tape drive you need to allocate it. This reserves use of the drive to the current process. To do this, use the same **open** function that you would use to open a file. For example:

```
(open "half-inch-tape:")      ;note colon

                              ;or...

(open "quarter-inch-tape:")      ;note colon
```

This will return a shared device object,

#⊂shared-device-style-object⊃#

that you will need to close when you are done.

Alternatively, you can allocate the device using **with-open-file**. This will allocate the tape drive, execute the *body* and deallocate the device automatically when it's done.

```
(with-open-file (s "half-inch-tape:")
...
            body...
)
```

Note that these functions merely reserve the particular device for your use; in order to do anything with the device you should use the generic tape functions described in the following section.

On the CADR, tape operations can be directed to only one controller. However, the Lambda can use both the quarter-inch cartridge drive (via the SDU) and an industry standard drive (usually a Cipher tape drive) through the TAPEMASTER controller connected to the Multibus. When the Lambda boots, the default tape device is whichever one is actually present; if both tape devices are available, the quarter-inch cartridge drive is selected by default. To change the default drive type, use one of the following functions:

fs:use-quarter-inch-tape Function

Changes the mode of the generic tape functions (the **mt-** functions) to use the Quarterback 1/4" tape drive.

fs:use-half-inch-tape Function

Changes the mode of generic tape operations to use the TAPEMASTER controller with a half-inch tape drive, either Cipher or Kennedy.

Sometimes the controller may end up in an inconsistent state, and thus appear to not be working. Before investigating a hardware problem, try calling this function:

fs:mt-reset Function

Resets the tape controller according to the value of **fs:quarter-inch-tape-mode**. Use this function only to unwedge the tape controller. In general, if you want to abort a tape operation, use **CTRL-**$\boxed{\text{ABORT}}$ to ensure that the tape controller isn't left in a strange state.

The variable **fs:*quart-paranoia-margining*** (see page 12) may be set if the quarter-inch drive is giving poor results. However, you should do this only as a last resort.

1.2 Tape Utilities

This section documents higher-level functions for using the tape drives. Such functions are self-contained, report what they are doing, and may query the user. Mostly, they copy various kinds of data (files, disk partitions) back and forth between the tape and disk (usually, a file system). If you wish to use some facet of these functions for your own specialized application, please refer to the latter section of this document, which describes the lower-level access to the tape software and various calling and storage conventions used for storing files on tape.

fs:mt-write-partition *partition* &optional (*unit* **0**) Function

Writes ***partition*** on ***unit*** to tape; then writes an **eof** mark and spaces backwards to it. This ensures that the last partition is in fact followed by another **eof**. Note that the ***unit*** argument refers to the unit number of the *disk* on which the partition is stored; the partition is always written to tape unit 0. If the partition to write is not on your local machine, then ***unit*** must be specified as a string giving the name of the machine. For example:

```
(fs:mt-write-partition "lod5" "lam3")
```

The function **fs:mt-write-partition** is built on top of **si:copy-disk-partition**. You can specify **"mt"** to be used as a source or a destination with this function.

To restore a partition from disk, use **fs:restore-magtape** (see below).

fs:restore-magtape &key (*host* **si:local-host**) (*query* **t**) *transform directories* *copy-options tape-options* Function

Restores the files (possibly some disk partitions) from the tape onto the file system of *host* (by default, the local machine). If *query* is **t** (the default), then you are asked, file by file, whether to restore the file onto the disk. Answering `'P'` to the query turns off querying for the remainder of the files on tape, except for partitions, which are considered a special case. If a partition is encountered, you will be queried for a place to copy the partition (this may even be a band on another machine).

If *directories*, a list of directories in internal form, is supplied, only files on tape from those directories (and their subdirectories) are considered for restoring.

The *copy-options* argument, passed to **fs:fs-copy-file**, is usually not needed. The *tape-options* argument is supplied to **fs:make-mt-file-stream** (see page 9); this argument is usually not used, either.

If *transform* is supplied, it is either a function of five arguments or a symbol whose **fs:tape-restore-transform** property is a function, as described below.

The transform function's arguments are the *host* (which is a host object, not a string), the *directory* (in internal format), the *name*, the *type*, and the *version* of the file which is ready to be restored. The function returns either a pathname (which then becomes the name of the file to restore the file from tape onto disk) or **nil**, meaning to skip the file and go on to the next one. When a *transform* is supplied and *query* is **t**, the filename displayed in the prompt is the one that the transform returns.

There are currently two predefined transforms. One is called **:ask-and-default**, the other is **:standard-sys**. The first asks you what translation to use every time it encounters a new file directory on tape. When it asks you for a transform (note the question mark "?"prompt], type in a directory to use; the host will default to the one used in the prompt. You should always supply the directory component. For example:

```
Translation for the directory FRED: FOO; ? BAZ:
```

will translate the file `FRED: FOO; DOC.TEXT` to `FRED: BAZ; DOC.TEXT`, and

```
Translation for the directory FRED: L.PATCH; ? SRC: L.S98.PATCH;
```

`FRED: L.PATCH; SYSTEM-98-23.QFASL` to `SRC: L.S98.PATCH; SYSTEM-98-23.QFASL`

Notice that you can supply a different host from the default one.

When *query* is **t**, you should supply the correct directory for translation even when you are not planning to restore the first file from that directory. If you want **:ask-and-default** to use a new set of translations (i.e:, you want it to forget the ones given it before), **setq** the variable

fs:*ask-per-directory-defaults* to **nil**.

The second predefined **:transform**, **:standard-sys**, should be used only when restoring a tape of system source files, when your site's **sys:** host translations are not of the form **SYS: FOO; => L.FOO;**. **:standard-sys** expects that all the files on the tape were dumped from directories under a **L:** or **NL:** hierarchy. (This transform should not usually be used by customers unless documentation that comes with a release source tape says to.)

Here is an example of a user defined tape transform function:

```
;;; Change X; to TAPE.X; on the mounted tape for all directories; also,
;;; change TXT to TEXT, and LSP to LISP, for those types. Finally, make,
;;; all versions :NEWEST and don't restore files from the SYSTEM
;;; directory.
(defun (:property :my-xform fs:tape-restore-transform)
       (host dir name type version)
       version ; this argument not used
   (if (equal dir "SYSTEM") () ; ignore this file
     (fs:make-pathname ':host host
       ':directory (if (stringp dir) (list "TAPE" dir)
                       (cons "TAPE" dir)) ; Multilevel directory here
       ':name name ':version ':newest
       ':type (cond ((string-equal type "TXT") "TEXT")
                    ((string-equal type "LSP") "LISP")
                    ((t type))))))
```

After evaluating the above form, **:my-xform** can be given as the **:transform** argument to **fs:restore-magtape**.

fs:fs-copy-file *from to ...* Function

Use this function to write one file to tape; specify **mt:** as the *to* argument. Currently, you may not specify a new filename for the file on tape. **mt:** cannot be specified as the *from* argument; use **fs:restore-magtape** to accomplish this (See page 5.) Ellipses indicate keywords that exist but are rarely needed or used.

This function is considered obsolete for all purposes except tape applications.

fs:copy-files *files to ...* Function

This function takes the same options as **fs:fs-copy-file**. *files* should be a list of filenames to copy. Ellipses indicate keywords that exist but are rarely needed or used.

This function is considered obsolete for all purposes except tape applications.

fs:copy-directory *from to* &key *copy-only selective since* (*copy-subdirectories* **t**) Function

(This function, exists only in the magnetic tape software. It is like the regular system function **copy-file**, except that you have more control over which files are copied.) The arguments *from* and *to* can be any pathnames that name files; *to* takes its defaults from *from*.

If *copy-only* is supplied, its value names what versions of files will actually get copied. The supplied value, then, can be **:wild** (the default), **:newest**, **:oldest**, or a number. If *selective* is **t**, the user is queried about each file to be copied. If *since* is a date (meaning a string that can be parsed as a universal time), then only files with a creation date after that are copied. *to* can be **"mt:"** for copying to tape.

If *copy-subdirectories* is non-**nil**, then files in subdirectories are copied as well; the target directory for those files is a subdirectory under the target directory with the same name. In older versions of the software, *copy-subdirectories* would not propagate the exact specifications of *from* when recursing; as of Release 2 (System 99), it will take name, type, and version defaults from the superior directory unless the value of *copy-subdirectories* is **:wild**, in which case the old behavior will occur, namely, to copy *all* files in the contained subdirectories.

fs:magtape-list-files &optional (*out-stream* ***standard-output***) (*unit* **0**) Function

Lists the files dumped on the currently mounted tape, printing out onto *out-stream* the byte size, creation date, and file name. Structured directories are printed out as lists.

In the following four functions, the rest argument, *options*, gets passed to **fs:make-mt-file-stream** (see page 9).

fs:print-magtape &rest *options* Function

Copies the contents of the tape, until the **eof** marker, to ***standard-output***. This is useful for debugging and trying to figure out foreign tape header formats.

fs:copy-magtape-file *fn* &rest *options* Function

Like **print-magtape**, but sends the output to the file **fn** (which will be a character file). Somewhat like the UNIX command **dd**. No character set translation is done.

The following two functions do not use header conventions. They deal with unlabeled tapes using the ASCII character set. They are intended as quick ways to exchange files between a Lambda and another computer which is not accessible via the Chaosnet, but which does have a tape drive.

fs:print-ascii-magtape &rest *options* Function

Copies the contents of the tape until the **eof** to ***standard-output***, translating from ASCII to the LISP Machine character set.

To direct the output to a file:

```
(with-open-file (s file :direction :output)
   (let ((standard-output s)) ; directs standard-output to
     (fs:print-ascii-magtape))) ; this file "s".
```

fs:write-ascii-magtape *fn* &rest *options* Function

Writes file *fn* to tape; translating to the ASCII character set. No header is written.

1.3 Other Tape Formats

The following functions allow you to read tapes from other computers onto the Lambda. Currently the ANSI, and TOPS-20 formats are supported.

tops20:restore-tape &optional &key *to-host query tape-block-size file-opener* Function

Restores a tape written using the **dumper** program under the TOPS-20 operating system. *tape-block-size* is in units of PDP-10 words (5 bytes) and should usually be a multiple of 1024. *file-opener* is a function to receive a TOPS-20 file specification in the form of a string and return either NIL or a file object.

tops20:*default-tape-block-size* 5120 Variable

The default for the *:tape-block-size* argument for the above function.

tops20:*unhandled-file-types* ("QFASL" "EXE" "BIN") Variable

Filenames of these types are skipped over.

ansi:restore-tape &optional &key *file-id-append skip-if-exists query verbose* Function

This restores an ANSI labeled tape written according to American National Standard X3.27-1978. For example, the **copy** command under VAX/VMS produces such a tape. Directory name information is not preserved; therefore the *file-id-append* argument is provided which defaults to `"LM:TMP;"`. The value for *Verbose* is either **t**, **1**, **2**, or **3**. **t** and **1** provide a printout for each file restored; **2** provides information for each ANSI label and header processed; **3** gives information about each record processed. This last may be useful for debugging purposes.

ansi:*record-format-handlers* an a-list Variable

Presently the `"D"` (variable record) and `"F"` (fixed record) formats are handled.

2. Programming Information

This chapter describes more user functions that interface to the tape software; most of these functions are meant to be called from programs. There are functions for obtaining streams to the tape device, functions for controlling tape-specific operations, and condition names for handling tape-related errors.

The main interface for getting data to and from the tape is the use of *streams*, and the standard ZetaLISP stream functions. Special functions are used to obtain tape streams; the model of tape storage is not very compatible with a "file system" model of storage, so pathnames are not the recommended interface to tape. The user functions that deal with files (see the utilities that are documented starting on section 1.2, page 4) should provide most of the functionality needed for transfer between tape and file systems. If you really need to use the tape as a file system for an application, refer to section 2.1, page 9, which describes the conventions for storing files on tape.

2.1 Obtaining Streams

Streams made with the following two functions can be read or written with the standard buffered stream operations (see pages 476-77 of the *LMM*) for "record access". (The actual buffer objects returned are **RQBs**, a system structure normally used for disk I/O.) However, they are buffers all the same, and simple array functions such as **aref** can be applied to them. Of course, all standard stream operations work on these streams.

There are two functions for creating magtape streams. One, **fs:make-mt-stream**, is for using the tape in "raw" mode; the other, **fs:make-mt-file-stream**, creates streams that structure the data with headers so that **fs:restore-magtape** (see page 5) and other functions that use the tape as a file system can read the data back in. All the arguments to these functions are keyword arguments:

fs:make-mt-stream &key *direction characters byte-size* (*unit* **0**) *ibm-mode* (*record-size* ***default-record-size***) Function

Makes (and returns) a magtape stream that uses the tape without trying to use any file or operating system format – no header is written or parsed. The keyword arguments are the same as the function **fs:make-mt-file-stream** below.

fs:make-mt-file-stream &key *direction characters byte-size* (*unit* **0**) *ibm-mode* (*record-size* ***default-record-size***) *plist* (*format* **:mit**) Function

Makes (and returns) a magtape stream that acts like a file stream. It will have an associated pathname, author, creation-date, etc., if you pass that information along in the *plist* argument. If *characters* is **:default**, the value is determined from the plist passed in, and the **:byte-size** will default after checking the character argument, if it is not passed explicitly.

To really produce a file stream, callers should make sure the *plist* argument has **:directory**, **:name**, **:type**, **:version** (which should be a number), **:creation-date**, **:byte-size**, and **:characters** properties, though you may add more. (The **:author** property, for example, should usually be supplied as well.) If the **:byte-size** and **:characters** properties appear on the *plist*, then not passing those parameters as keyword arguments will do the right thing.

These are the keyword options used by the two functions discussed above.

:byte-size This can be a number (8 or 16 on the Lambda, also 1, 2, or 4 on the CADR) or **:default**, which is the default value.

The symbol **:default** means different things depending on the direction of data transfer. When *direction* is **:output**, it means 8 for a character stream and 16 for binary stream. When *direction* is **:input**, the actual byte size is the one stored in the property list in the header, if the function is **fs:make-mt-file-stream**. Otherwise, the default byte size will be 8 for a character stream, or 16 for a binary stream.

Currently, the decoding of Common LISP stream-element type arguments is not supported.

:direction This is either **:input** or **:output**. Probe opens, where the *direction* is **:probe** or **nil**, are not allowed and signal an error.

:characters This argument can be **t** or **nil** for **fs:make-mt-stream**. For that function, the default value is **t**.

An additional value accepted by **fs:make-mt-file-stream** is **:default**, which is the default value for that function. On input, using **:default** as the value causes the **:characters** property in the header plist on the tape to be used; on output, **:default** behaves the same as **t**.

:unit The tape unit number to use; usually 0.

:record-size

This variable controls how large the block size will be for the stream. It defaults to **fs:*default-record-size*** (see page 12).

:ibm-mode This keyword is used on the CADR only; it is ignored on the Lambda. By default, **nil**. An argument of **t** causes the "IBM mode" bit to get set on the Wesperco controller on write or read operations.

Both functions return the **fs:end-of-tape** error if the end of tape is reached. Although you are allowed to make streams in the **:output** direction on tapes with the write ring removed, the **fs:write-only-tape** error is signalled as soon as you try to write to the tape.

The actual flavor of the stream is determined by the *characters* and *direction* arguments. The device selected also is taken into consideration; on the CADR, this is always the same (the Unibus controller); on the Lambda, this is controlled by the selected tape device through the variable **fs:quarter-inch-tape-mode** (see page 12).

The current version of the software can read and write only file streams using **:mit** format, which is the type used by the LISP Machine. There are also some miscellaneous, unsupported functions to read other formats, but they do not present a stream interface.

The **:open** method for **mt-filehandle** (the equivalent of a tape pathname, but not as complex as a real file pathname) accepts the usual **open** keywords, as well as **:defaults-from-stream**, which, when supplied with a stream as the value, will put the relevant properties from that stream onto the *plist* argument of the function **fs:make-mt-file-stream.**

The **:close** operation on a magtape file input stream advances the tape to the next record after the **eof** mark. Thus, **fs:make-mt-file-stream** can be called again to get the next file.

2.2 Tape Movement

The following tape-drive controlling functions have default *unit* arguments of 0 and default *ntimes* arguments of 1 unless otherwise noted.

fs:mt-rewind &optional *unit* Function

Rewinds the tape mounted on *unit.*

fs:mt-offline &optional *unit* Function

Brings *unit* offline.

fs:mt-unload &optional *unit* Function

A synonym for **fs:mt-offline.**

The following six functions are usually called by programs that are reading or writing the tape when viewed as a stream of records.

fs:mt-space &optional *unit* (*ntimes* **1**) Function

Spaces forward *ntimes* record(s) on *unit.*

fs:mt-space-rev &optional *unit* (*ntimes* **1**) Function

Spaces back *ntimes* record(s) on *unit.*

fs:mt-space-to-eof &optional *unit* (*ntimes* **1**) Function

Spaces forward past the next filemark. If *ntimes* is more than 1, then that many files are skipped over.

fs:mt-space-rev-to-bof &optional *unit* (*ntimes* **0**) Function

Spaces back to the beginning of this file, or to the *ntimes*th file if *ntimes* is greater than 0. This function returns prematurely if the beginning of the tape is encountered before all the files are skipped.

fs:mt-space-to-append &optional *unit* Function

Searches for two **eof** marks and places the tape header over the second one, so that writing a stream will add more files to the end of the tape.

fs:mt-write-eof &optional *unit* Function

Writes an **eof** mark. After the last file on tape, there should be two **eof** marks; one is written when the MT file stream is closed; and the other must be written by the user (or the application program).

2.3 CIPRICO TAPEMASTER Specific Functions

fs:tapemaster-initialize Function

Initializes the TAPEMASTER controller. If you don't initialize the controller, tape commands will time out, and you will be put into the error handler.

fs:tm-print-unit-status &optional *unit* Function

Prints out the status of the tape drive *unit* connected to the controller.

2.4 Variables

fs:*default-record-size* Variable

Initially 10000 (octal). Old (pre-release 1.2) Lambda tapes were written with this variable set to 2000 (octal), but you can still read these tapes without changing the value.

fs:quarter-inch-tape-mode Variable

Determines which tape device will be used by generic functions. **t** represents the quarter-inch drive, **nil** the TAPEMASTER 1/2" drive.

This variable should not be set directly. Use the **fs:use-*type*-tape** functions (see page 4) to change the tape drive to use.

fs:*quart-paranoia-margining* Variable

Setting this variable to **t** will reset the drive before some operations to ensure the correct state of the device and in other cases to allow a **process-sleep** period so that the device can fully react to a given command and be ready for the next command. It should not be used unless problems occur. Call LMI first.

2.5 Conditions

fs:tape-error (error) Condition flavor

This is the flavor of condition that is signalled when there is trouble with a tape operation.

fs:end-of-tape (tape-error) Condition

Signifies end of tape. This is an especially useful condition name in user programs when used with **condition-case**.

fs:write-only-tape (tape-error) Condition

Signalled by the output operations of tape streams when the write ring is not there. This error is *not* signalled by the tape output stream making functions.

Both tape error conditions handle the following operations, in addition to standard error condition operations. These keywords are also init options to the **fs:tape-error** condition flavor.

:unit Returns the offending tape unit.

:rqb Returns the **RQB**. This is usually not needed.

:ibm-mode **t** if writing in IBM mode; applicable to the CADR only.

:command A number; the meaning is hardware-dependent.

:byte-count

Intended byte count.

:density Usually **nil**. **t** means high density. The exact meaning is hardware dependent.

2.6 LISP Machine Tape Format (LMFL, :MIT)

The format used by the function **fs:make-mt-file-stream** is very simple: The first four characters are "**LMFL**". Then immediately following is a property list, printed and read with the package bound to **fs:** and the base (number radix) bound to ten. Then there is some padding with spaces; the next block starts the actual data. The length of the first block should be 1024 bytes, but it may be different if the tape was written with old software (Magtape 14 or before). The end of file is signalled, as usual, with an **eof** block.

A typical header might look like this. (The example has been changed for readability; the newlines are not actually there.)

```
(:characters t :byte-size 8 :creation-date 54583923
 :directory ("RPK" "LM") :name "HOSTAB-SERVER" :type "LISP"
 :version 17 :length 6002 :AUTHOR "RpK")
```

The properties may come in any order. **Note** that the data on the tape is stored in eight-bit bytes. If the file on the tape consists of characters (i.e. when a non-null **:characters** property appears), then the characters are in the LISP Machine character set. (Refer to the discussion of the character set in the *LISP Machine Manual.*) If the data is binary, the bytes are packed in "little-endian" order. For a file with a **:byte-size** of 16 the low eight bits will be encountered first. For a binary file tape made on the CADR whose byte size is less than eight, the first nibbles are the least significant ones.

Here is a list of the defined properties that the system functions understand. Sometimes **nil** may appear as `"()"` in the tape header. Programs written for non-LISP Machines should interpret both `"()"` and `"NIL"` as **nil**. You may add your own properties to tapes that you write yourself.

:directory The directory of the file, a string or list, given in the internal format.

:name The name of the file, a string.

:type The type of the file, usually a string, but sometimes **nil** if the file was dumped from a file system that does not require a type component (like LMFL), to be present.

:version The version. This is always a number, even though there are other allowable values for versions in pathnames.

:characters Whether the file contains characters or not.

:byte-size The file's byte size. This should be 8 for text files, or a power of two between 1 and 16 inclusive for binary files. However, the Lambda does *not* support byte sizes of 1, 2, or 4 (the CADR does).

:creation-date
The creation date of a file as a universal time, a number.

:author The file's author, given as a string.

:length The length of the file in bytes, a number.

The stream's file properties should not be accessed via the **:get** operations, but instead should send the message directly to the stream:

```
(send mt-stream :get :directory)        ; Wrong
(get mt-stream :directory)              ; Wrong
(send mt-stream :directory)             ; Right
```

The following properties are used when the "file" is actually a disk partition. (Unlike with the properties above, you can use **:get**, even on the **:name** attribute.)

:partition **t** if a partition, either **t** or **nil**.

:comment Description of the contents, a string.

:name Original partition band on disk, a string.

:size Size in blocks.

2.6.1 Lambda/CADR Format Discrepancies

Header Block Size

In old software, before the Lambda, the header block was as big as it needed to be: this was related to the length of the printed representation of the plist, plus a few extra characters for **"LMFL"** and spaces for ensuring that the **read** function did not cause the next tape block (which contains the beginning of the actual file data) to be read in. When the software was adapted for the Lambda, it was discovered that the (half-inch) controller did not handle irregular block sizes gracefully. So, now the header block is 1024 (decimal) characters, padded with spaces. A standard header block usually has about 200 characters of "real" data, so user-added properties can be added safely. Just don't go overboard. ZetaLISP software will read the header block correctly no matter when it was written; so this information is of interest to those who want to write programs to handle tapes made on the LISP Machine to run on other machines.

Package Problems

All symbols in the plist are supposed to be in the keyword package, but various changes in the package system have complicated getting certain properties off the plist. Therefore, some property symbols in tape headers written by older software may not be preceded by colons. This occurs with the following symbols: **:byte-size**, **:directory**, and **:length**. The **:byte-size**, **:directory**, and **:length** methods for **mt-file-streams** compensate for the problem.

Characters

Earlier versions of the software did not write the **:characters** property to the header. On input, the function **fs:make-mt-file-stream** (with direction **:input**) assumes for such tapes that if the **:byte-size** is 8, then it's a character file.

This means that if an eight-bit binary file (usually a press file) has been mistakenly restored as a text file, it will not work correctly if the machine to which it was copied is based on the PDP-10; as a text file, it will get stored in seven-bit bytes. If such a file is restored to a system that had to be accessed through a network (any other machine besides the LISP Machine with the tape drive), the file will be stored in eight-bit bytes, but unfortunately the LISP Machine character set translation will have been applied to the bytes in the file.

However, if the file was restored locally (onto the disk), then the easiest solution is to:

1. Invoke the ZMACS command `META-X Dired` on the containing directory.
2. Invoke the **.** (period) command (`Dired Change File Properties`) on the file. A menu will pop up with various file system properties of the file.
3. Click on the word `Yes` on the `''Characters''` line of the menu.
4. Click on `''Do It''` on the bottom of the menu.

LISP Index

A

F

T