Dear LMI LAMBDA Customer:

We are very pleased that you have selected the LMI LAMBDA to satisfy your advanced computer needs.

We are confident that you will find it offers flexibility never before available in a system of this class. The ability to utilize both a powerful LISP-based processor and a very flexible 68000/UNIX-based processor in the same system allows users to combine the strengths of LISP and more traditional languages and architectures. You need not sacrifice performance by selecting only a LISP-based or a non-LISP-based processor.

LMI is committed to after-the-sale support. Hardware and software maintenance agreements are available on all LMI systems. Furthermore, we provide a "help line" to support our customers. The LMI help line number is:

(617)876-6819

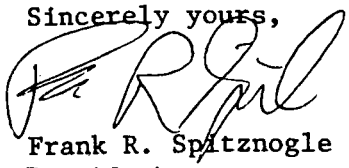
Simply ask for "the help line," and an LMI representative will be pleased to serve you in whatever way possible. This service is offered from 9:00 A.M. to 5:00 P.M. (Eastern time) Monday through Friday.

Although we trust that our maintenance and help line services will resolve the majority of your problems and questions, the following list of LMI contacts should prove useful if normal channels do not resolve matters to your satisfaction:

* Ms. Dawna Provost	Manager, Customer Support	(617)876-6819
* Dr. Nathan Dodge	Vice President, East Coast Operations	(617)876-6819
* Mr. Dennis Stevens	Director, Marketing Administration	(213)642-1116
* Dr. Frank Spitznogle	President, Chief Operating Officer	(213)642-1116 (office) (213)544-2663 (home)

I personally look forward to meeting you, and I am confident that you will be pleased to have selected LMI to satisfy your advanced computing requirements.

Sincerely yours,



Frank R. Spitznogle
President
Chief Operating Officer

Dear Customer:

With your new Lambda you will be receiving our revised set of LAMBDA USER DOCUMENTATION. According to your configuration, this should consist of from nine to twelve binders of documentation, organized to aid you in making the best use of your Lambda.

Among the new features of this documentation release are revisions of the INTRODUCTION TO THE LAMBDA, ZMACS INTRODUCTORY MANUAL, FIELD SERVICE MANUAL, and LAMBDA TECHNICAL SUMMARY. New material on CommonLISP can be found in the COMMON LISP NOTES, SYSTEM 99 NOTES, and LISP MACHINE MANUAL.

Within each binder you will find, just behind the striped Index Sheet, a System Map showing you the location of every manual in the system. These manuals are organized in binders under several subjects:

- Basics: This contains the manuals you should look at first. The LAMBDA TECHNICAL SUMMARY provides general technical specifications for each module of your Lambda. The LAMBDA FIELD SERVICE MANUAL guides you through site preparation, installation procedures, power-up, booting, and user debugging procedures for the Lambda. The INTRODUCTION TO THE LAMBDA gives programmers a basic introduction to the functions and support mechanisms of machine.

- System Notes: These manuals document system changes for System 94, LMI's last pre-CommonLISP system, and System 98, our first release incorporating CommonLISP. CommonLISP is discussed in a separate set of notes.

- LISP Manuals: These three binders contain the LISP MACHINE MANUAL (System 99, with CommonLISP), together with the WINDOW SYSTEM MANUAL and the INTERLISP and LM-PROLOG manuals. (These last manuals will be included if you have chosen INTERLISP or PROLOG as an option on your Lambda.)

- Editors: Our new ZMACS INTRODUCTORY MANUAL introduces this binder. The complete ZMACS REFERENCE MANUAL will follow it in early first quarter 1985. MINCE and SCRIBBLE, a ZMACS-like UNIX-based editor and associated SCRIBE-like formatter, are further options.

- UNIX: The first of these two binders consists of UNIX RELEASE AND UPDATE INFORMATION, NUMACHINE OPERATING SYSTEM information, and Volume 1 of the UNIX PROGRAMMER'S MANUAL, the basic guide to the UNIX system. The second binder

contains Volumes 2A and 2B of the UNIX PROGRAMMER'S MANUAL. All these are current for UNIX Distribution 3.

- Hardware notes: Three binders contain (in Volume 1) information on the hardware configuration of the Lambda, at a higher level than that in the Basics binder; (in Volume 2) disk drive and controller manuals; and (in Volume 3) tape drive and controller manuals.

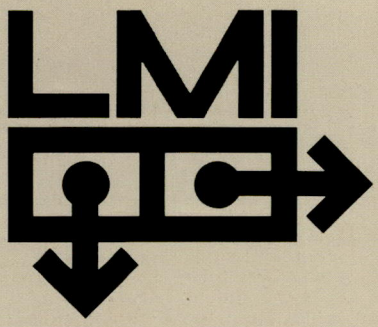
- Options: The contents of this binder will depend on options you have chosen.

We at LMI Documentation believe this new organization will help you use your Lambda more efficiently and pleasurably. We plan further changes to this documentation system, and always welcome your suggestions and comments. Please write to me at LMI, or, for faster service, send me mail through your customer account on the VAX (username SWRS).

Sincerely,

Sarah SmithKH

(Dr.) Sarah Smith
Director, Documentation
LMI



LMI Lambda Technical Summary

LAMBDA

LMI Lambda Field Service Manual

NuMachine Installation and User Manual

Introduction to the Lambda

Programming on the Lambda

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

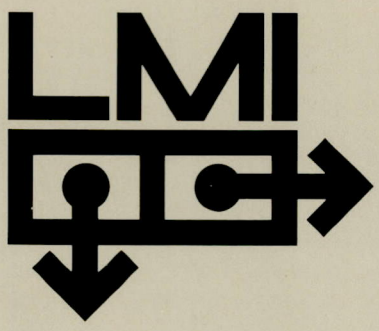
UNIX 1:
- **NuMachine Release and Update Information
- **NuMachine Operating System
- **UNIX Programmer's Manual, V. 1: Section 1
- ** Sections 2-8

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
  - Motion Tek Color System
  - MTI System



LAMBDA

THE LMI LAMBDA

TECHNICAL SUMMARY

The LMI Lambda: Technical Summary

6033 W. Century Blvd.
Suite 900
Los Angeles CA 90045 USA

Principal writer of this manual was Sarah Smith. Portions of the manual dealing with NuMachine architecture have been adapted by permission from material prepared by the NuMachine Development Group at Texas Instruments.

Table of Contents

Chapter 1

Introduction

This technical summary is the first in a series of manuals dealing with the hardware and software components of the LMI Lambda. It is designed for programmers, system designers, engineers and other professionals investigating LISP development systems and applications for business and research. We assume that the reader is familiar with computer hardware, software and terminology, but not necessarily familiar with LISP and the artificial intelligence field.

This manual will also be of interest to those interested in integrating LISP and UNIX, since the LMI Lambda is unique among LISP machines in its multiprocessor bus and consequent ability to support concurrently executing LISP and UNIX environments. As will be discussed below, this ability allows the evolutionary introduction of "intelligence" into a traditional computing environment without requiring an all-at-once changeover.

Chapter 2 briefly covers the history of LISP and LISP machines and the genesis of the LMI Lambda, while Chapter 3 is a summary overview of the most significant features of the machine. Chapter 4 covers the LMI Lambda's technical specifications. Chapters 5 through 11 address the hardware components of the LMI Lambda in more detail, and Chapters 12 through 15 cover the software furnished with or available for the system.

More complete technical information on many of the subjects discussed in this summary is available from other manuals in the series:

— Software

* *Introduction to ZetaLISP-Plus: Operations Manual*
* *ZMACS Introductory Manual*
* *The Window System*
* *ZMAIL Manual*
* *LISP Machine Manual*
* A revised version of the *LISP Machine Manual*, including CommonLISP compatibility, will be available in second quarter 1984.

— Hardware

* *The LMI Lambda Field Service Manual*

To order any of the above manuals, or for additional information on any subject covered in this technical summary, please contact:

LMI
6033 W. Century Blvd. Suite 900
Los Angeles CA 90045 USA
Telephone (213) 642-1116 Tlx 664 608

Chapter 2

Design Philosophy of the LMI Lambda

The design philosophy of the LMI Lambda has been to create a LISP-centered multiprocessor environment, capable both of running the LISP language efficiently and of interacting with other research and business programming environments.

2.1 Why LISP?

Developed in 1958 by John McCarthy as a mathematical processing language, LISP has been the language of choice in the artificial intelligence field for many years. It has evolved into a language that can deal with complex and unpredictable data that traditional programming techniques cannot handle. A very powerful set of editing and debugging tools has been developed for programming in the language. As a result, large, complex programs can be written, tested, and modified much more easily with LISP than with any other programming language. In addition, its freely extensible structure allows it to be used as the basis of higher-level languages designed to address application-specific problems. Typical applications utilizing LISP include expert systems, CAD (especially VLSI and other highly complex systems), natural language processing, intelligent databases, robotics, and many others.

However, until recently, anyone wishing to take advantage of LISP's potential had to be prepared to compromise. Before the advent of LISP machines specifically designed to execute the language efficiently, LISP implementations ran in traditional mainframes. Since these computers were optimized for numerical languages such as Fortran, much of the LISP environment was in software, imposing a great deal of overhead on the execution of LISP programs. As a result, the LISP language remained in the research lab, where functionality, rather than speed, was the major consideration.

2.2 The Development of LISP Machines

One of the most fruitful sources of innovation in the computing world has been the Artificial Intelligence Laboratory at MIT, which has contributed a great deal to the development of such concepts as timesharing, networking and other now widely-used techniques. Its multi-million-dollar research program is sponsored by a variety of governmental agencies and corporate donors who support the laboratory's overall goal of researching and developing advanced computer technologies.

A LISP Machine project was started at the MIT AI Laboratory in 1973 by a group that included Richard Greenblatt, one of the founders of LMI. The project goal was to develop a machine capable of running LISP efficiently. The first LISP machine, the CONS, was developed in 1976, but it required the support of a PDP-11 for I/O and memory. The project's second effort, named the CADR (after a LISP function that returns the second element of a list), was developed in 1977. It was the first stand-alone LISP machine.

In September of 1980, Richard Greenblatt and F. Stephen Wyle founded LMI to make the CADR available to the AI community at large and to introduce LISP machine technology to the commercial marketplace. The

company shipped its first CADR in May of 1981. The CADR is a personal, networked computer for programmers developing large and complex software systems. It has performed effectively in research and development labs in the United States and in numerous foreign countries.

2.3 A New Computer Architecture

Contemporary with the LISP Machine project in its later stages, a project group at the MIT Laboratory for Computer Science was developing a new computer architecture. Unable to find a solution to their future computing needs in existing technology, they planned a new architecture design characterized by flexibility and expandability: an architecture that would remain viable through the 1990s. The resulting architecture, which is now known as the NuBus, formed the nucleus of a revolutionary design called the NuMachine. The NuMachine was taken to prototype stage by engineers at Western Digital and was then acquired by Texas Instruments, which now markets and supports it.

LMI recognized that the NuBus offered a solution to some of the problems that still stood in the way of the commercialization of LISP. Despite the power and productivity of personal LISP machines such as the CADR, these machines were still not flexible and adaptable enough to serve the needs of most industries. They did not offer a wide range of configurations, nor could they easily accommodate advances in computer technology. In addition, they did not integrate easily with existing, traditional systems and the software running on them. What was needed was a LISP machine based on a new architecture: the LMI Lambda.

2.4 The LMI Lambda: The Concept of Modularity

Development of this revolutionary new LISP Machine architecture began late in 1981, and LMI shipped its first LMI Lambda in September of 1983. This machine draws heavily on the design experience gained from the CADR, while adding numerous LISP-oriented enhancements in the framework of the NuBus. But it is also the first LISP machine fully integrated into a multiprocessor environment. Its combination of a LISP processor with the NuBus architecture offers a modular, expandable LISP machine with multiprocessor capabilities, Multibus compatibility, and Ethernet-II networking.

The LMI Lambda's multiprocessor capability allows it to execute LISP and traditional software concurrently in independent, optimal environments, with full communication between processes. Its integral Multibus gives the user or system integrator a wide choice of peripherals and board-level accessories from over 100 manufacturers, so it can be configured to suit the needs of a specific application; and with its optional Ethernet-II interface the LMI Lambda can share files with other systems. Its unique virtual control store and LISP microcompiler allow the LMI Lambda to be easily and rapidly conformed to a specific application, for increased speed and efficiency—ideal both for rapid prototyping and for program optimization. And the LMI Lambda's ZetaLISP-Plus software system offers the most powerful programming environment available today, while maintaining full CommonLISP compatibility.

All of these features make the LMI Lambda the machine of choice for anyone interested in applying the power and productivity of LISP to their industry. The open-ended design of its NuBus architecture allows the incorporation of advances in computing technology as they become available, thus ensuring a long product life.

Chapter 3

The LMI Lambda System: Design Overview

Modularity is central to the design philosophy of the LMI Lambda. The system is designed so that it can exist in a large number of configurations around a 32-bit, device-independent bus, thus yielding a computer system capable of addressing a very large range of applications.

3.1 The NuBus

At the heart of the LMI Lambda's modularity is the NuBus. The NuBus is a device-independent architecture, centered on a high-speed 32-bit synchronous bus with a peak transfer rate of 37.5 Mbytes/second. Unlike traditional bus designs, which are centered on a specific processor, the NuBus is a communications-centered design, allowing the rapid interchange of data between a variety of devices within its 4-gigabyte logical address space. Each board in the NuBus is designed for a specific function, thus maintaining the system's modularity. This design provides the ideal framework for systems that require multiple general-purpose or special-purpose processors. Figure 3-1 illustrates the difference between the NuBus and traditional designs.

By supporting 32-bit addressing and 32-bit data transfers, the NuBus provides an environment markedly superior to competing 68000, 8086, 80286 and Z80000 microprocessor-based systems, all of which are primarily 16-bit systems. When a true 32-bit CPU, such as the MC68020, becomes available, competing 16-bit designs will require a complete system redesign. In contrast, each installed LMI Lambda will be able to accept a new 32-bit processor card without any changes to the rest of the system: bus, memory, peripheral controllers and so forth will remain the same. In addition, the LMI Lambda can continue to use the original processor side-by-side with the new one, making the changeover much simpler and less disruptive.

3.2 The System Diagnostic Unit

A key component of the LMI Lambda's architectural flexibility is the System Diagnostic Unit, an 8088-based multi-tasking monitor that serves as both an architectural supervisor and a smart diagnostic front end. Upon power-up, the SDU verifies bus and board integrity, identifies the installed modules and configures the system accordingly, then signals the presence of any defective modules and boots the system. A slot identification scheme eliminates the jumpers and DIP switches traditionally used for address or device number assignment. Two RS232 ports can serve as general-purpose serial ports or for remote diagnostics.

The System Diagnostic Unit also accounts for a great deal of the reliability and ease of servicing of the LMI Lambda. Most failures can be diagnosed by the SDU for rapid, on-site repair; the diagnostic capabilities of the SDU include bus-clock margining and power supply margining. The use of highly reliable DIN connectors, proper component cooling, and attachment of cables to the backplane, combined with the avoidance of jumpers, DIP switches or special backplane wiring, also help make the LMI Lambda easy to maintain.

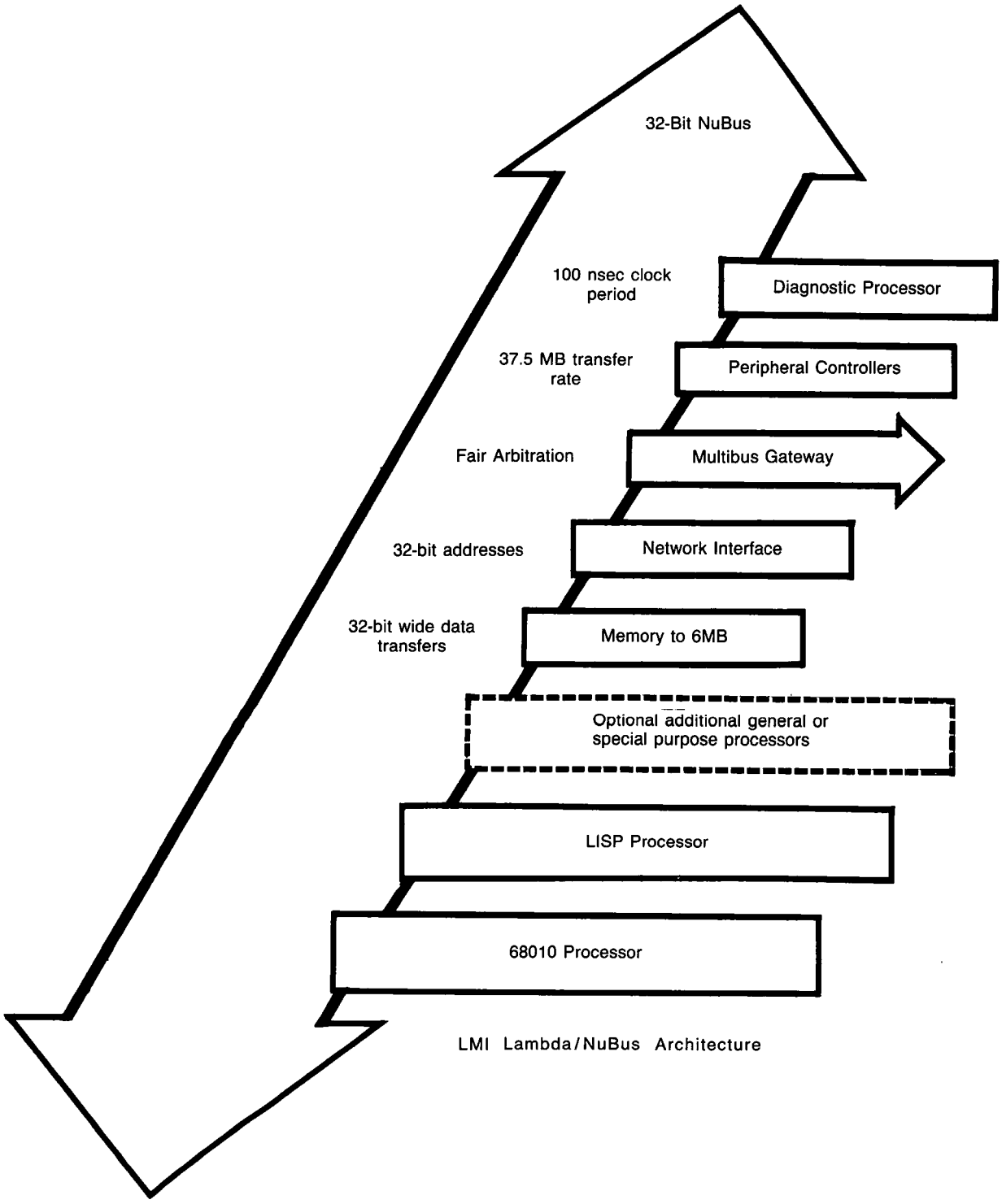


Figure 3-1. NuBus Architecture.

3.3 Multibus Compatibility

The versatility of the LMI Lambda is further enhanced by the system's integral Multibus interface, which gives the user or system integrator access to over 1000 peripherals and board-level accessories from over 100 different manufacturers. This makes it easy to configure the LMI Lambda to fit a specific application, at a lower cost than with other LISP machines.

The LMI Lambda's System Diagnostic Unit serves as the interface between the NuBus and the Multibus. The two busses operate independently except during bus conversions, which are accomplished by a hardware-mapping scheme that requires no participation by the SDU's processor. The entire one-megabyte address space of the Multibus appears as one continuous block in the 4-gigabyte NuBus address space; the conversion from Multibus to NuBus is so transparent that a NuBus processor can access data or even execute a program out of Multibus memory. In the other direction, conversion is accomplished by a page mapping scheme. A unique feature prevents deadly embrace if both busses request each other at the same time. Interrupts from one bus to the other are also handled by the SDU.

The LMI Lambda Multibus is supplied with a SMD disk controller capable of handling four disk drives, and also holds the optional Ethernet-II and magnetic tape interfaces.

3.4 Multiprocessor Capabilities

The LMI Lambda's NuBus architecture gives it something that no other LISP machine offers—true multiprocessor capabilities. The device-independent nature of the NuBus enables the LMI Lambda to accommodate a range of processor combinations, all executing independently and concurrently, with full communication between them.

A typical configuration includes a LISP processor and a UNIX processor. This configuration, in combination with LMI's exclusive STREAMS interface, offers users the ability to add intelligence to an existing, traditional software package, operating in UNIX, by placing it under the supervision of an evolving LISP program. For example, a database too large for manipulation by traditional means can be augmented by a LISP-based program that adds natural language and intelligent search capabilities, without disrupting use of the database.

3.5 The LISP Processor

The heart of the LMI Lambda is its LISP processor, which is a highly specialized microprogrammable processor designed for efficient pipelined execution of complex order codes. As supplied, it is optimized for the efficient 16-bit order code produced by the LISP Machine LISP compiler. The processor features an industry first, a virtual control store, which in combination with the LMI LISP microcompiler enables the machine architecture to be easily modified to suit a specific application. This results in increased speed and efficiency without the expense and difficulty associated with traditional microcode delivery. In addition, these features allow the machine to be easily optimized for higher-level, application-specific extensions of LISP and furnish an excellent facility for prototyping advanced computer architectures.

3.6 LISP Software

LMI's ZetaLISP-Plus furnishes an extensive array of software development tools that greatly increase programmer productivity. The modular nature of the LISP software means that any of its elements can be easily incorporated into applications software, freeing programmers to concentrate on application-specific design. Since the entire operating system of the LMI Lambda is written in LISP, it is easily modified or extended to suit specialized needs. In addition, the LMI Lambda supports CommonLISP, which facilitates the interchange of programs between different LISP environments.

Some of the programming tools supplied with the LMI Lambda include:

- ZMACS—A screen-oriented, real-time editor, which is fully integrated into the global LISP environment and uses a mouse and the LMI Window System for increased editing speed.

- The LMI Window System—A powerful utility that allows the creation and placement of separate virtual screens of arbitrary size anywhere on the display. Each window may have a different process associated with it.

- The Inspector—A powerful graphically-oriented tool that enables programmers to examine and modify complex structures with ease and convenience.

- The Window Error Handler—A graphically-oriented display of the functional environment involved in an error.

- FLAVORS—A message-passing extension of LISP akin to SMALLTALK, but more versatile. FLAVORS is a way of constructing abstract objects that bundle up procedural and declarative information about the thing represented by the object.

- CommonLISP—The new industry-standard LISP dialect. LMI's CommonLISP implements the entire CommonLISP subset as defined by the CommonLISP Group and is integrated completely within the ZetaLISP-Plus software system. This facilitates the exchange of software between LMI software sites and other sites running CommonLISP. LMI's CommonLISP includes upward lexical closures and both rational and complex numbers.

- The LMI File System—Providing LISP-like file handling even for remote, non-LISP hosts.

- ZMAIL—A communications system enhancing intra- and inter-system communication.

In addition, a very powerful version of PROLOG is available for the LMI Lambda, featuring high-speed execution through custom microcode, as well as a number of versatile control facilities not found in other versions.

3.7 Optional UNIX Processor

The optional UNIX processor for the LMI Lambda is a 68010-based processor operating at 10 MHz and capable of peak speeds of 1 MIPS. Its demand-paging virtual memory implementation generates 24-bit virtual addresses, resulting in 16-megabyte address spaces for each process in the multitasking, Berkeley-equivalent UNIX environment. The processor also features a 4 Kbyte write-through cache, 32-bit arithmetic, and byte addressability.

3.8 UNIX Software

UNIX is a highly sophisticated operating system with many powerful functions, including directory hierarchies, the Bourne shell and C shell, file-like devices, a window system, processes, and pipes. It provides a flexible programming environment that interfaces easily with the LISP environment of the LMI Lambda, and is ideal for applications supporting multiple users with either dumb or intelligent terminals. In addition, UNIX has been recognized for some time as one of the finest environments for the preparation and publication of technical documents. Major features of the current UNIX system for the LMI Lambda include:

- A large file system

- A new, special file type called FIFOs, which behave like pipes but have names and can be used to pass data between multiple, non-related processes.

- An enhanced C-compiler

— The MINCE screen-oriented editor

— Machine-to-machine spooled file transfer capability

The languages currently supported include C and Fortran 77; Pascal will be available in third quarter 1984.

3.9 Extended STREAMS Interface

The LISP and UNIX processors execute concurrently and independently, but programs can communicate between environments via LMI's unique extended STREAMS interface, which links the process communication primitives of LISP and UNIX. A dual-processor LMI Lambda can utilize the UNIX processor as a multitasking, multi-user front-end to package and send requests to the LISP processor, or as an environment for existing, traditional programs (such as process controllers or large databases) under the supervision of an evolving, intelligent LISP program.

3.10 Networking

The LMI Lambda network capabilities provide dedicated computing resources under the direct control of the user, while also facilitating inter-user communication and resource-sharing.

The optional Ethernet-II interface provides an industry-standard Local Area Network operating at 10 megabits/second. It is possible to specify interfaces to the mainframes of additional LMI Lambda systems as well as to those of other selected vendors, such as DEC.

3.11 High Resolution Graphics

The LMI Lambda's video display is an 800 × 1024 pixel, bit-mapped, non-interlaced system with dual one-megabyte video buffers (useful when screen updates should appear to be instantaneous), a 1/60 second refresh rate, and onboard ALU support for logic functions. The CRT is a vertically mounted 15-inch, small-footprint design with a very readable display. The video display's many onboard functions furnish increased speed and performance. The LMI Lambda can support additional displays—B&W, medium-resolution color with optional frame-grabber, or high-resolution color.

3.12 Keyboard and Mouse

The 100-key AI keyboard furnishes a superset of ASCII adapted to LISP programming needs, and has unlimited rollover: any key depressed, in no matter what order, is sensed. The keyboard offers a choice of standard character sets, including alphanumeric, Greek, and a third character set that can be bound to functions selected by the user. All keys are fully reprogrammable, so that the machine can be customized to interpret the keyboard in any fashion. Four "control" keys (CONTROL, META, HYPER and SUPER) allow further functions to be bound to individual keystrokes.

The three-button optical mouse and mouse pad are fully integrated with the keyboard; combination mouse-and-keyboard bindings are allowed, and many frequently used functions can be performed with the mouse alone. Selection of windows, buffers, or files, marking and moving, and graphics are among the functions that are facilitated with the mouse. With no moving parts, the optical mouse offers greater reliability and sensitivity than its mechanical predecessors, enhancing programming efficiency.

Chapter 4

Product Specifications

Two versions of the LMI Lambda are available, which offer the same LISP processor in different physical packages. The LMI Lambda/S is the smaller of the two. It has an ultra-quiet fan and a smaller card cage, and is intended for office use where a low noise level and small size are important. It is identical to the full-size LMI Lambda except for the smaller card cage.

Overall, three configurations of the LMI Lambda are available from LMI.

Lambda	32-bit processor, 169-megabyte disk
Lambda	32-bit processor, 470-megabyte disk
Lambda/S	32-bit processor, 169-megabyte disk

4.1 System Specifications

Card Cage

LMI Lambda	21 slots — 13 NuBus, 5 MULTIBUS, 3 slots either
LMI Lambda/S	12 slots — 6 NuBus, 3 MULTIBUS, 3 slots either

Cabinet

LMI Lambda	19-inch rack mount — 60″ × 35″ × 22.5″
LMI Lambda/S	Office cabinet — 26″ × 21″ × 22″

Power

LMI Lambda	110 VAC, 30 Amps., 60 Hz (foreign versions available)
LMI Lambda/S	110 VAC, 12 Amps., 60 Hz (foreign versions available)

Heat dissipation

LMI Lambda	Approximately 6400 BTU
LMI Lambda/S	Approximately 2500 BTU

Power consumption and heat dissipation are dependent on configuration. The figures given are representative.

Environment	55 to 85 degrees F, 20 to 80% humidity, non-condensing.
System Console	800 x 1024 pixel B&W, AI keyboard, 3-button optical mouse, operational up to 150 feet from mainframe. (High and medium resolution color optional)
Disk	
	LMI Lambda 169- or 470-megabyte Winchester
	LMI Lambda/S 169-megabyte Winchester
	SMD controller will control up to 4 disks, removable media disks are also available.
Tape	Optional
	— 1/2-inch streamer
	— 1/4-inch streamer
	— Industry-standard reel-to-reel
Memory	300 nanosecond peak transfer rate, 256K-word standard

Chapter 5

System Diagnostic Unit

The System Diagnostic Unit (SDU) is present in all NuBus systems to provide many important one-per-system functions and "smart" front-end and diagnostic capabilities. By concentrating these functions on the SDU rather than on separate CPU cards, the system is able to support multiple processors without conflict.

On power-up, the SDU verifies the integrity of the bus through a series of bus transfer tests. The SDU then identifies all boards within its system environment; initiates and monitors diagnostic and self-test routines to ensure that the boards are functioning properly; signals the operator, using an on-board serial I/O facility, if any boards have malfunctioned; and initiates the system bootstrap program.

The SDU also serves as the NuBus-to-Multibus converter. Facilities are provided for mapping Multibus cycles into NuBus cycles and vice versa; for mapping Multibus interrupts into NuBus events; and for generating Multibus interrupts from the NuBus.

The SDU consists of an Intel 8088 microprocessor with on-board memory, two serial I/O ports for communication with the operator and peripherals, and many other system maintenance and diagnostic features. A functional block diagram is shown on the following page (Figure 5-1).

5.1 Main Features of the SDU

The main features of the SDU are:

- System Clock: The SDU is the source of the 75% duty cycle, 10MHz System Clock (CLK/),[1] to which all bus operations are synchronized. The system clock rate can be varied by program, up or down by 10%.

- Timeout Recovery: The SDU provides NuBus timeout recovery by monitoring the time between the START/ and ACK/ control signals. If more than 32 clock cycles occur, the SDU asserts the ACK/ signal with the appropriate TM1 and TM0 code for a timeout.

- Nonvolatile Features: The SDU contains 2 Kbytes of battery backed-up CMOS RAM. This memory is used to store the system configuration information in a nonvolatile manner. The SDU also provides a battery backed-up time-of-day clock.

- Interval Timer: The SDU contains a programmable timer for generating periodic events to specific CPUs. The timer may be used for many important system functions, such as process scheduling.

1. Negative true signals are indicated by '/', as in CLK/.

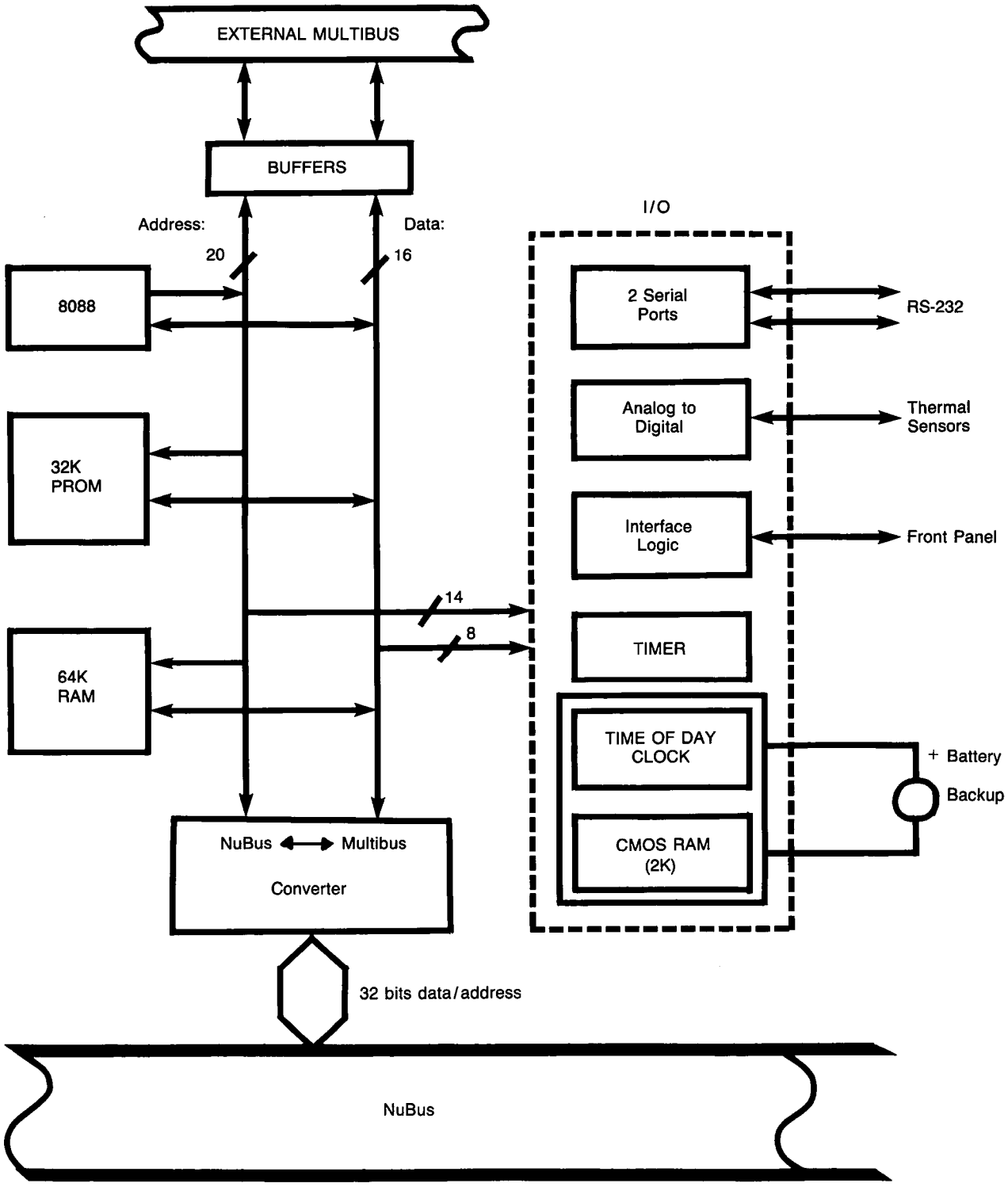


Figure 5-1. System Diagnostic Unit Block Diagram.

— Debug/Diagnostic Facilities: The SDU provides the hardware maintainer of the LMI Lambda system with several diagnostic tools. A monitor in the SDU PROM allows either serial port to be used to execute diagnostics as well as to read and write bus locations in order to initiate SDU self-tests.

— System Status Display: On power-up, the SDU first runs a self test, next a bus test, and then individual board tests. The SDU uses the front panel LEDs to summarize the test results; the detailed results may be read using the system console. An LED on each board is automatically turned on at power-up and turned off by the SDU if that board passes power-up diagnostics. A board with its LED on is therefore presumed bad.

— Serial Port: The SDU contains two serial communications ports, either of which may be used as the smart front panel/remote diagnostics port, depending on the position of the diagnostic rotary switch. Otherwise, they are both available as general-purpose serial ports.

— Power Loss Detection: The actual detection of power failure is done by the power supply, which signals this occurrence to the SDU. The SDU then posts "events" to the installed CPUs to let them know that the power will soon be off.

— System Boot: The NuBus system boots automatically on power-up by action taken by the SDU. However, if for any reason program control is lost or undefined, the system may be reinitialized manually from the SDU. A system halt, reset, and reboot may be initiated from the console serial port.

— NuBus/Multibus Interface: The Multibus interface is transparent in the sense that it translates NuBus transactions and Multibus transactions into each other. The interface is not "intelligent" but rather implements a direct mapping of transfers from one bus to the other. DMA transfers in both directions are supported.

Chapter 6

The NuBus

The NuBus interface is the flexible bus structure used to interconnect the family of processors and other hardware functionalities on the LMI Lambda, including the LISP and 68010 processors, memory boards, video interfaces, network interfaces, and peripheral controllers. It supports direct addressability of over four gigabytes (4×10^9 bytes) through byte-oriented 32-bit addresses, block transfers of multiple words, and 8-, 16-, and 32-bit data transfers.

The bus structure is built upon the master-slave concept, in which the master device in the system takes control of the NuBus interface and the slave device, upon decoding its address, acts upon the command provided by the master. Any module that decodes the address line and acts on the commands from the bus master is a bus slave. This handshake between master and slave devices allows modules of different speeds to use the NuBus interface and allows fast peak data transfer rates of 37.5 Mbytes per second.

Any module on the NuBus can interrupt a processor module by writing into an area of address space that is monitored by the processor. This "event" mechanism is an important aspect of multiprocessor support. Interrupt lines are eliminated, allowing the priority of the interrupt to be software-specified by memory-mapping the priority level.

Another important NuBus feature is the ability to connect multiple master modules for multiprocessing configurations. The NuBus interface provides control mechanisms for connecting multiple masters in a "fair" priority fashion.

In summary, the NuBus features:

- 32 bits of byte-oriented address
- 32 bits of data, multiplexed on address lines
- Simple handshake protocol synchronized to clock cycles
- Bus arbitration mechanism allowing fair bus bandwidth sharing, able to take place in parallel with data transfer operations
- Event mechanism supporting multiprocessing, eliminating interrupt lines—basically a memory-mapped interrupt scheme
- Increased effective bus bandwidth through block transfers
- Division of address space by slot position rather than by jumpers or switches

— Single 96-pin DIN connector allowing for a very small form factor and electrically compatible systems in the future

— Address and data transfer integrity protectable by parity logic

— "Bus glomming" to minimize bus arbitration.

The NuBus supports the unified bus interconnection of up to 16 system modules. The motherboard presents each board location with a unique identification number. Any system module can occupy any board location; thus it is not necessary to define address space on individual cards by jumpers or switches. No special backplane wiring is needed.

6.1 NuBus Signals

There are five classes of NuBus signals, divided according to function performed. They are:

Card Slot Identification

4 signals; assigns the physical location to each module.

Control

6 signals; performs all control functions. These are:

— Clock (CLK/)

— Reset (RESET/)

— Start (START/), which signals the beginning of data transfer

— Transfer Acknowledge (ACK/), indicating the end of data transfer

— Transfer Mode 0 and 1 (TM0/ and TM1/), encoded by the current bus master to indicate type of transfer.

Address/Data

32 signals; carries 32-bit address at beginning of cycle and 32 bits of data within remainder of cycle.

Bus Arbitration

5 signals; regulates bus arbitration.

Parity

2 signals; System Parity (SP/) transmits parity information between cards implementing NuBus parity checking.

System Parity Valid (SPV/) indicates whether system parity is valid or not for that transaction.

The NuBus also includes conductors that carry power and ground signals. Voltages available are +5, −5, +12 and −12. Power and ground are arranged around active signals to minimize crosstalk.

6.2 Data Transfer Operations

Data transfer on the NuBus is accomplished through a synchronous master/slave protocol. Bus transactions are synchronous to the system clock and their durations are multiples of the clock period. The protocol uses two handshake signals to coordinate the transfer: START/, generated by the current bus master, and either TM0/ or ACK/, generated by the slave.

Information is placed on the bus synchronous with the rising (assertion) edge and is sampled on the falling (sample) edge of the clock cycle. This technique provides protection from race conditions caused by bus transmission skews.

6.3 Read Transactions

Read operations with data widths of 8, 16, and 32 bits are selected by the transfer mode lines (TMx) and the two low-order address lines. Whatever processor is the current bus master is responsible for selecting the appropriate byte or halfword for internal use if all 32 bits are not relevant.

6.4 Write Transactions

Write operations with data widths of 8, 16, and 32 bits are selected by the transfer mode lines (TMx) and the two low-order address lines. Bytes, halfwords, and words are organized as follows:

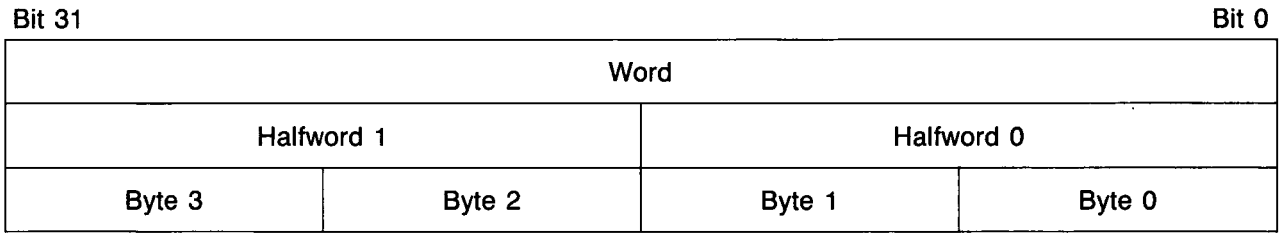


Figure 6-1. Layout of Words, Halfwords, Bytes.

The current bus master has the responsibility of aligning the data to be written onto the appropriate ADx/ lines for halfword and byte writes. For example, a write byte 3 requires that the data be placed on AD24/ through AD31/; all other ADx/ lines are not defined and can be driven to any state.

6.5 Block Read

Block reads on the NuBus consist of a single command/address transfer initiated by the current bus master, followed by n 32-bit data outputs from sequential ascending locations of the addressed slave. n is equal to 2-, 4-, 8-, or 16-word blocks and is determined by AD2/, AD3/, AD4/, and AD5/ on the START/ cycle.

6.6 Block Write

Block writes on the NuBus are similar to block reads except the TM1/ (read/write) is driven to the write state during the START/ cycle, and the current bus master drives the data bus while the slave accepts data.

6.7 Error Acknowledgment

The two TMx/ lines are encoded at ACK time to indicate whether or not the bus transfer was successful. Four states are possible:

ACK	Successful completion of transfer.
NAK	Unsuccessful completion of transfer because of an error condition.
Timeout	Unsuccessful completion of transfer because 32 bus cycles elapsed while the bus was "busy."
Try Again Later	Unsuccessful completion of transfer but no error condition exists; the master should rearbitrate for the bus and try again.

6.8 Multiprocessor Support

The NuBus supports multiprocessors through the bus arbitration mechanism, facilitated by the "memory-mapped" interrupt mechanism.

6.9 NuBus Arbitration Mechanism

Arbitration for control of the NuBus takes place independent of data transfers.

Arbitration takes place each time control is transferred between bus masters. The winner of the arbitration contest takes control of the bus and retains control until an arbitration contest is won by another bus master. Actual transfer of control does not occur until the current bus master completes any data transfer in progress.

The NuBus provides fair bandwidth sharing between processors in a multiprocessor system. The sharing is accomplished by a bus convention among processors. This bus convention is:

> If several devices request the bus at the same time, they are given the bus in priority order, highest to lowest. Fairness is provided by the rule that no new bus requests can originate from any device, including those in this group, until all devices in the group have acquired the bus.

This rule guarantees that processors of higher physical priority do not starve a processor of lower priority.

Once a bus master has acquired the NuBus, it is by definition the highest-priority bus master within the group of modules that requested the bus. An undivided set of data transfers, such as in the operation "test-and-set," can be accomplished by the bus master continually arbitrating for, and winning, the bus.

6.9.1 "Bus Glomming"

An arbitration contest takes place only when control of the bus must be transferred between bus masters. Therefore, if no other processor has requested the bus, the current bus master may initiate data transfers without first rearbitrating for use of the bus. This capability relieves the current bus master from the overhead of arbitrating for use of an idle bus and is termed "bus glomming."

6.10 Interrupts

The NuBus accomplishes interrupts through the write transaction operation. No unique lines or protocols are required for interrupt operations. Any module on the NuBus can interrupt a processor module by performing a write operation into an area of memory space that is monitored by the processor. The address used to post the interrupt can be at any location in the address space. This general technique allows interrupts to be posted for individual processors in a multiprocessor system and allows the priority of the interrupt to be software-specified by memory-mapping the priority level.

6.11 NuBus Electrical Characteristics

- Tri-state bus with 48-milliamp drivers.
- Terminated bus using a controlled impedance strip-line backplane.

Chapter 7

Multibus Interface

Although the LMI Lambda is designed around the high-bandwidth NuBus, the Multibus (IEEE-796) is also part of the system. Incorporating both bus structures into the system solves the limitations of offering only one or the other of the buses. If only the Multibus had been used, NuBus advantages such as flexibility, speed, expandability and multiprocessing would have been lost; if only the NuBus had been used, too few peripheral options would have been available.

With the Multibus, customers have access to approximately 1000 Multibus-compatible, board-level products from over 100 manufacturers. Your system can interface to the peripherals that are best for a given application.

The two busses operate independently of one another except during a conversion. Bus conversions occur at high speeds.

7.1 NuBus/Multibus Conversion

The NuBus/Multibus converter resides on the System Diagnostic Unit (SDU). (See Chapter 5 for a description of the other functions performed by the SDU.) Bus conversion in both directions is done by hardware mapping logic and requires no intervention by the 8088 microprocessor on the SDU board. The transparent conversion acts like a bus window in that the entire Multibus memory space appears in the NuBus slot space of the SDU. When an address emitted by a NuBus master falls into this area, the converter acquires Multibus mastership. When the converter sees a transfer acknowledge from the Multibus, it causes an ACK on the NuBus. A similar window effect causes a Multibus transaction to be converted to a NuBus transaction.

NuBus 8- and 16-bit transactions are converted directly into 8- and 16-bit Multibus transactions. NuBus 32-bit transactions are turned into two separate Multibus transactions to successive addresses. Block transfers are not supported.

The implicit conversion between the two buses is represented in Figures 7-1 and 7-2.

Figure 7-1 shows that the 1-Mbyte Multibus memory space appears as one continuous megabyte in the NuBus physical address space. The transparency of the conversion is such that a NuBus processor can access data or even execute programs out of the Multibus memory.

7.2 Conversion Protocol

When a Multibus master acquires the bus and initiates a transfer, the converter (which is a slave device on the Multibus) checks to see if it is being addressed. This test to determine if the Multibus address is within a page that must be converted to a NuBus page is accomplished by using the upper 10 bits of the Multibus address to reference the Multi-to-Nu page map. A bit in each entry in this page map, called the Valid Entry Bit, determines if a conversion is to take place. If a conversion is required, the 22-bit PFN (Page Frame Number)

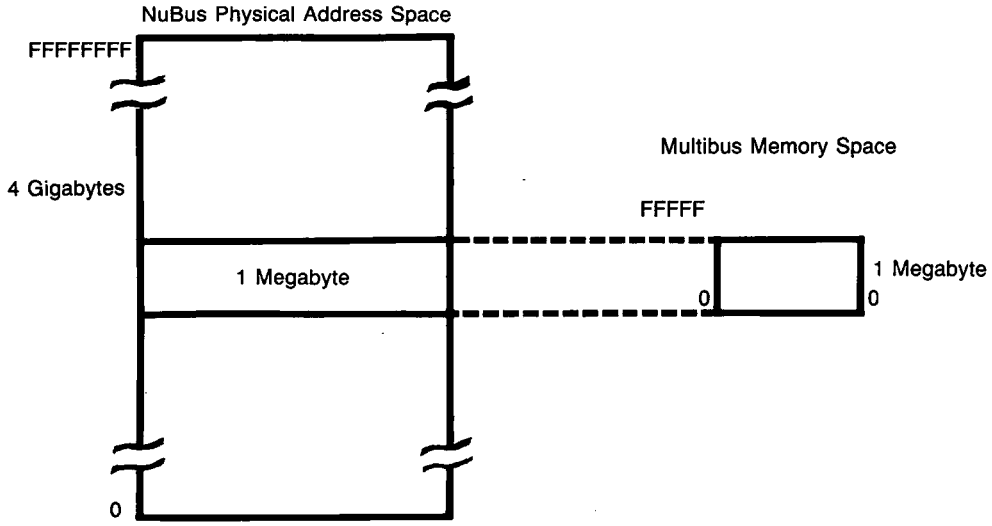


Figure 7-1. NuBus-to-Multibus Conversion.

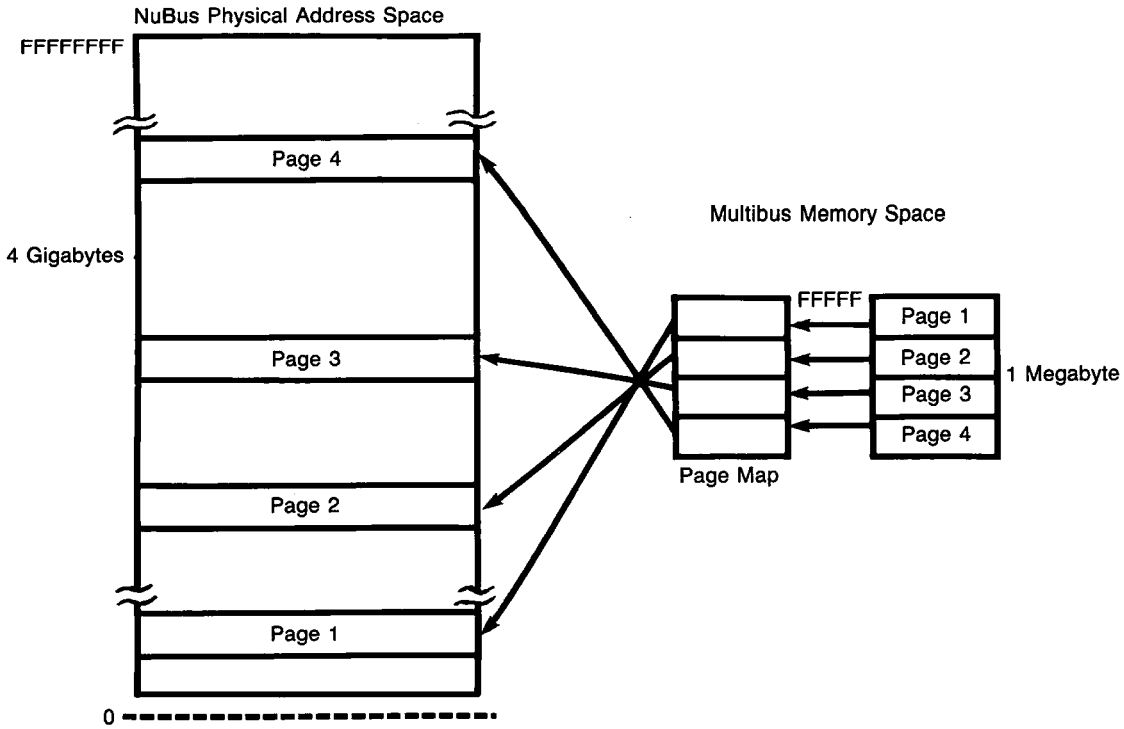


Figure 7-2. Multibus-to-NuBus Conversion.

field of the PTE (Page Table Entry) determines the page of NuBus memory referenced. The completed translation of the NuBus address is the concatenation of the 22-bit PFN field with the lower 10 bits of the Multibus address. Figures 7-2 and 7-3 show this process from two points of view.

The 64-Kbyte Multibus I/O space is mapped by the converter to a 64 K-word region in the NuBus SDU slot space. Each byte of the Multibus I/O space is mapped to a separate word of the NuBus space. Thus NuBus masters that are capable only of performing 16- or 32-bit operations are able to access individual byte locations in the Multibus I/O space. Figure 7-4 illustrates the Multibus I/O space mapping.

7.3 "Deadly Embrace" Solved

In bus conversion, a potential problem exists; when both buses request the other simultaneously, a "deadly embrace" could occur. The LMI Lambda protocol provides an elegant solution to this problem. If a NuBus master requests use of the Multibus converter at the same time that a Multibus master has requested the NuBus, the SDU returns a "Try Again Later" response, informing the NuBus master that it is unable to respond at that moment. The NuBus master then releases control of the NuBus and must rearbitrate for the NuBus at a later time. This scheme allows the Multi-to-Nu transfer to complete, freeing the conversion path. (See Chapter 8 for internal LISP processor handling of this problem.)

The mapping of Multibus interrupts is handled by the 8088 processor on the SDU board. When a Multibus interrupt is received by the 8088, the 8088 uses a table that associates Multibus interrupts with NuBus interrupt address locations to cause the appropriate interrupt by writing into the NuBus address space.

Event cycles on the NuBus are mapped into Multibus interrupts by an addressable latch on the SDU. The NuBus writes to this area of the address space to create Multibus interrupts.

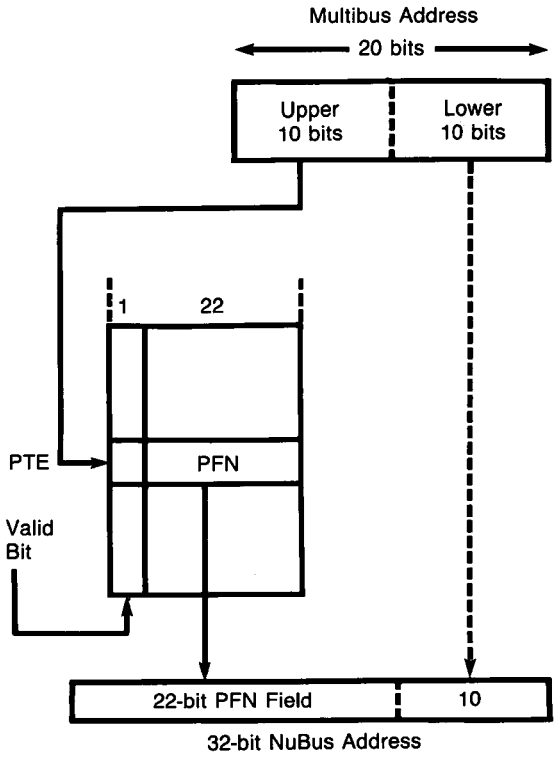


Figure 7-3. Multibus-to-Nubus Address Conversion.

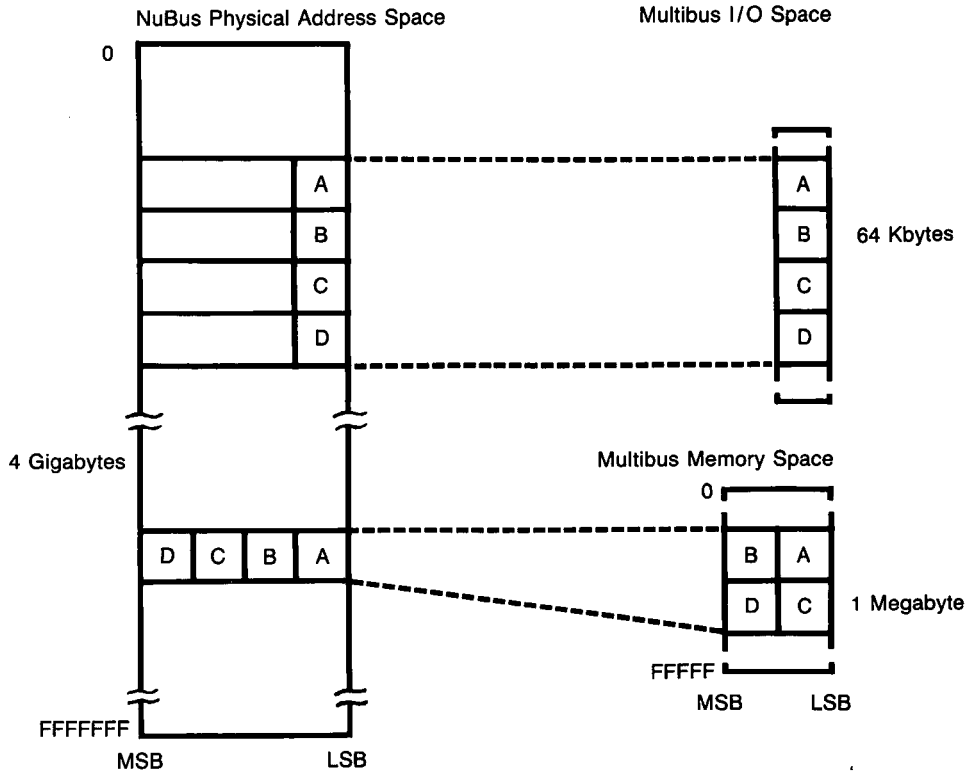


Figure 7-4. Multibus I/O Space Conversion.

Chapter 8

The LISP Processor

The LMI Lambda LISP processor provides a hardware and microcode architecture specifically designed to support the LISP Language. The processor architecture is developed from the Series 3 CADR, LMI's first generation LISP Machine. In this generation, however, several significant architectural improvements have dramatically increased performance. These features are:

- Pipelined macro-instruction dispatch hardware
- 4K x 32 single-sector, write-through cache
- 64K virtual control memory
- Hardware-assisted, 4 volatility level garbage collector
- Hardware multiplier chip
- Timing ram for microinstruction margining

Other major features of the LMI Lambda LISP processor include:

- Dynamically writeable microcode
- Paging microcode
- 32-bit data and address paths
- 25-bit virtual address space
- Stack-oriented architecture
- High-speed stack buffer and hardware stack pointers
- Cdr-coded word format identification
- Flexible dispatching and subroutining
- Packet operations
- Hardware tagging and detagging
- Cache verification factor
- Excellent byte manipulation abilities.

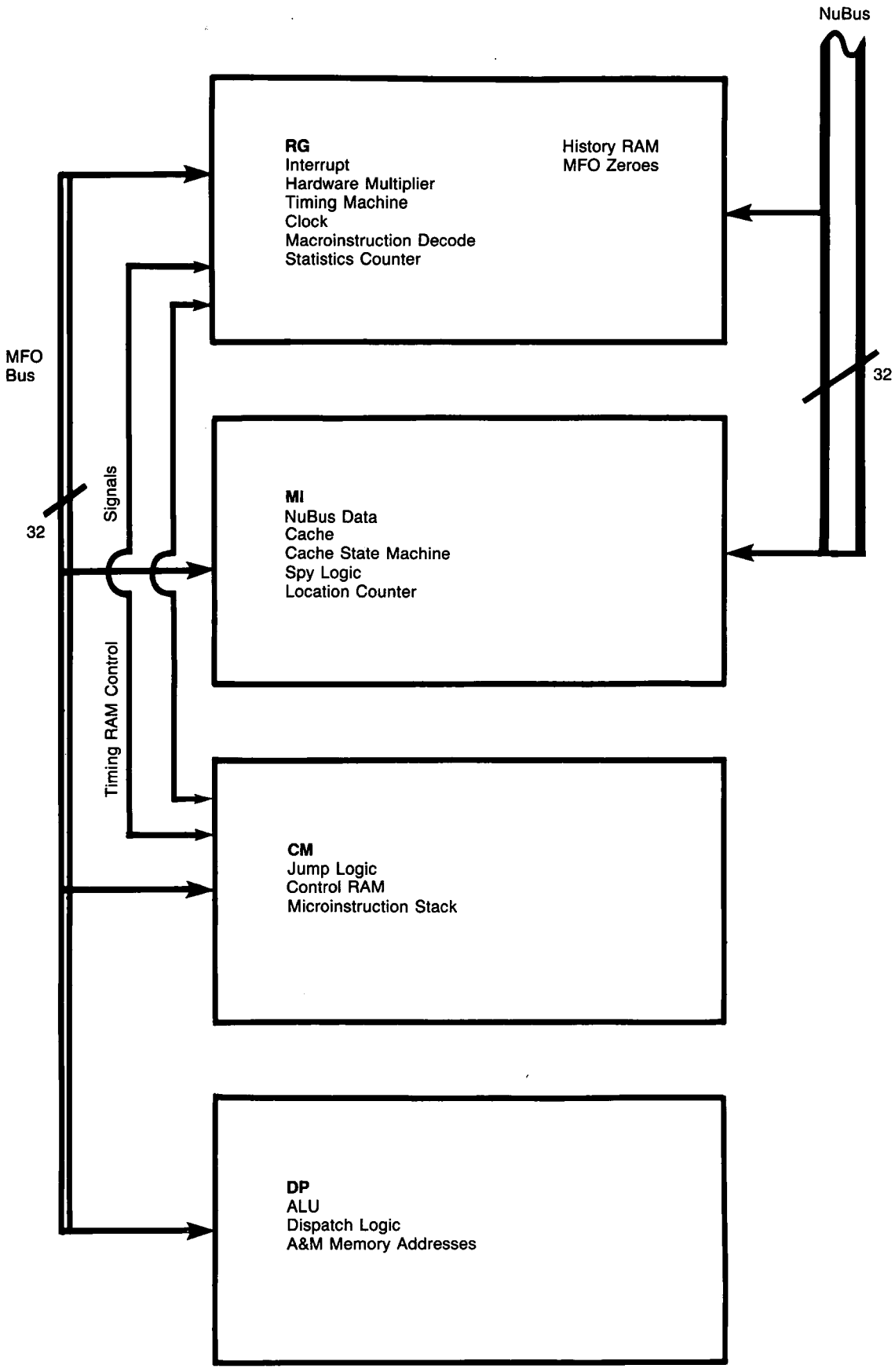


Figure 8-1. The LMI Lambda LISP Processor.

No critical parts of the design are "wired in"; thus any changes in the implementation of LISP can be accommodated easily. However, the hardware does include several features designed to speed up certain common operations in LISP microcode. These are described in later sections of this document.

8.1 General Features of the LISP Processor

The LISP processor of the LMI Lambda is a general-purpose, 32-bit microprogrammable processor designed for convenient emulation of complex order codes. It interprets the bit-efficient 16-bit order code produced by the LISP compiler. The data paths of the LISP processor are 32 bits wide. Each 64-bit-wide microcode instruction specifies two 32-bit data sources from a variety of internal scratchpad registers; the two data-manipulation addresses can also specify a destination address. The internal scratchpads include a 2K pointer-addressable RAM intended for storing the top of the emulated stack, in a manner similar to a cache. Since a large percentage of main memory references are to the stack, this materially speeds up the machine.

The LISP processor has a 16-bit microprogram counter, which behaves much like that of a traditional processor, allowing up to 64K of writeable microprogram memory. (Though only 16K of physical microcode storage is provided, the whole 64K address space can be utilized through a microcode paging feature.) Also included is a 256-location microcode subroutine return stack.

Memory is accessed through a two-level virtual paging system, which maps 25-bit virtual addresses into 32-bit physical addresses.

The processor consists of four modular circuit boards. These boards are linked by the MFO bus, a private bus separate from the NuBus, to avoid competition with other devices on the NuBus during internal operations. Separate busses are used for dispatch information, timing control signals, and debugging.

Each board centralizes functions in a particular area of the LISP environment. The DP (Data Paths) board holds the Arithmetic Logic Unit (ALU), dispatch logic, and A and M memories. The CM (Control Memory) board handles the jump logic, the microcode, and the microinstruction stack. The MI (Memory Interface) board acts as the NuBus hardware master, and holds the cache, cache state machine, location counter, NuBus data and spy logic (which is used for debugging).

The RG board contains a wider variety of functions, including interrupt logic, macroinstruction decode, the statistics counter, the multiplier, and the clock. The RG board generates timing control signals for the CM board to control the microinstruction RAM's operation. Old program counters are stored in the History RAM on the RG board, enabling quick state restoration in case of a machine halt for debugging.

8.2 Micro-Architecture

The core of the LISP processor is a conventional microprogrammable register-oriented processor. It has a 32-bit Arithmetic Logic Unit, which can do logical operations and simple arithmetic such as addition and subtraction. The A side of the ALU can be driven from a bank of 4096 registers. The first 64 locations of A memory are overlaid by the first 64 locations of M memory; therefore, a write to any of the first 64 locations of M will cause a write to A. The M side can be driven by one of 32 "functional sources." Functional sources are the paths by which special pieces of hardware are used by the machine. Examples of functional sources include the top-of-stack cache, the main memory interface registers, and the interrupt control logic.

The output of the ALU is connected in several ways to a 32-bit selector, whose output is the output bus of the machine. Among the eight possible inputs to that selector are the direct ALU outputs, the ALU shifted left or right by one bit, a bit-reversed version of the ALU, and the output of the barrel shifter.

There are two users of the output bus. In any ALU instruction the number on the output bus can be written back into the register scratchpad. There is no restriction on writing into a register that is used as a source in the same cycle. A value written to one of the lower 64 registers will be available to both sides of the ALU,

while the rest of the 1024 registers can be used only on an A input. The other users of the output bus are the functional destinations. This is the other half of the interface to the special hardware in the machine. Writing to a functional source may cause a special register to be written, for instance a mode register for the processor, or may start some complicated action, for instance a memory cycle.

8.3 Microinstructions

There are four classes of microinstructions. Each specifies two sources, A and M; the ALU and BYTE operations also specify a destination (A, or M plus functional). The A bus supplies data from the 4096-word A scratchpad memory, while the M bus supplies data from either the 4096-word M scratchpad memory (a copy of the first 64 locations of the A scratchpad) or a variety of other internal registers.

The four classes of microinstruction are:

ALU	The destination receives the result of a Boolean or arithmetic operation performed on the two sources.
BYTE	The destination receives the result of a byte extraction, byte deposit, or selective field substitution from one source to the other. The byte so manipulated can be of any non-zero width.
JUMP	A transfer of control occurs, conditional on the value of any bit accessible to the M bus, or on a variety of ALU and other internal conditions such as pending interrupts or page faults.
DISPATCH	A transfer of control occurs to a location determined by a word from the dispatch memory selected by a byte of up to seven bits extracted from the M bus.

There are several sources and destinations whose loading and use invoke special action by the microprocessor. These include the memory address and the memory data registers, whose use initiates main memory cycles.

Some of the ALU operations are conditional, depending on the low order bit in the Q register and the sign of the A source. These operations are used for steps in multiplying and dividing.

8.4. Jump Instruction

Like other standard processors, the LISP processor has a jump instruction. Jump instructions may be conditional either on results of operations performed in the ALU or on special conditions in the machine. They have the same fields as the ALU instructions to specify A and M sources, and an ALU operation. The carry and sign bits are available to the condition checking logic. The microassembler makes it convenient to jump on the usual conditions of A^6M, A^5M, and so on. Another bit of the jump instruction inverts the jump condition so that the microprogram does not have to be unduly convoluted. Other jump conditions allow testing of conditions in the machine such as page faults and interrupts.

The machine also has a 256-level microstack, and the jump instruction has bits to specify pushes and pops. The choice of where the next address will come from is made separately from the calculation of the jump condition, therefore allowing fully conditional calls and returns. Microstack overflow is not protected against.

8.4.1 Byte Rotator

One of the most useful LISP-oriented features of the machine is the byte rotator, a 32-bit barrel shifter that allows a string of any number of bits from one word to be inserted into any position in another word. LISP uses this feature primarily to extract the data type from a word and use it in dispatch instructions. It is also heavily used in examining the headers of packed data structures, for instance array headers and compiled function descriptions.

8.4.2 Dispatch Instruction

The dispatch instruction takes all of the source fields associated with the BYTE instruction. It also takes a "dispatch address", which is a 12-bit address into dispatch memory. The instruction is usually written so that some string of bits from an M memory location is extracted into the low bits of the output of the byte rotator. The result, say a 5-bit data type, is logically ORed into the dispatch address, and then that word of dispatch memory is accessed to get the microcode address to jump to. The dispatch memory also contains bits to allow the instruction to act like a CALL or RETURN or even a NO-OP.

8.4.3 Memory Interface

The memory interface is controlled by two registers, the virtual address register (VMA) and the memory data register (MD). As usual with special registers, they are accessed from a microprogram as functional sources and destinations. Each register has two functional destinations, one that just writes the register and the other, which also starts the memory cycle. To read a word from memory, one must usually write the desired address to the functional destination VMA-START-READ. A design rule of the microprogram is that the next instruction must check to see if there has been a page fault, and, if so, call the page fault handling routine. After that is taken care of, any following instruction can use the MD register as a functional source to get the value read from memory. If the value of the MD is needed before the memory device responds, the processor is automatically delayed until valid data is latched into the MD.

The LISP processor has a two-level virtual memory map to convert from the internal representation of memory as 2^{25} 32-bit words to NuBus addresses that are 2^{32} 8-bit bytes. The maps include bits to disallow read or write access to a page, and to remember the most volatile pointer stored in a page, as well as a few read/write bits that are not significant to the hardware but are provided for the convenience of the microprogram.

8.5 Microcode

At the heart of the LMI Lambda LISP processor is a high-speed control RAM holding the LISP microcode instruction set. The RAM holds 16K x 64 of physical microcode and paging is used to provide 64K x 64 of virtual microcode. This makes microcompilation practical and provides ample space for users to implement additional, customized instruction sets.

8.6 Macroinstruction Dispatch Unit

The macroinstruction dispatch unit is the core facility for macroinstruction performance improvement over the earlier system architecture of the CADR. The Macroinstruction Dispatch Hardware consists of two parts: the Macroinstruction Dispatch Register and the Macroinstruction Dispatch RAM.

The Macroinstruction Dispatch Register holds up to two 16-bit macroinstructions in a pipelined fashion. Once a macroinstruction is placed in the macroinstruction dispatch register, that instruction is immediately decoded in hardware. High-speed decoding circuitry then references the macroinstruction decode RAM.

The macroinstruction decode RAM holds up to 256 references, which point to all the macroinstruction emulation routines that are written within the system microprogram. Each one of these macroinstruction emulation routines performs the intended semantics for each macroinstruction within the system. In macroinstruction dispatch, hardware determines where the macroinstruction emulation routine resides in the beginning of its execution. The procedure does not require any additional microinstruction processing and takes place in one clock tick of the system clock.

The macroinstruction decode register utilizes a pipelining technique. One may have up to two macroinstructions pipelined through for decoding, and may eliminate system overhead required to re-setup and re-fill the pipe after the dispatching of individual macroinstructions.

The increase in system efficiency may be judged by the example of decoding and executing a system microinstruction, which moves information from the stack top to an area of physical memory. On the earlier CADR-style techology machines, in which both phases were done in microcode, this operation took 18 micro-instructions. On the LMI Lambda this operation now takes two micro-instructions—a 900% increase in efficiency for this operation.

8.7 Interrupts

The LISP processor uses a vectored interrupt system. Each device is given an address space in the interrupt slot and an address is assigned in software to each type of interrupt. The status of each interrupt is stored in a RAM, which is scanned to see if any device wishes to interrupt. When an interrupt is found, scanning stops so that no other interrupt will be seen until the current one is serviced and cleared. This prevents interrupts from being lost, since no other interrupt is noted until the pending one has been handled.

The LISP processor allows two types of interrupts, fast and slow. Interrupt scanning alternates between the 16 fast interrupts and one of the 15 groups that each contain 16 slow interrupts. Fast interrupts are detected within at most 3.2 microseconds, and slow interrupts are detected within approximately 50 microseconds.

This interrupt mechanism provides several user-definable fast interrupts and many user-definable slow interrupts, creating a very flexible interrupt environment.

8.8 Garbage Collection

Garbage collection is the process by which a LISP environment recovers memory space that has been allocated but is no longer being used. Garbage collection occurs incrementally, in parallel with computation, or as a batch operation.

Most LISP machines collect garbage indiscriminately throughout the machine memory space. On the LMI Lambda, garbage collection has been augmented by hardware to operate efficiently with less processing time. The machine supports four levels of volatility in memory space to maximize garbage collection against scavenging time. Areas most likely to need garbage collection are scavenged most frequently; more static areas are scavenged less often.

8.9 Debugging Hardware

For debugging purposes, the LISP processor can be halted without loss of information. The user can program the machine to halt in the event of a parity error or any one of a number of other conditions, such as execution of a particular instruction or execution of an instruction a set number of times. (This can be checked by the statistics counter.)

At the time of the halt, the machine state is saved. The timing microinstruction clock is stopped so that all latched information is preserved. The halt takes effect before the next instruction is executed, so the machine state is precisely as it was during execution of the instruction that initiated the halt.

Spy logic circuitry can then be used for unclocked transactions, finding information about the state of the halted machine to locate the problem. The machine can also be single-stepped, with the user generating clocks in order to track down a possible timing problem. The spy logic uses the "HBus," a private backplane connecting the CM and RG boards in the History RAM to store old program counters.

The spy logic can also be used while the machine is running as long as the spy operations used do not interfere with regular machine operations.

8.10 Improved Efficiency through Caching

Resident within the LISP processor of the LMI Lambda is a cache to increase performance through the buffering of virtual memory in a high-speed processor internal register. The cache is single-sectored, 16 words

per sector, and provides approximately an 80% hit rate. The cache takes advantage of the NuBus's block transfer facility, which updates 16 32-bit words autonomously, without processor intervention.

Second-level virtual memory mapping registers have 4K of high-speed register memory and program size is 2^{25} 32-bit words. Page hash table sizes have also been increased.

8.10.1 Cache State Machine and Cache Memory Management

The cache state machine, a specialized high-speed processor, controls the processing of data to and from the LISP processor. When it receives a request for information from the processor, it checks for a read. If it does not find one, it arbitrates with the NuBus for a reply, which it passes to the machine and stores in the cache.

Information retrieval efficiency has been increased through "packet operations", made possible because of the NuBus. The cache state machine presumes that, if you require one word, you are likely to require words nearby, so it transfers information in packets; it stores the entire packet even though the processor currently needs only one word. Similarly, it assumes that if you've referenced data recently, you are statistically likely to want it again soon, and it keeps recently referenced data packets for a period.

For speed and simplicity, the cache is single-sector and write-through, avoiding the complication of having to check "dirty bits" in the cache every write cycle and allowing cycles to be controlled by fast writes rather than slow reads.

8.10.2 Cache Verification Feature

With a multiprocessor environment incorporating a cache, there is a danger of processors interfering with each other's data. Caching on the LISP processor is done on physical address, thus removing the problems associated with flushing the cache when another processor or device writes to an area of memory. The cache monitors the bus for all transfers and automatically invalidates entries for any write to a location that happens to be cached.

8.11 The Virtual Control Memory

The LISP processor provides the user with a uniquely large virtual control memory, a virtual control space implemented in a 16K × 64 physical area within the processor. This physical area is divided into two partitions, each 8K in length. One 8K partition is wire and is non-pageable, while the other 8K partition pages within the virtual space.

The virtual control memory appears as an array within the virtual address space of the LISP Machine. New control programs can be loaded very easily by LISP functions accessing a dedicated array within the LISP address space. Additionally, by using the AREA feature of ZetaLISP-Plus, one can physically lock in portions of the virtual control memory such that it does not page all the way out to the disk, but instead always resides within the physical memory of the machine. This allows the programmer to place his microcode according to priority: critical microcode in the physical control memory (the 8K space that is wired), less critical control programs locked within physical memory (so that they won't be paged out to the disk), and finally even less critical control programs paged all the way out to the disk.

8.11.1 The Microcompiler

In addition to the large virtual control memory, LMI offers an optional microcompiler, which takes LISP source and, rather than taking it to system macrocode, translates it directly to microcode. Avoiding painful hand assembly, the microcompiler speeds throughput in the production of microcode and gives the LISP applications programmer access to a highly efficient 5 MIPS microengine for system critical code.

8.12 The Hardware Multiplier Chip

A critical area of processing within ZetaLISP-Plus is the accessing and manipulation of arrays. The most critical component of array processing is the calculation of array indices into virtual memory addresses. LMI provides within the basic LISP processor architecture a high-speed 16 x 16 multiplier chip that provides multiplication in parallel in 100-200 nanoseconds. This allows extremely high-speed array processing, which heretofore had been done primarily in microcode. Graphics throughput is increased through faster screen location computations.

Additionally, the multiplier chip assists in simple arithmetic by being used in place of the many "shift and add" operations that otherwise would be necessary to multiply two 16-bit numbers. The multiplier's operation enhances machine performance by significantly reducing the amount of time necessary for such a multiplication.

This feature improves the processor's arithmetic ability by increasing calculating speed. Overall operating speed is increased by reducing the time spent on each multiplication, and disk accessing time is shortened.

Finally, the multiplier chip is accessible by the user for high-speed signal processing or arithmetic processing requirements. This chip provides a basic multiplication utility available throughout the machine architecture.

8.13 The Timing RAM

Microinstruction execution is built around a Timing RAM, which holds a bullion vector, describing how each microinstruction is to execute. Loadable by software, the Timing RAM describes how long or how short a time each microinstruction need take, and allows systems integrators to margin the machines toward particular speeds. The Timing RAM can be loaded with a number of different configurations, allowing people to margin the execution of microinstructions.

8.14 Summary

The LISP processor architecture within the LMI Lambda was built with extensibility as a key component of the design. It maintains the philosophy of previous LISP Machines: integrated hardware-software development environment with high-speed throughput and extensive, customizable support of LISP. However, the LMI Lambda upgrades the LISP environment significantly, making it far more reliable, efficient and fast. Modular boards and special debugging hardware and software make debugging faster, without the external interfaces required on previous LISP Machines. Increased memory sizes and special-purpose processors increase processing speed, provide room for customized environments, and ensure greater reliability. Garbage collection has been redesigned for maximum efficiency; the virtual address space has been designed for easy interfacing with other standard environments; and a vectored interrupt system prevents interrupts from being lost while allowing a remarkably flexible interrupt environment. Cache management allows maximal use of high-speed memory while preventing inadvertent use of obsolete data. Most importantly, the integration of the LISP processor into the total LMI Lambda environment allows the LISP programmer a high-speed, reliable LISP interface to a large number of additional computing environments.

Chapter 9

The 68010 Processor Board

(This information will be current as of third quarter 1984.)

The UNIX-based CPU board design in the LMI Lambda incorporates a Motorola MC68010, a high-speed cache memory, and a virtual memory translation unit.

In the LMI Lambda, the 68010 generates 24-bit virtual addresses, which allow access to 16 Mbytes per virtual address space.

Some of the features of this unit are:

- 32-bit arithmetic
- 17 registers
- 24-bit virtual addresses for each process
- 10 MHz operation
- Byte addressability
- 4 Kbyte data cache
- No wait states on cache "hits"
- Demand-paging virtual memory implementation
- Translation engine to translate virtual addresses to physical addresses
- 512-entry translation "look-aside" buffer.

9.1 Virtual Address Space

In a typical multiprogramming environment, several tasks are active at the same time. The 68010 CPU provides separate virtual address spaces for each of these tasks. Because even a single virtual address space may be too large to be contained in the available main memory, the active part of the virtual address space must be mapped to the available physical address space. To effect mapping, the physical and virtual memory are divided into 1-Kbyte units called pages. A page of virtual memory is mapped either to a page in physical memory or to a page on mass storage. The virtual memory hardware and software assign virtual addresses to specific physical memory locations in real time while the process is executing, in some cases having to

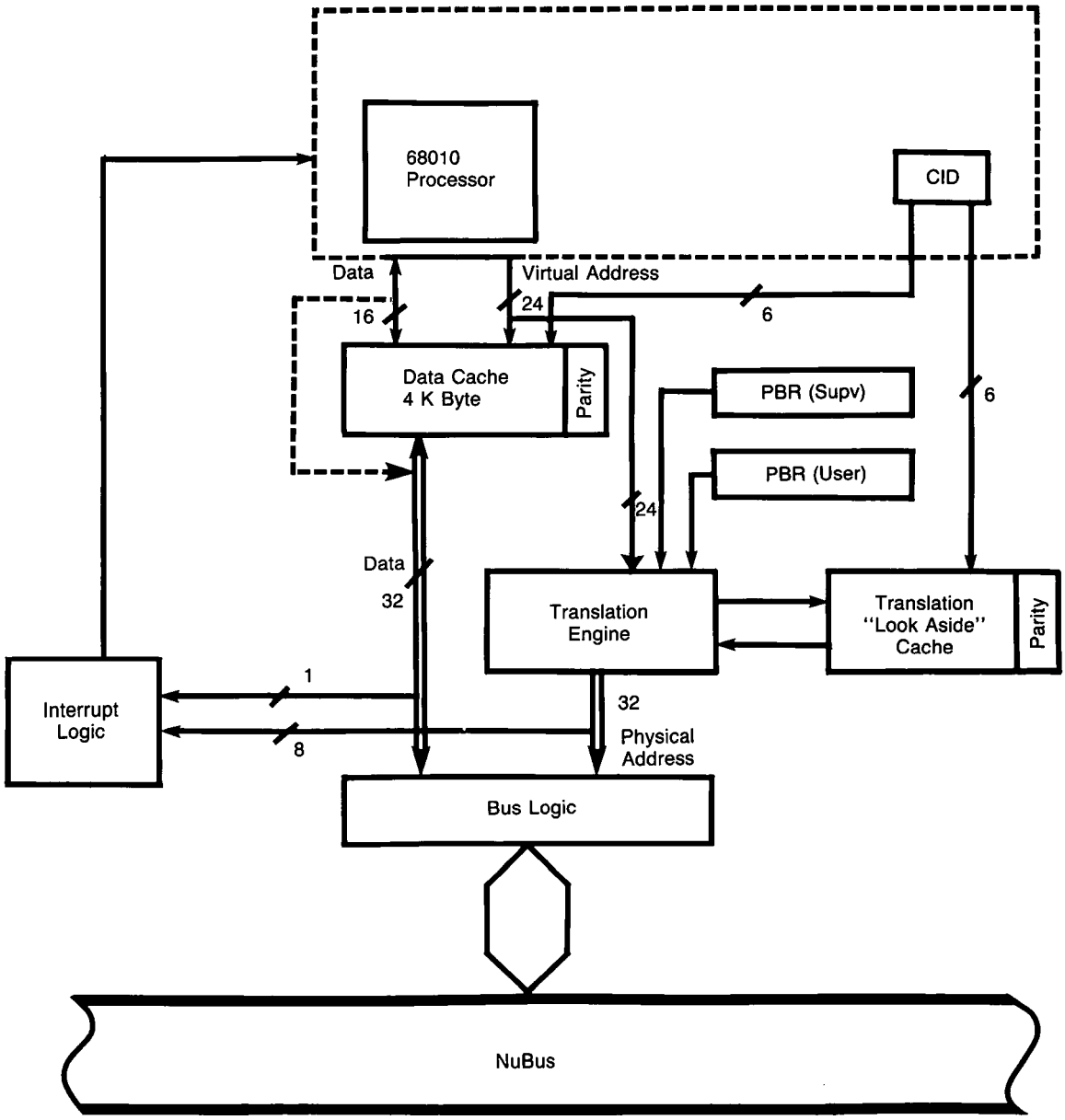


Figure 9-1. The 68010 Processor Board.

bring the required location into memory from the disk. Usually the physical memory assigned to different processes will be non-overlapping. However, the paging mechanisms do allow one physical location to be mapped into the virtual address spaces of two or more processes to implement data or code sharing.

9.2 Processor Control

The data cache is a high-speed memory that maintains a copy of frequently-accessed words of main memory providing faster average access to instructions and data. Because the processor looks for the data in this cache first, the cache architecture offers faster system speed for the cost of a small amount of fast memory and assorted logic.

If the data or instructions are located in the data cache, 8 or 16 bits of data are sent back to the 68010. This occurrence is termed a "hit," and the hit rate for this particular cache is approximately 85%.

9.3 Translation Engine

If the data is not found in the cache, the virtual address must be translated into the physical address in order to be used by the bus logic to fetch the data from memory. This translation is accomplished by the translation engine, a microcoded sequencer that accesses a page-table structure in main memory that defines the virtual-to-physical address mapping. Such a structure exists for each virtual address space. The Process Base Register (PBR) points to the base of the page table structure for the current virtual address space.

A page table entry (PTE) is a 32-bit word that represents the physical mapping for one virtual page. Although this sophisticated memory-mapping scheme allows unlimited numbers of virtual address spaces, there is a theoretical overhead of two memory references for each address mapping. To circumvent this overhead, another caching mechanism, a Translation "Look-Aside" Buffer (TLAB), holds 512 second-level PTEs and 64 first-level PTEs. (The first- and second-level PTEs are described in a subsequent section.) If a hit occurs on a second-level PTE, no memory references are necessary for translation. If a hit occurs on a first-level PTE, one memory reference is required, and in the rare case where a hit does not occur, two memory references are required to translate the address. Detailed explanations of the functions briefly described above are given in the following sections.

9.4 Interrupt Mechanism

The 68010 microprocessor supports seven levels of interrupt priority, labeled one through seven, with seven having the highest priority. The 68010 unit provides 32 unique vectored interrupts for each of these levels, totaling 224 possible cases. As described in our overview of the NuBus, the bus protocol does not support traditional interrupt signals. Traditional interrupt approaches fail for the multiprocessor environment because they are oriented to interrupting a single processor. In a multiprocessor environment, interrupts or "events" must be dynamically assignable to any of many processors.

On the NuBus, events are posted by ordinary bus write cycles to special areas of the address space to which the processor units respond. The 68010 board has a 224-word area that translates all writes addressed to that region into 68010 interrupts. These writes are latched and presented as interrupts to the 68010 in priority order.

Because interrupts, like memory and I/O, are memory-mapped in the single NuBus address space, computing units can readily interrupt each other or can even interrupt themselves from high-level language programs. The benefits of memory-mapped I/O have been extended to interrupts.

The priority level of an event is determined by its address within the 224-word area. For example, the top 32 words of the area correspond to level-7 interrupts; the next 32 words, to level-6 interrupts; and so on.

When the 68010 requests the interrupt vector, it is given the unique address of the highest-priority event. This action immediately clears that event.

9.5 Cache Operation

The 1024 x 32 (4K-Byte), write-through data cache provides the CPU with high-speed access to frequently-used words of main memory. Both instructions and data are stored in this unit and are treated equivalently. This set-associative cache contains 1024 one-word entries and carries byte parity. If desired, the cache can be disabled for diagnostics or to allow reduced performance providing graceful degradation in the event of cache failure. The write-through feature insures cache integrity because this cache is always a copy of memory.

The speed of this cache (45 nanosecond RAM) allows the 10MHz 68010 to run with no wait states.

The design of the 68010 data cache is that common to many superminicomputers. The 1024-entry cache stores data and instructions on a word-by-word basis. The upper 12 bits of the virtual memory address, of which the data are a copy, are stored as a tag with each data word.

The cache functions by performing two basic operations. The first operation is a search of the cache to determine if the desired data are present. The search is performed continuously for all 68010 memory reference operations as long as the cache is enabled. Each word of virtual memory has a place in the cache where it will be stored if it is in the cache. The cache must examine only the location in which the word would be found if it were present.

The entry in the cache where a 32-bit word is stored, if it is stored at all, is determined by the lower 10 bits of the address. That is, data from location 2003 (Hex) would be stored in the third location of the cache as would data from location 5003 (Hex). Only one of these pieces of data may be cached at any one time. The lower 10 bits of the address emitted by the 68010 are used to address the 1024-entry cache in implementing that operation of the CPU. The address tag, or upper 12 bits of the virtual address space of stored information, is compared with the upper 12 bits of the emitted address. A hit occurs when the 12-bit address tag field stored at that entry matches the emitted address being sought by that particular 68010 cycle.

A miss occurs when the upper 12 bits of the emitted address of the 68010 do not match the 12-bit address tag field of the stored data. The translation engine is then invoked, causing the bus logic to read the complete 32-bit word over the bus. The desired data item is placed in the cache entry at which the miss occurred, overwriting whatever data and address tag were there.

As mentioned above, the cache is write-through. Therefore, on write operations, when either a hit or a miss is made, the translation engine and bus logic are always invoked, causing the written data to go into main memory. Thus main memory is always a true copy of the state of the virtual address space, and the cache is a copy of that data. The cache is checked for a hit even on a write, because if a hit occurs that cache cell must be updated to the new value.

Cache data integrity is protected by byte parity across the data and tag fields. Any entry showing a parity error does not cause a hit and is treated as a normal miss except that the parity error flag in the error status register is set.

Enables are registered on the cache control register as follows:

0	Translation enable
1	Cache parity enable
2	TLB1 low enable
3	TLB1 high enable
4	TLB2 low enable
5	TLB2 high enable
6	Cache low enable
7	Cache high enable

Cache "hit" status is registered as follows:

0	Cache hit
1	TLB1 hit
2	TLB2 hit
3	Unused

For diagnostic purposes, the cache may be partially enabled or disabled via bits in the cache control. Partial enabling allows a diagnostic to test the cache without running out of the cache itself.

9.6 Selective Caching of Pages

An aspect of cache control that is not directly implemented by the cache logic is the ability not to cache selected pages of virtual memory. This capability allows support of writeable, shared data areas, for instance, semaphores that must be accessed by multiple processes. Another application of selected caching is I/O registers. When an I/O register is read multiple times, the desired data is not multiple copies of the first value but instead whatever state is current.

Selective caching of pages is not implemented directly by logic in the caching system; it is a feature of the translation system.

9.7 Advantages of 68010 Cache Architecture

As described previously, the data cache on the 68010 is directly connected to the processor and therefore caches data related to virtual address. An alternative organization might be to put the translation unit between the processor and the cache, in which case the cache would store copies of data tagged with its physical memory address.

The advantages of the approach taken by the LMI Lambda are obvious. Having the high-speed cache as close as possible in the architecture to the 68010 eliminates wait states for 10MHz 68010s. In future, faster 68010s will be accommodated by increasing the speed of the cache memory path. The problem to be overcome in systems such as this one is the need for "flushing" the cache whenever the context is switched. Because cached words are tagged by their location in the virtual address space, whenever virtual address spaces change, the data cached are no longer valid; they are copies from the old virtual address space, not the new, and must be flushed.

A six-bit register has been added to the 68010 to increase the speed at which the cache may be flushed. The six extra bits of tag are called the cache identification, or CID, field. Each active virtual address space has a unique six-bit CID code. When contexts are switched, the six-bit CID register associated with the 68010 is also changed. Therefore, cached data from previous virtual address spaces are instantly and automatically invalidated (flushed). This design operates so that, in the worst case, the cache must be flushed only once every 64 context switches. Only 64 processes may have data cached at the same time, but an unlimited number of processes may be active at any one time.

9.8 Address Translation

The 68010 CPU card provides hardware to translate 24-bit virtual addresses to 32-bit physical addresses in 1024-byte page increments. Supervisor mode and user mode have separate address spaces; however, software sets up the page tables so that they overlap.

Each page

- — may be marked as resident/nonresident by the "valid entry" bit;
- — has read and write access protection for both supervisor and user modes;

— has a bit to disable data caching on the page;

— and has both a page-accessed and a page-modified bit.

Address translation of the virtual address is determined by a structure in memory, which is pointed to by a process base register on the 68010 card. Basically, the scheme is a two-level page map where the level-one page map defines the physical addresses of the virtual address space. Only the first-level page table is required to be resident in main memory. Second-level page tables may be either resident or paged out to the disk.

Some abbreviations used in this description are listed below:

CID	Cache Identification Number
PADDR	Physical Address
PBR	Process Base Register
PTE	Page Table Entry
PTE1 and PTE2	Level-one and level-two Page Table Entries
PFN	Page Frame Number
VADDR	Virtual Address (from the 68010).

The level-one map is 64 words long and pointed to by the 24-bit PADDR. The hardware contains two PBRs—one selected for supervisor mode, one for user mode—and thus it directly supports separate address space for these modes. Bits 23 through 18 of VADDR are appended to the PBR to form a physical address for a PTE in the level-one page map. This PTE is read from memory.

PTEs have a field, the PFN, that contains the high-order 22 bits of the physical address of a page. Pages always begin on multiples of 1024 bytes; therefore, the PFN with 10 bits of 0s concatenated is the address of the page.

Each memory reference is checked for proper access privileges and for the validity of the PTE. The following shows the definition of the access bits. The level-one page table must be present in memory.

Access bits:

BIT 31	Reserved
BIT 30	Reserved
BIT 29	Privilege bit 1 (PV1) access
BIT 28	Privilege bit 0 (PV0) access
BIT 27	Update cache
BIT 26	Valid entry (page resident/nonresident)
BIT 25	Accessed
BIT 24	Modified
BIT 9	Reserved
BIT 8	Reserved

The access privileges are further illustrated below:

PV1, PV0 ACCESS PRIVILEGE

0.0	Supervisor read only
0.1	Supervisor read/write
1.0	Supervisor read/write, user read only
1.1	Supervisor read/write. user read/write

A memory bit sets the modified bit in the level-two PTE, which in turn sets the modified bit in the level-one PTE if it has not already been set.

The PFN of the PTE from the level-one page table appended to bits 17 through 10 of VADDR forms the physical address of a PTE in the level-two map. This PTE is read from memory. The PFN of this level-two PTE, appended to bits 9 through 0 of VADDR, represents the final physical address.

If address translations were implemented by the 68010 hardware only as previously described, each processor reference would require two memory accesses to do translation. (Of course, a hit to the data cache, as previously described, would eliminate the need to do translation at all.) To speed up these translations, the hardware additionally caches level-one and level-two PTEs. All caches are updated as necessary by the translation engine.

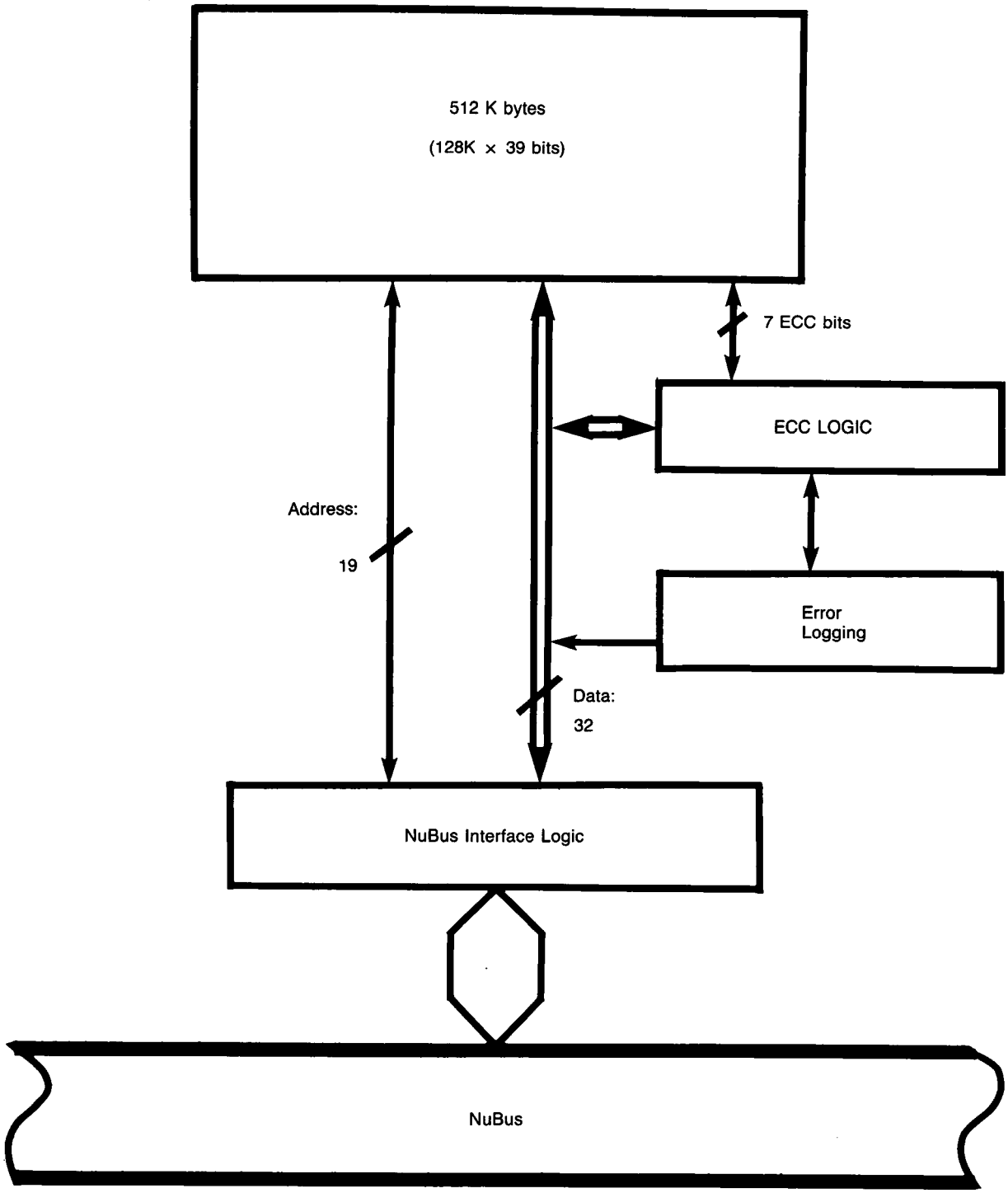


Figure 10-1. Memory Block Diagram.

Chapter 10

The Memory Board

The LMI Lambda memory board is a self-contained memory controller and 1/2 Mbyte memory array that includes support of error correction, block transfers, and error logging. The memory board that interfaces to the NuBus is a slave device, as explained in Chapter 7.

The memory board uses a 32/39 code for error correction and detection; seven ECC (Error Correction Code) bits are appended to each 32-bit data word. This code enables correction of all single-bit errors and allows detection of all double-bit errors within the 39-bit word (32 data bits plus seven ECC bits). Additionally, some multiple-bit errors are detected. A multiple-bit error is one in which three or more bits in a particular word are in error.

10.1 Normal Operations

The following paragraphs describe normal operations for reads and writes on the memory board. The memory board supports all data transfers over the NuBus. A functional block diagram of the memory is shown in Figure 10-1.

10.1.1 Read Operations

A read cycle always results in a 32-bit bus transfer while a write may transfer 8, 16, or 32 bits. (Because of the way byte and half-word reads are specified, doing a full 32-bit read of the word containing the addressed elements produces the proper result.) The memory board checks the data for errors on all read cycles and then performs one of three operations:

— If no errors are detected, data are placed on the bus, and a "bus transfer complete" (ACK) response is given.

— If a single-bit error is detected, it is corrected and the corrected data are placed on the bus with an ACK response. This correction is transparent to the bus master. The corrected data are not written back to memory.

— If a double- or multiple-bit error is detected, an error condition (NAK) response is given and the data transferred are undefined.

10.1.2 Write Operations

Write cycles may result in 8-, 16-, or 32-bit data transfers. All write operations are carried out "off-line." That is, the data are latched and the "bus transfer completed" (ACK) signal is given, freeing the bus while the memory completes the write operation.

A 32-bit transfer is the simplest write operation. The latched data are presented to memory, a corresponding seven-bit ECC pattern is generated, and both are written to the addressed location. The check bit is generated as either an XOR or an XNOR.

On 8- and 16-bit transfers, a read of the addressed location is performed first. One of three things then occurs:

— If no errors are detected, the new byte(s) substitute(s) for the old data; a new ECC pattern is generated; and the new word and ECC are written in the addressed location.

— If a single-bit error is detected, it is corrected before the new byte(s) is/are substituted. This new word and its ECC pattern are then written in memory.

— If a multiple-bit error occurs, the write portion of the partial write is not performed. Thus, the data with the error are left in memory so that a subsequent read will detect an error. No indication of the error is immediately returned.

10.1.3 Block Read Operation

Block read cycles result in 2-, 4-, 8-, or 16-word block transfers (32 bits/word). Each word is read and the data are checked for errors. Then one of three operations is performed:

— If no errors are detected, the data are placed on the bus. If that transfer is the last one, a "bus transfer complete" (ACK) response is given. If that transfer is not the last one, an intermediate ACK is given and the transfer continues.

— If a single-bit error is detected, it is corrected, and the corrected data are placed on the bus. If that transfer is the last one, an ACK response is given. If that transfer is not the last one, an intermediate ACK is given and the transfer continues.

— If a multiple-bit error is detected, undefined data are transferred, an error condition response (NAK) is given, and the block transfer terminates.

10.1.4 Block Write Operation

Block write cycles result in 2-, 4-, 8-, or 16-word block transfers (32 bits/word). Each word of data is latched and the data are acknowledged. An intermediate ACK response is given for all words except the last words transferred. When the last word is transferred, a "bus transfer complete" (ACK) response is given and the bus transfer is complete. Each word of data is latched and presented to memory; a corresponding seven-bit ECC pattern is generated; and both are written to the addressed location. Then the address is incremented to the next word. This process occurs for each word of data to be written until all words have been transferred. Because only complete words are written, no errors can occur on a block write operation.

10.2 Error Reporting

The memory board records information regarding ECC errors. The two registers are the Memory Control Register, which contains three bits indicating types of errors, and the Error Address Register, which contains the complete address of the first error location.

— Memory Control Register

0-4	Diagnostic control modes
5	Multi/single bit error
6	Double bit error

7	Single bit error

— Error Address Register

0-22	Board relative error address
23-31	Error syndrome bits

10.3 Two-Megabyte Boards

LMI intends to start deliveries of a 512K x 32-bit memory board by third quarter 1984. The board, under checkout at this writing, is designed with high-density "surface mount" technology and provides parity error detection.

The board design has been implemented for both the 64K RAM and 256K RAM. The initial 512K × 32 boards will have 64K RAMs; when 256K RAMs become plentiful, these boards can easily be upgraded to 2048K × 32-bit . We expect this to occur in late 1984 or early 1985. This will allow a maximal physical memory configuration on the LMI Lambda of 5 Mwords utilizing the 64K RAM technology or 20 Mwords utilizing the 256K RAM technology.

Chapter 11

User Interface: Video, Keyboard, Mouse

The video display subsystem is a single-board, text and graphics display system that incorporates a bit-map memory plane and video control circuitry to drive a high-resolution non-interlaced monitor. Two video buffers on the board provide high bandwidth from the NuBus and high bandwidth to the display monitor. This board can be a master on the NuBus in order to generate interrupts to the processors. Features of the video display subsystem are:

- One megabit of dedicated display memory in each of two video buffers
- 1/60-second display update per screen
- 64K RAMs
- Programmable number of words per line and lines per frame
- Arithmetic Logic Unit (ALU) supporting on-board logical functions such as XOR
- Normal/reverse video
- Horizontal, vertical, and composite synchronous outputs
- TTL and ECL video outputs available
- Programmable interrupt generations using vertical blank
- 70 MHz maximum pixel display

11.1 Architecture

The video display subsystem uses a unique architecture with two dual-ported memory arrays. This approach is motivated by the need to provide a high bandwidth from the video memory to refresh the high-resolution non-interlaced display, and, at the same time, to provide high-bandwidth ability to write into the bit-map memory to change the display. The two dual-ported arrays, called A and B, are each a 16 × 64K memory array. As seen on the block diagram, the lower (or B) memory is constantly addressed by the refresh address counters and is used to continually repaint the screen by providing a 70-Mbyte bit stream. This design leaves the A bit-map free to accept new data being written into it over the bus from the processor controlling the graphics.

As seen in the block diagram (Figure 11-1), not only may data be written from the bus directly into memory A, but also a read-modify-write cycle may be performed using a single ALU on the video board. This architecture allows information written to the board to be ORed, ANDed, or EXCLUSIVE ORed with values already

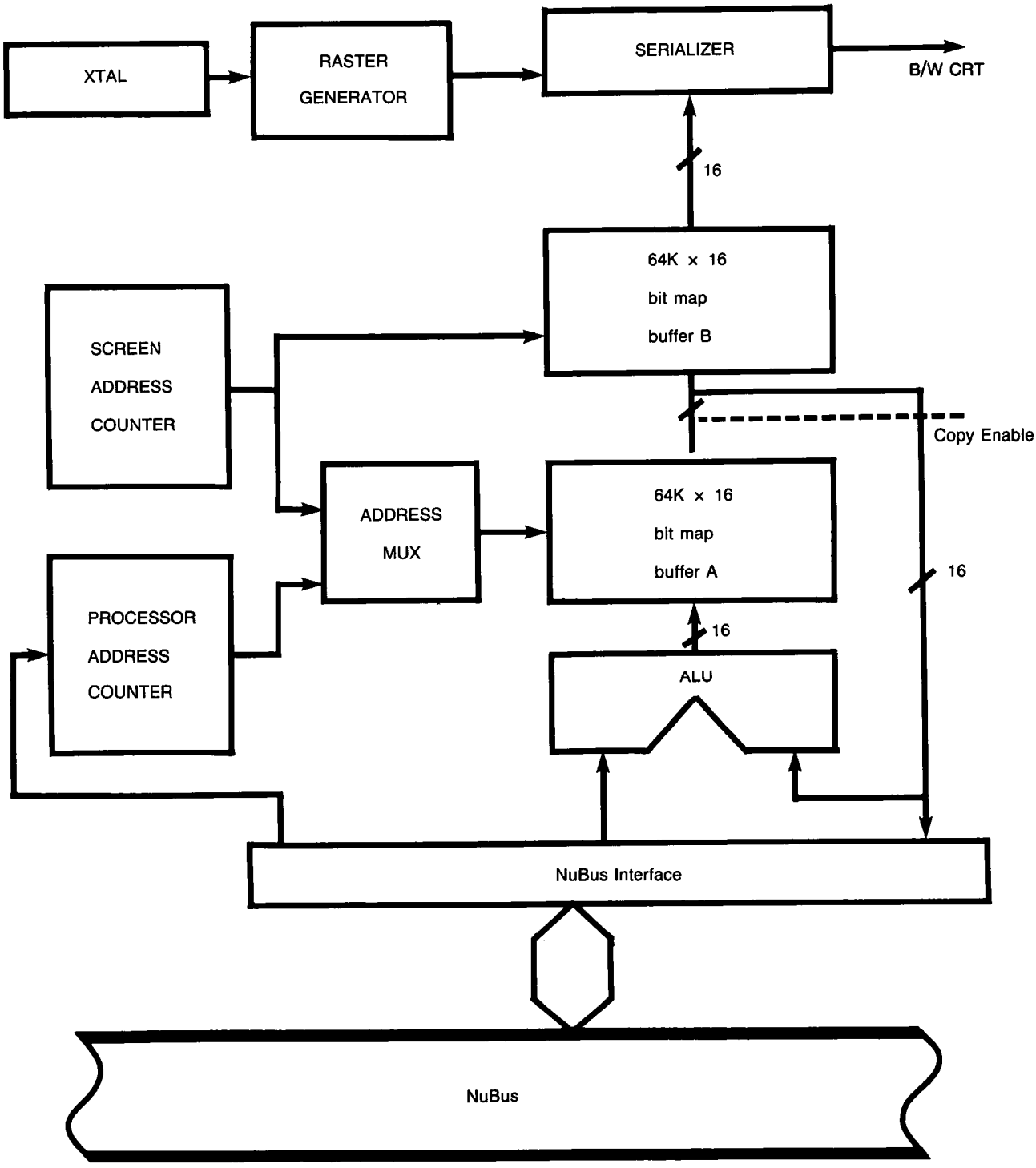


Figure 11-1. Block diagram — Video Display Subsystem.

in the bit map. This read-modify-write and on-board ALU allow many common operations to be performed within the video card itself, providing increased performance.

In order for the updated information actually to appear on the screen, bits must be copied from the A memory to the B memory. In fact, at any time when the A memory is not being accessed by the NuBus, it is being copied into the B memory, providing a continual update. However, the NuBus has priority over the updating operation, and therefore provides the maximum amount of bandwidth for getting new information into the display.

11.2 Customizable Refresh/Update

This architecture provides another feature that is very attractive in many applications. The copying operation from A to B may be inhibited by turning off a copy bit in a command register. In this state, new information that is written into the A memory is not displayed on the screen. Instead, the screen simply keeps refreshing the old information from the B memory. When the update is complete, the copy bit may be turned on, causing the new information to appear on the screen in one frame time (1/60 of a second). This technique allows the screen to show the old image until the new image is completely ready to be displayed. This feature makes the LMI Lambda suitable for use in animation and in other applications where it would be undesirable for the user to watch the screen being updated.

11.3 Scan Line Table

Another advanced feature of the architecture of the video display subsystem is the scan line table. The scan line table is a RAM that contains the starting addresses of each scan line in the B memory. That is, on a line-by-line basis, the place in the B memory where the data for that scan line starts may be specified in this RAM. This mechanism can be used for scrolling and for making rapid changes in small portions of the screen; it is also a feature that enhances the flexibility of this subsystem for use with a variety of display screens with different aspect ratios.

11.4 Serial Port for Additional Bit-Map Display

Although not directly associated with the video display function, an RS232 serial port is used on this board to complete the man-machine interface. The inclusion of this port on the board means that it has all the controller hardware to add an additional bit-map display to the system.

11.5 Ease of Servicing

A 128-byte FIFO is provided to simplify the software requirements to service this port.

The video display subsystem includes the following features for testability:

- Memory B can be read directly via the NuBus.
- Sync program can be single-stepped by software.
- Video path can be single-stepped by software; video out can be read via the NuBus.

11.6 Keyboard and Mouse

The AI keyboard furnished with the LMI Lambda contains a 100-key superset of the ASCII keyboard. Standard character sets sensed by the keyboard include alphanumeric and Greek, as well as a third programmable set of symbols; each key is fully reprogrammable, so that any key may be bound to any character or function. Unlimited rollover allows the keyboard to sense any combination of keys in any order. Four control keys allow many further functions to be bound to the keyboard, and a number of standard functions (for instance, HELP) are also given special keys.

The keyboard works in full cooperation with the three-button optical mouse and pad, standard equipment on the LMI Lambda. Because it has no moving parts, the optical mouse exceeds previous mouse systems in reliability and sensitivity, allowing easy operation. Many functions in the LISP environment can be performed with the mouse alone (pointing at windows, menus, or buffers, scrolling text, marking regions, moving objects, and so forth). Integration of the mouse into font editing and graphics environments speeds design. Like all parts of the LMI Lambda environment, the mouse interface is fully customizable.

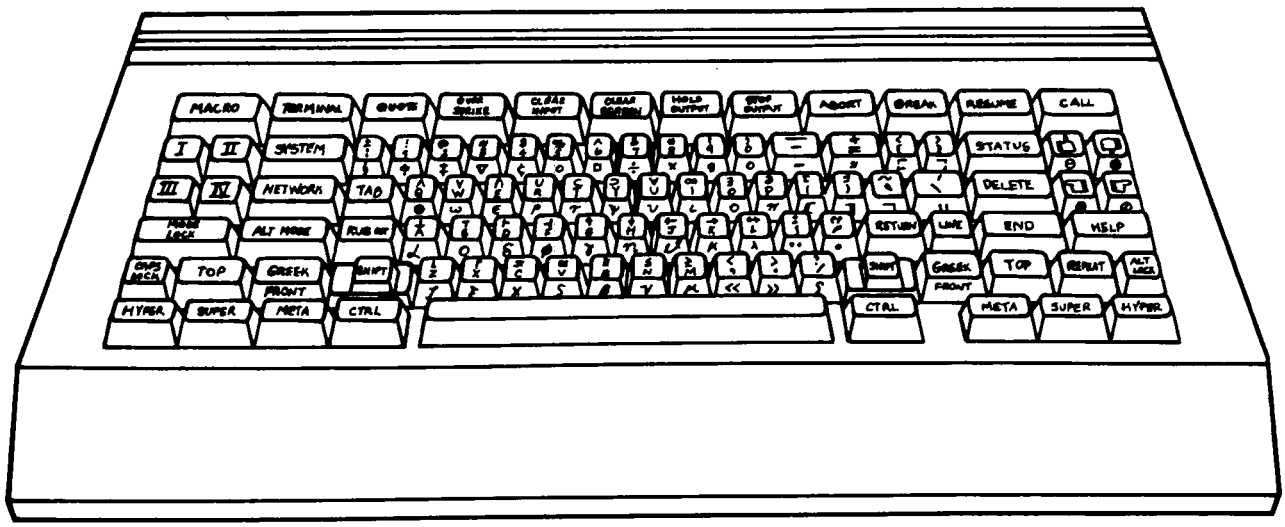


Figure 11-2. The LMI Lambda AI Keyboard.

Chapter 12

The UNIX Operating Environment

While the LISP processor provides a rich development and execution environment for LISP, the UNIX operating environment offers an industry standard and allows access to a wide range of commercially available software.

UNIX is portable over a wide range of processors, from advanced microcomputers to mainframes. Once applications have been initially designed, they can usually be implemented easily on a wide range of computers.

Advantages of the UNIX operating environment include:

- Portability from LMI Lambda to other computing environments
- Versatility
- C language base
- Hierarchical file system
- Selection of interactive debuggers
- Over 200 utilities for string and file manipulation, simplifying program design and debugging
- Screen editors: programmer-oriented, full-screen interactive editing
- Utilities for text formatting and typesetting
- Spelling checker and dictionary with customization features
- Utilities for checking diction and readability
- File and string manipulation: sorting by definable parameters and by string content; relational database systems
- Electronic mail and bulletin boards
- Maintenance and accounting
- Communications and networking

— Shell command/programming language. As a command language, it reads user input and translates it as requests to execute programs. As a programming language, it includes:

  * Control-flow primitives
  * Parameter passing
  * Variables and string substitution
  * Logical constructs

— Languages, including C, Fortran 77

The System V implementation of UNIX, to be distributed by LMI beginning in third quarter 1984, is scheduled to include the following additional features, designed to provide significant gain in overall system performance, in improved system response time and in increased user capacity per system:

— 1 Kbyte block file system

— Multiple physical I/O buffers

— Larger number of buffers, increasing the potential for cache "hits" and decreasing disk I/O traffic

— More efficient file table position entry through free list

— Faster *fork* and *open* system calls

— Hash table for sleep/wakeup mechanism

— More robust file system through ordered writes

— Dual *fsck* capacity, speeding up file system checking by a possible 40% or more

— "Fast" option to *fsck*, providing quicker recovery from crashes

— Improved file-system access, speeding up throughput by as much as 200% over randomly-organized file systems

— New command to kill processes associated with a specific file system, terminal line, or file

— New *sar* performance measurement counters

— Interprocess communication routines, including messages, shared memory, semaphores, and process locking

— New interface programs, *init* and *getty*

— New general disk driver, supporting four different standard disks, and up to four disks at once

— Changes to the C programming environment, including:

  * Improvements in efficiency of the Software Generation System
  * New Machine Language Processor

    * Enhancement in performance of many library routines
    * User-level profiling available for C and Math libraries
    * Math library enhancements, making it easier for users to detect and recover from errors
    * Expanded Fortran programming support tools, *asa* and *fsplit*
    * New library functions
    * Improved parser generator (*yacc*), symbolic debugger (*sdb*), and profiling command (*prof*)

— New common synchronous interface, replacing individual protocol drivers

— Ability to interface to user-written terminal-handling routines

— Improvements to Virtual Protocol Machine, including time stamping on trace for better debugging, extension of power-fail/restart to all protocols using the machine, and a new command for performing loop-back tests on a specific synchronous line

— Enhancements to Uucp, including optional mail suppression, improved dialing algorithm, forced routing, and job IDs

— New commands to perform faster tape backups and incremental saves

— The *vi/ex* display-oriented, interactive editor, a tested and effective tool for program development and document preparation[1]

— Optional MINCE, an EMACS-like editor, with SCRIBBLE, a SCRIBE-like formatter[2]

— New accounting options

— A Line Printer Spooling System (LP), able to queue and calculate print requests, to allow or prevent queuing to specific devices, to change printer configuration, to find its own status, and to process explicit START PRINT and STOP PRINT requests

System V will support PASCAL beginning in third quarter 1984.

1. The *vi/ex* editor is based on software developed by the University of California, Berkeley, California, Computer Science Division, Department of Electrical Engineering and Computer Science, and such software is owned and licensed by the Regents of the University of California.
2. MINCE and SCRIBBLE are licensed and distributed by Mark of the Unicorn Products, Inc. Enhancements to the system copyright 1984 LISP Machine Inc.

Chapter 13

ZetaLISP-Plus Operating System: Overview

Most computers are built first, with software then tailored to match the machine. The architecture of the LMI Lambda's LISP processor has been implemented around the LISP language, providing a fast and flexible system designed to meet the needs of efficient LISP operation in a multiprocessor environment.

Basic LISP functionality is provided largely through system microcode, providing flexibility and expandability. Certain efficiency features are directly implemented in hardware.

Built on top of this architecture is an extensive LISP implementation, providing thousands of rich and useful features in over 500,000 lines of code. These features extend the power of the basic LISP language in ways not possible on conventional architectures. Applications programmers can thus concentrate on the task in hand, choosing from an extensive set of appropriate system functions and constructs without having to re-implement system utilities or restrict their own software design.

13.1 Software Development Tools

Additionally, the LMI Lambda provides an extensive array of software development tools. Beyond the basic LISP interpreter, the system provides

- A powerful screen-oriented text editor, ZMACS
- Compilers that generate either micro- or macrocode
- FLAVORS
- A powerful display and I/O controller, the Window System
- Interactive debugging
- Ethernet networking
- Extensive file system software
- A shared communications interface with the UNIX system
- An optional microcompiler

13.2 System-Wide User Documentation

HELP is provided not only through hardcopy documentation of the system but interactively by the system itself. The editor subsystem, ZMACS, contains extensive documentation on all its commands through the

HELP key, which accesses the self-documentation system. By typing HELP and a letter, you can access documentation on commands, variables, and functions.

HELP also will undo the last extended command you performed, give you a list of your most recent 60 keystrokes, and do other valuable functions.

Because most program editing is done in the ZMACS subsystem, HELP provides a virtually complete guide to the numerous user-support functions of the system. A system-wide integrated HELP is planned for future release.

Further system-wide help is available through extended commands such as Meta-X Apropos and "Meta-Point", which are documented in LMI's INTRODUCTION TO ZetaLISP-PLUS. Because all of ZetaLISP-Plus is written in accessible LISP code, experienced users can easily read the code itself for further documentation of its use.

13.3 LISP-UNIX Interface

Since the LMI Lambda supports a multiprocessor bus architecture (i.e., the NuBus), LISP and UNIX systems can operate simultaneously within a single mainframe. A shared memory interface supports process-to-process communications between LISP and UNIX, linking both operating environments. Applications can utilize UNIX as a multi-user front-end, packaging transactions for the LISP processor, which completes the required solution and returns the result. LISP can use UNIX as a back-end processor running a Fortran, C or Pascal routine in support of a much larger LISP application.

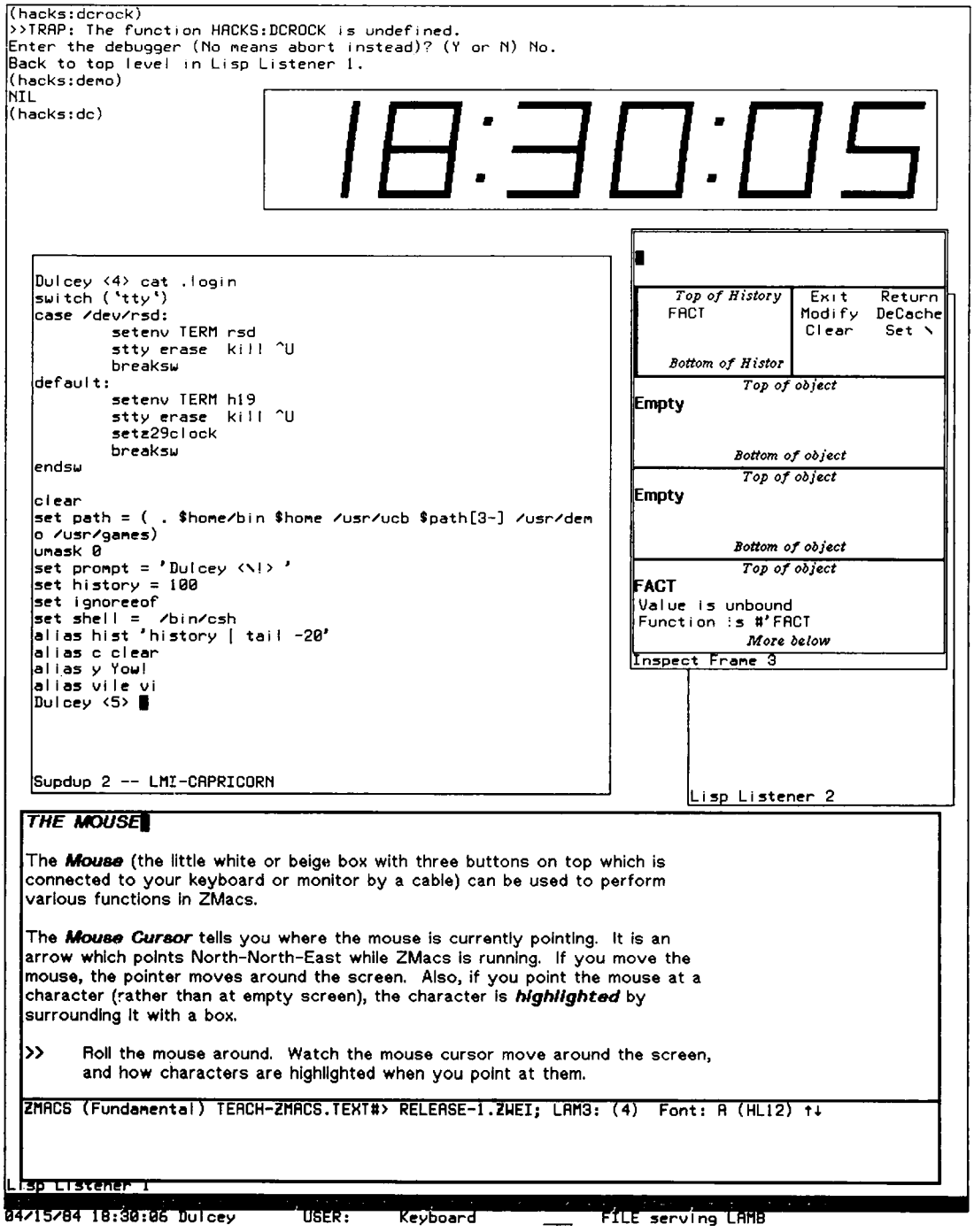


Figure 14-1. A Screen with Multiple Windows.

Chapter 14

ZetaLISP-Plus Utilities

14.1 The Window System

The Window System allows the creation and placement of separate virtual screens of user-chosen size anywhere on the display. Each window may have a different process associated with it. Windows can be resized or moved, or may partially or totally overlap one another. This allows the display to be in the most convenient format for your use.

Mouse and menus are fully integrated within the system for maximum convenience and efficiency. To manage windows, the user has access to a sophisticated set of menu-driven routines allowing him or her to create windows at will and to switch among any number of windows in the system. A simple mouse command invokes the menu; the user need only select the desired window by pressing a button.

14.1.1 Applications

Among typical uses for the Window System are:

- Maintaining both editing and execution environments side by side on the screen. During the debugging cycle the Window System allows bouncing back between a ZMACS window, containing program source, and a LISP window containing its execution.
- Maintaining several editing windows simultaneously on the screen, insuring that specific editing changes are reflected in dependent code.
- Displaying your own applications and/or menus.

14.1.2 Customizability

Although its extensive features are sufficient for almost any application, the Window System is also completely user-customizable. You may choose from any of the standard features available within the Window System or build new ones. This is possible because the Window System is created with FLAVORS, a message-passing capability integrated into the LISP System, so that defining the specific behavior required for a window is easily done.

Existing functionalities of the Window System include:

- Customization of screen commands and behavior.
- Generation of
    * Screen objects such as borders and labels

    * Screen graphics such as boxes and rectangles
    * Menus

— Ability to represent multiple fonts within a given window.

— Definition of items such as

    * "Mouse-sensitive items" (screen objects that are sensitive to being pointed at by the mouse)
    * Blinking objects
    * Areas to be handled as a unit

— Text and graphics manipulation primitives, including character deletion, scrolling, "**MORE** Processing" on non-scrolled windows, and automatic clipping of text and graphics.

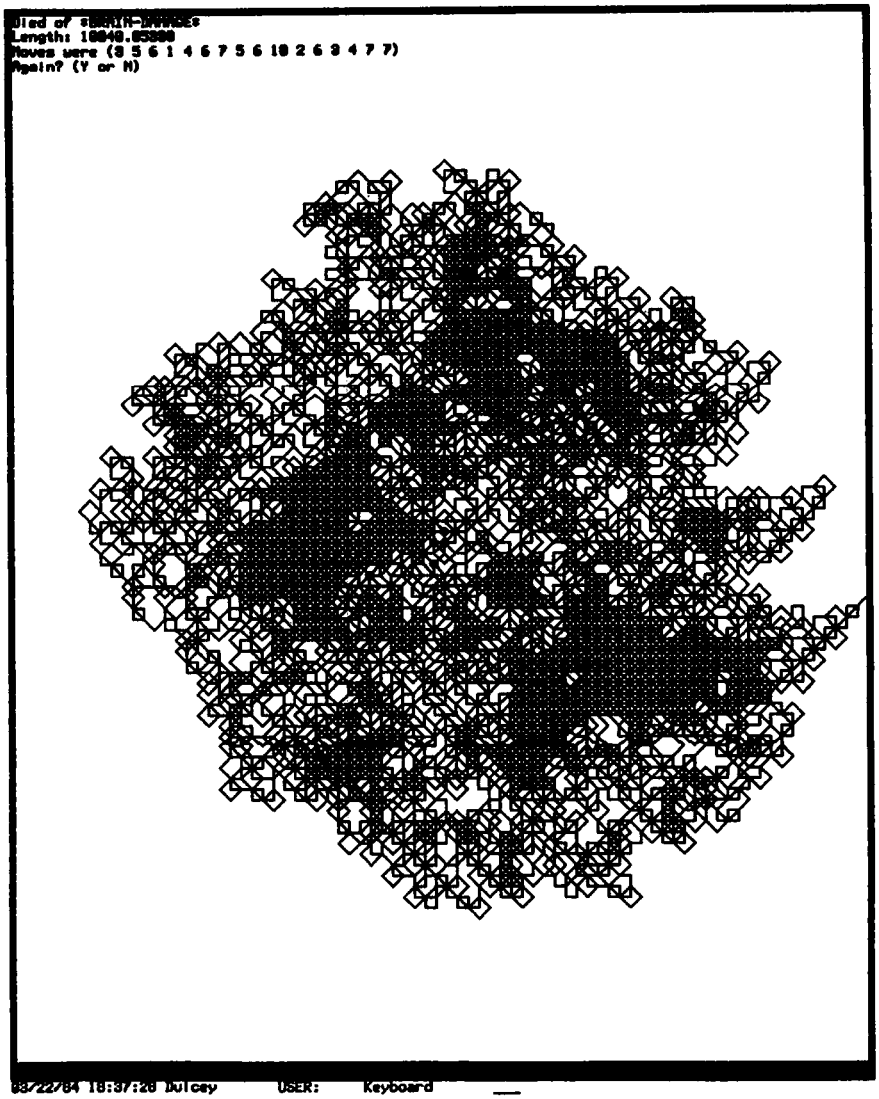


FIGURE 14-2. Graphics on the LMI Lambda.

14.2 ZMACS, the System Editor

At the heart of ZetaLISP-Plus is ZMACS, the system's display-oriented, real-time editor. The power of ZMACS derives not only from its extensive abilities to edit text and LISP, but also from its integration into the global LISP environment. Thus it is both an important system utility and an example of the unified yet modular structure that makes the LISP Machine environment so powerful.

ZMACS accepts over four hundred commands in addition to allowing you to construct your own. Most commands are implemented in easy-to-remember mnemonic arrangement, allowing experienced users great speed. For the novice and the user exploring new features, the system offers extensive online help. Commands may be given on the mouse, as text strings (with automatic name completion), or as a combination of function and letter keys. By using the APROPOS command, you can make a keyword search through the system for all commands dealing with a particular function.

14.2.1 Graphics and Mouse Integration

ZMACS has full access to ZetaLISP-Plus graphics. The mouse is fully integrated into ZMACS, allowing rapid positioning of the cursor and selection of functions with its three buttons. For example, the mouse may be used to designate a section of text to be moved to a new location, to scroll quickly through a file, or to move to a new location within the file.

14.2.2 ZMACS Applications in Your Programming

It is important to note that these features, while utilized by ZMACS, are also available to the programmer for specific applications. Because of the modularity of the LISP Machine environment, many features available in system software may also be incorporated in user programs; you need only call the necessary library routine to use the functionality.

14.2.3 Modular LISP Program Development with ZMACS

In addition to text processing, ZMACS also excels in software development. Because ZetaLISP-Plus is dynamically linked, pieces of program can be changed and independently compiled without the need to relink the entire program. One can edit a function in ZMACS and, with one keystroke, compile it and load it into the current LISP environment. Code that calls that function, or is called by it, need not be touched; one just reloads the edited function and initiates execution.

14.2.4 Source File Accessibility through ZMACS

Complementing the modular editing capability of ZMACS is its ability to retrieve LISP source code. Functions within the LISP environment maintain source file information on their property lists. In this way, a ZMACS user can find and edit the source definition of any function within the system. The function may be specified either by typing in its name or by pointing at the function with the mouse. Similarly, one can list all functions in the environment that call a selected function. After the list appears, one simply "mouses" the desired calling function to edit its definition. By providing easy source code retrieval integrated into the editing process, the LMI Lambda eliminates the need for cumbersome filing systems to keep track of source code text and thus facilitates multiple uses for a given piece of software.

14.2.5 LISP Support through Automatic Functions

ZMACS also knows a great deal about LISP syntax. As you edit a function, levels of parenthesis are aligned and indentation performed automatically by an "incremental pretty-printer." Keystroke commands are available for comands such as "forward one S-expression" or "delete S-expression," in analogy to standard character- and word-manipulation commands. Positioning the cursor next to a right parenthesis causes the corresponding parenthesis to flash, though the system ignores parentheses that are commented out. ZMACS transforms

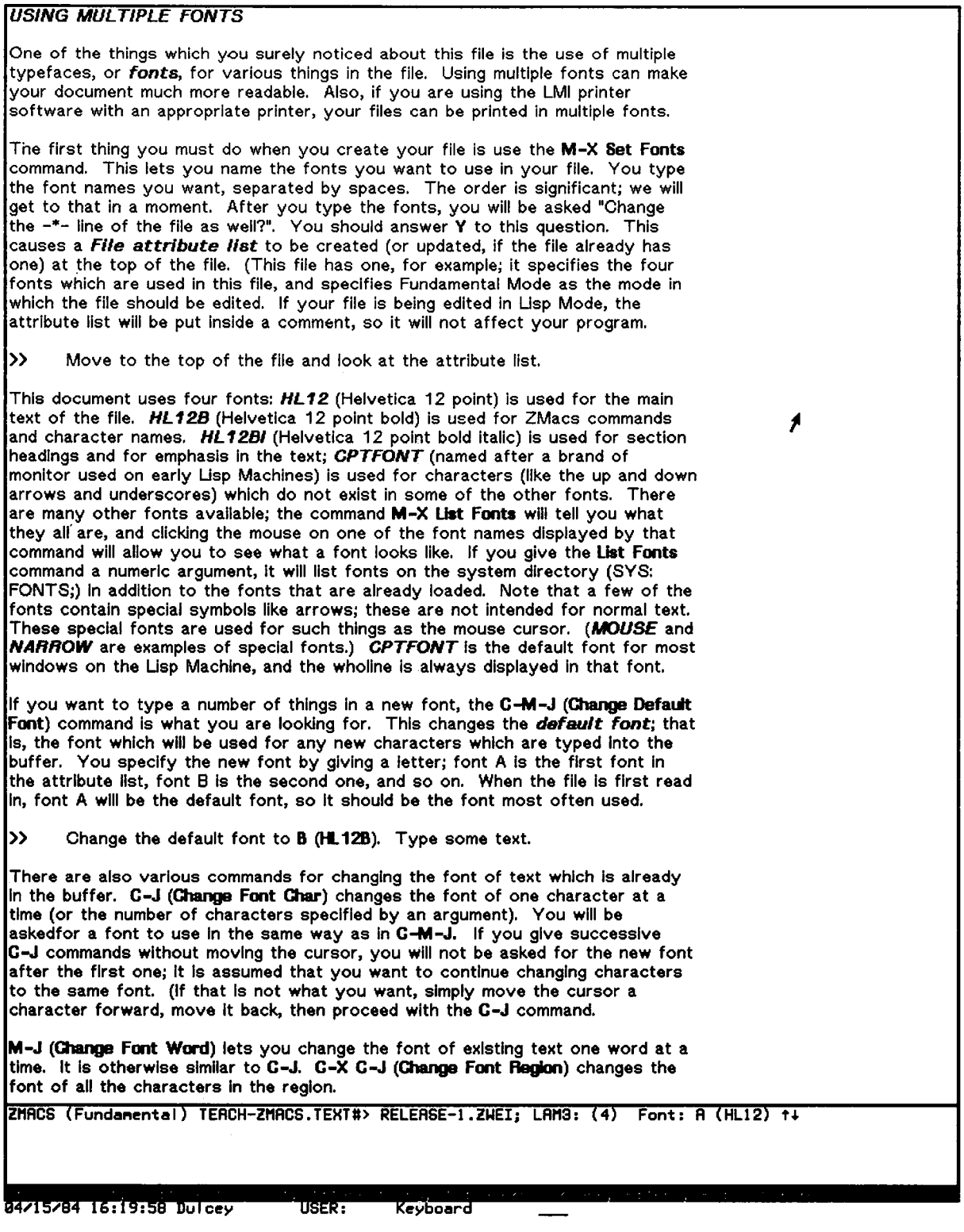


Figure 14-3. ZMACS, Showing Multiple Fonts.

LISP programming from a language annoying to edit into one that is relatively easy. By providing LISP-specific aids, ZMACS encourages readable, elegant code.

Capabilities of ZMACS include:

- Text editing: Full-screen interactive editing, customizable variables; killing, yanking, and moving by means of large-capacity Kill Ring; full undelete functions; searching and replacing
- Buffer-and-file system: Automatic saving of earlier states of editing; minimal file loss in case of difficulties with the system; easy moving of material from file to file
- LISP editing: Common form and code operations accessible with easy keystrokes; pretty-printing of LISP code done with single keystrokes
- LISP code evaluation with a single keystroke in editing mode
- Full graphics integration
- Use of multiple, user-modifiable fonts
- Directory editing, allowing printing, deleting, saving of files from a single buffer
- HELP functions: online tutorial, online documentation of commands and functions through the HELP key
- Automatic command completion or user-selectable choice of completions
- Window operations: selection of single or multiwindow format; customizable multiwindow and multisystem format
- Mouse functions: marking, cursor moving, kill and yank
- Menu-driven functions: select, kill, save buffers; edit, list functions; compile current function; give arglist; change fonts.

14.3 The LISP Listener

The LISP Listener, the window that first comes up when you enter the system after a cold boot, contains the "read-evaluate-print" loop, by which the LISP Machine evaluates functions. The functions of the LISP Listener and ZMACS can be combined in several ways: through a LISP(EDIT) window, a LISP Listener, or a ZMACS window. In ZMACS, a simple keystroke command accesses the read-eval-print loop; in LISP windows, functions are evaluated automatically on completion.

14.4 The Error Handler

Along with a sophisticated editing environment, the LISP Machine architecture also provides exceptional runtime error facilities. Each word has an eight-bit tag field, five bits of which indicate its data type. While data types have normally been limited to numbers and characters, ZetaLISP-Plus provides for a whole host of different data objects besides the more traditional atoms, functions, strings and arrays. Data type information is kept at the architectural levels of the machine to allow data checking and error reporting to be done quickly and efficiently within the system hardware and microcode.

The error handler is activated automatically when it discovers an error in program execution. At this point, the user has two options available. The first is the more traditional approach toward debugging seen in past LISP systems. The user has access to a LISP read-eval-print loop within the environment of the stalled routine.

The programmer can query the system about the various local variables and data structures that are currently active at that point in the computation. While this provides the user with sufficient information about the current software state, it also forces him into querying the debugger about each component of the error environment. To alleviate this problem, the LISP Machine provides users with the Window Error Handler.

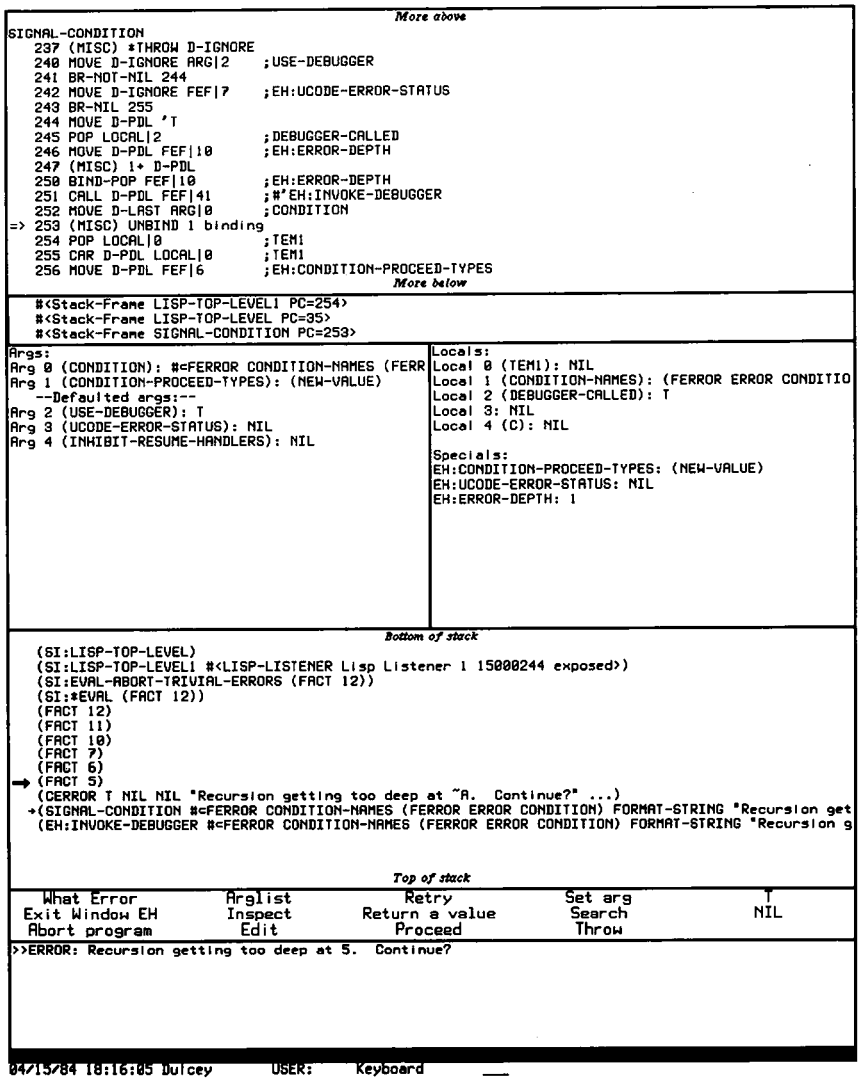


FIGURE 14-5. The Window Error Handler.

14.5 Graphics-Oriented Window Display: The Window Error Handler

The Window Error Handler is a graphically oriented display of the offending functional environment. The screen is divided into areas representing currently active functional arguments and local variables, a history of the source-level expressions that have been evaluated, and the current macrocode environment and stack frame. The programmer can either point to various objects with the mouse to expand their contents or evaluate atoms or expressions within the LISP error environment. In addition, ZMACS is integrated within the Window Error Handler, allowing easy editing and reloading of the offending source code.

14.6 The Inspector

Examining complex data structures within LISP has always been problematical. Earlier LISP systems required that the debugger CAR and CDR his way through complex pointer structures with little or no system help. The Inspector provides a graphically oriented tool through which you can examine and modify complex structures carefully and conveniently.

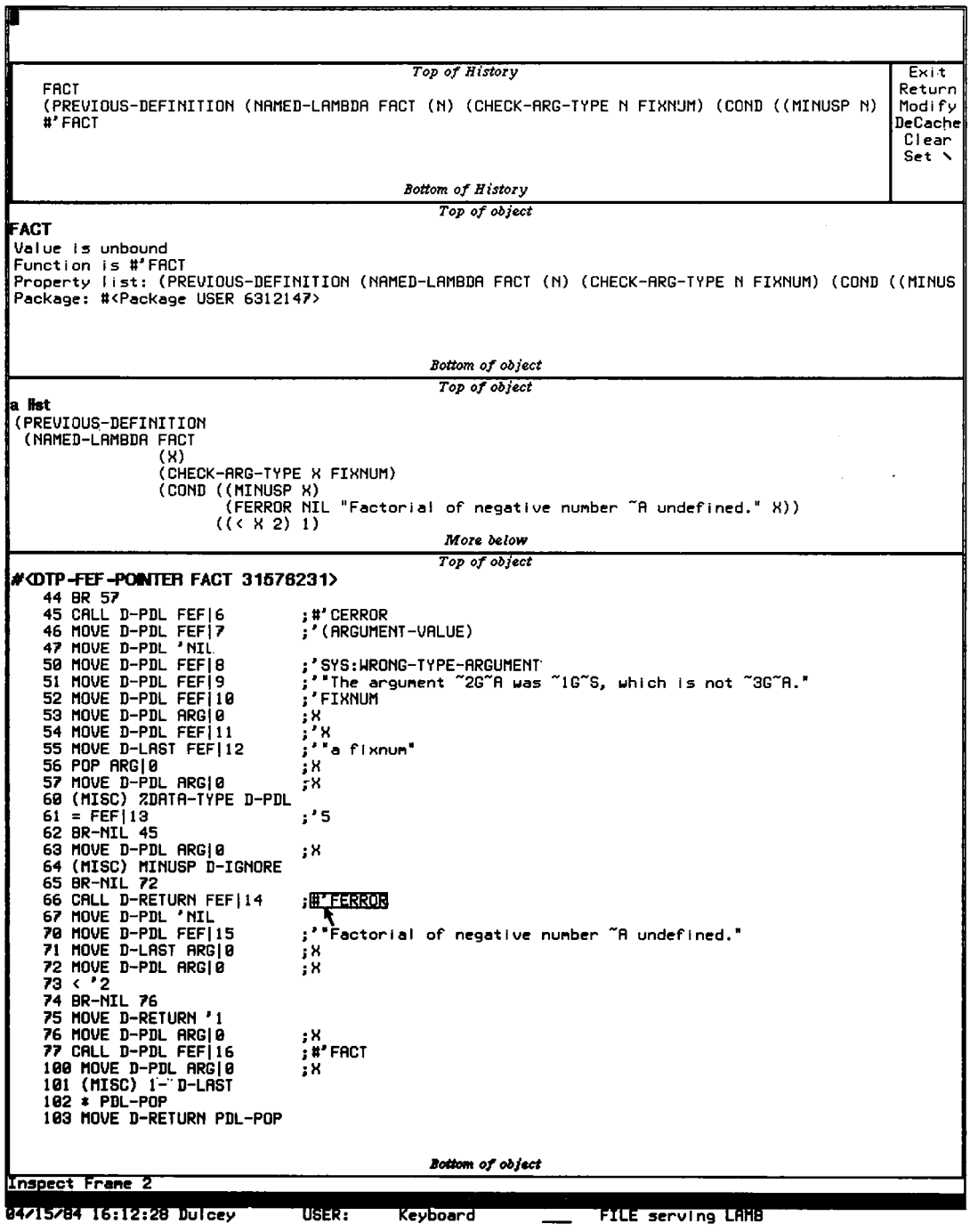


Figure 14-6. The Inspector.

The Inspector (like other LISP Machine subsystems) consists of a window further divided into a number of areas. At the top of the screen is a LISP read-eval-print loop providing an environment for evaluation. Next is a history list providing a full record of all objects visited during the course of the Inspect session. The final three areas contain full expansions of the three most recently examined objects.

To operate the Inspector, you either type an object into the LISP read-eval-print loop or, more often, point to an existing object on the screen. The object is then placed on the history list and expanded. Based on what is seen, the user can then repeat the process.

Incorporated into the Inspector are many of the text-handling features resident in ZMACS, for instance, utilization of the mouse to scroll through long structures or to move proportionally through a structure. (Actually, the same code is used, which makes the Inspector an excellent example of how easy it is to incrementally build applications software within the LISP System.)

14.7 The Garbage Collector

Earlier LISP implementations periodically paused while garbage collection took place. With the advent of more sophisticated LISP systems, these pauses became unacceptably long.

The LISP processor solves this problem by offering a parallel garbage collector. Executing concurrently with other LISP Machine processes, the garbage collector does its job incrementally, allowing LISP computation to proceed without delay. You need not fear that a complete garbage collection might occur during a time-critical piece of code. With the addition of a parallel garbage collector, ZetaLISP-Plus allows LISP to directly compete with other "real-time" programming languages, but still retain its enriched programming environment.

Additionally, the garbage collector can distinguish memory space according to four levels of volatility. Memory space that changes state frequently is garbage-collected most often; relatively static areas, least often. This innovation allows the garbage collector to concentrate its resources on the most "collectible" areas.

14.8 The File System

The LMI Lambda LISP file system allows users easy access to files residing locally on the system disk and to remote hosts. By defining a uniform file syntax that incorporates a host declaration, users have a general mechanism to access files network-wide. Features such as automatic name completion and wildcard characters are available even though these facilities may not be native to the host.

For files residing within ZetaLISP-Plus, redundancy and protection are key. Files have version numbers and the system retains a specified number of generations per file. Additionally, you may take advantage of a two-level delete scheme, allowing both a delete and a non-reversible delete. At the lower levels there is totally redundant information within the file blocks so that the file headers can be totally recreated if need be. Additional file routines include a scavenger to collect displaced file blocks and an incremental/total tape dumping system.

14.9 ZMAIL

ZMAIL, the highly functional LISP System mail system, provides service both locally within individual LMI Lambdas and over the network. Designed in integration with ZMACS and the Window System, ZMAIL controls mail processing by simple mouse movements and menus; there is no need to learn special groups of functions.

With ZMAIL you can

- Delete or undelete a message
- Save messages on a file, selectively or globally

— Reply to an existing message

— Send new mail

— Move to another message by pointing to its entry in a summary window

— Define groups of persons to receive mail]

ZMAIL works in the full ZMACS environment, giving you the complete power of the system editor in generating or manipulating messages.

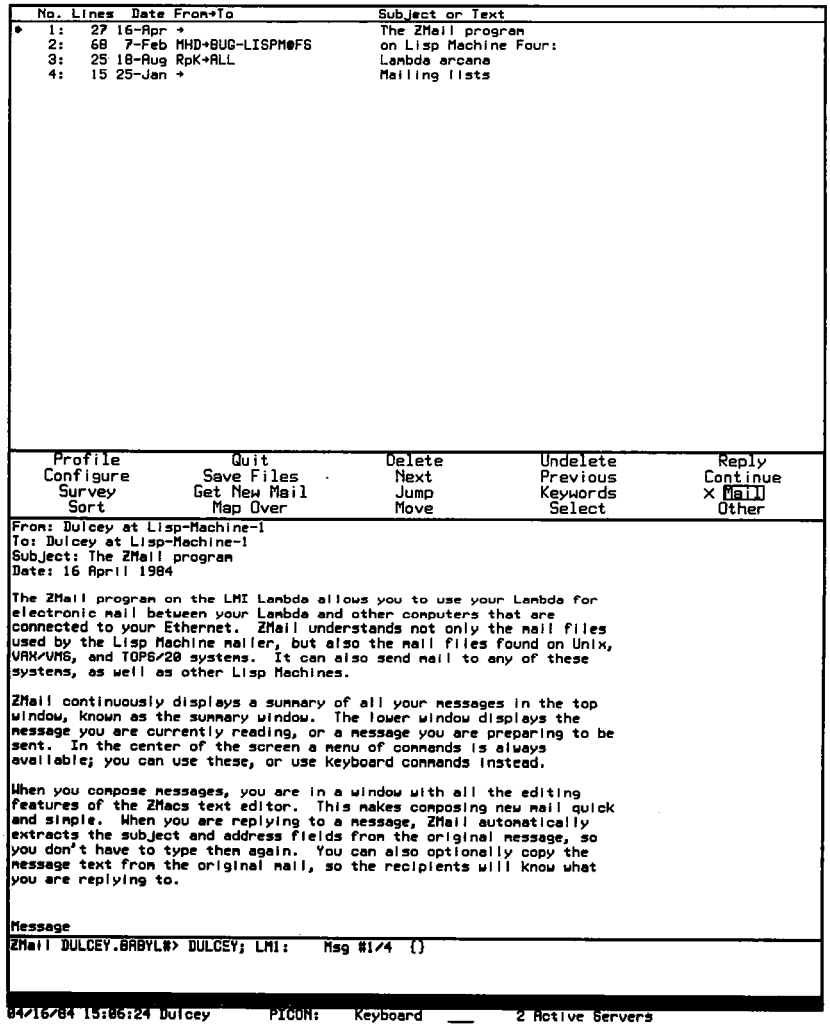


Figure 14-7. ZMAIL.

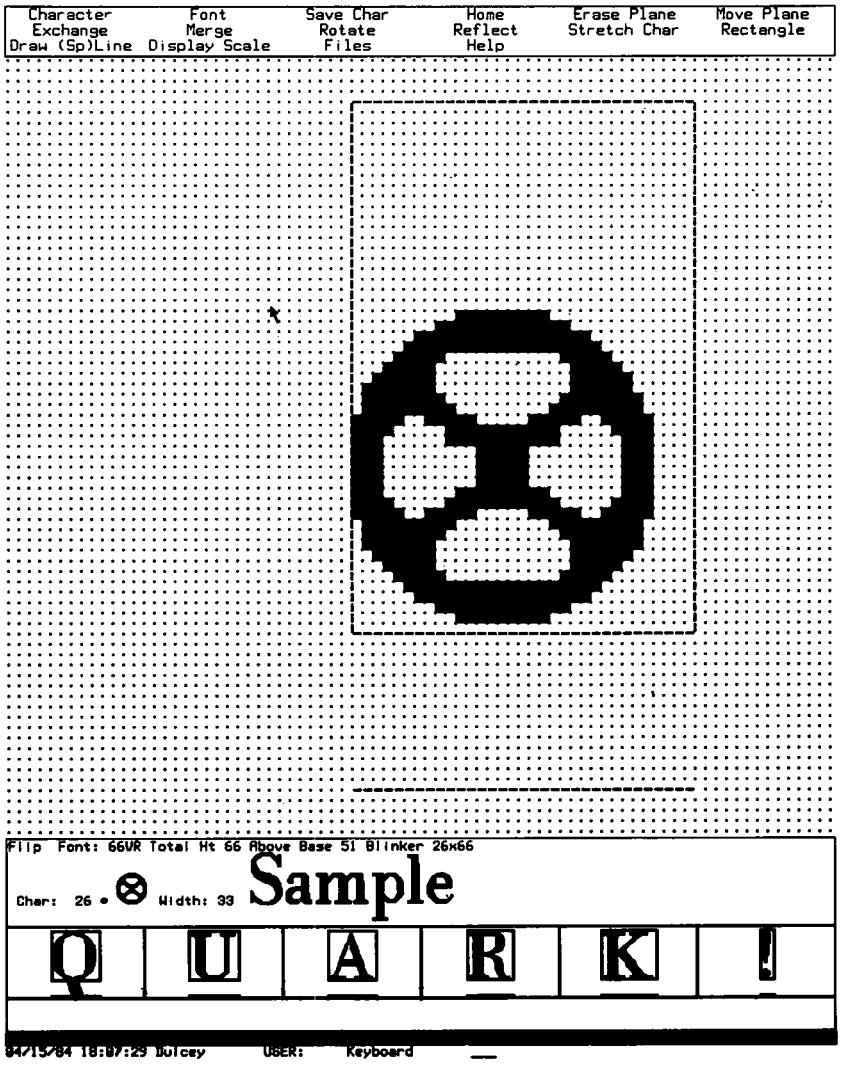


Figure 14-8. The Font Editor.

14.10 The Font Editor

ZMACS includes a generous number of standard fonts. However, users may design their own fonts or icons with the Font Editor. The Font Editor allows you to design them from scratch or to alter them from another font or icon. The Font Editor is fully mouse-integrated; among its graphics capabilities are mirroring, reversing, stretching and shrinking, character merging, and rotation. User-designed fonts may be saved into the ZMACS library of fonts.

In keeping with the ZetaLISP-Plus philosophy of design integration, the font editor is integrated with (and is actually a subsystem of) the graphics system. All fonts are generated through bit-raster graphics, maximizing compatibility.

The graphics capabilities integrated into ZetaLISP-Plus include not only the font editor but functions for drawing lines, geometric objects, mathematical graphs, logic gates, and many other useful objects.

Chapter 15

ZetaLISP-Plus Control Structures

15.1 Stack Groups

The LISP processor is a single-user, multi-process system. Separate processes are easily defined and controlled by use of the Stack Group feature.

A Stack Group holds all the variable bindings and control stack for each process, defining its state at any point in time. With the Stack Group feature, you may

- Execute background processes
- Generate a prioritized agenda of tasks to complete
- Do process scheduling via simple source level predicates
- Implement coroutines

15.2 Macros

Macros act similarly to LISP functions, but produce another LISP form that is evaluated in place of the original macro call. Within the macro body you have access to the full LISP processing environment. Routines can be as simple or as complex as you desire, and may be compiled. Macros are fully recursive; macro-defining macros are not uncommon. Macro expansion takes place at evaluation time in the case of interpreted functions; during compilation it takes place at meta-evaluation time.

15.2.1 Backquotes

A heavily used feature in macro programming, and a useful facility in general, is the backquote. Selectively turning off evaluation, the backquote allows easy construction of LISP forms that combine constants and functional expressions. Overly complex S-expressions containing multiple levels of LIST and CONS are no longer required. Instead, the elegant notation of quotes and backquotes achieves maximum clarity without sacrificing efficiency.

15.3 Packages

A problem in the implementation of large, multi-author software projects is the coordination of function and variable names. Everyone wants to use names generic to their application. Past LISP implementations determined which function body to execute based on a unique function name within the entire virtual address space. If the same name existed in more than one place within the address space, one definition would overlay the other, with unpredictable results.

15.3.1 Packages as Separate Name Spaces

To fix this problem, ZetaLISP-Plus structures the programming environment into "packages". Functions and variables continue to share a common data area, but each package deals individually with the symbols belonging to it. Within multi-author projects, you can implement your code within a separate name space that corresponds to a separate package. If you require service from another package, you simply make an explicit reference to the package and function you desire. In this fashion, large and complex systems are easily integrated, accessing common data, without conflicts among low-level routines.

15.3.2 Package Hierarchies

Packages can also be hierarchical. If a function cannot be resolved within the current package, ZetaLISP-Plus initiates a search procedure, examining the next package up the chain. In this way inheritance schemes can be implemented, defining attributes of a package in terms of its ancestors.

15.4 Message Passing—FLAVORS

FLAVORS are a way to associate code with data structures and to construct new FLAVOR objects out of currently existing ones. FLAVORS, called like functions with keywords or messages as arguments, search through their associated methods for one which handles that particular message. If no method is found, the message is referred to the various FLAVORS that make up the object. Eventually, either a method is found and its code executed, or an unclaimed message error is signaled.

15.5 FLAVORS and the Window System

Much of ZetaLISP-Plus software, particularly the Window System, uses FLAVORS. Each window is an instantiation of a FLAVOR, inheriting properties from the component FLAVOR "tv:window", which defines the generic behavior of any window. Typical methods supported for windows include accepting characters from the keyboard, drawing lines, or controlling the mouse.

One can also define groups of methods as a unit. While not qualifying as a full-fledged FLAVOR, these groupings (MIXINS) modify FLAVORS in consistent ways. They are additives rather than wholes.

15.6 STREAMS

Another powerful use of the FLAVOR system is in STREAMS, by which standard input and output get passed between different programs and processes. STREAMS are conceptually similar to UNIX PIPES, but much more versatile, since each is a full FLAVOR instantiation. Using existing FLAVORS and MIXINS, one can easily create an appropriate interface to exactly suit the particular application, with backup methods automatically invoked to handle any messages for which the optimized operations don't apply. Once a STREAM is created between two entities, information flows quickly and efficiently, and is easily redirected if required.

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

LISP Machine Inc.
Suite 900
6033 West Century Boulevard
Los Angeles, California 90045
(213) 642-1116
Telex 664-608

LISP Machine Inc.
1000 Massachusetts Avenue
Cambridge, Massachusetts 02139

The LMI Lambda
Field Service Manual
July 1984

Published by LMI 6033 W. Century Blvd. Los Angeles CA 90045
USA

This manual is current for the STREAMS Release 1 software. It will be updated as necessary to accommodate revised software.

To ensure your satisfaction with the LMI Lambda, LMI maintains a Customer Service Help Line, open Monday through Friday, 9 AM-5 PM (Eastern time). This service is free and open to all LMI customers. Call the Help Line, 617-876-6819, if you have bug reports or questions.

This manual printed July 1984.

Information in this document is under development, and may change without notice.

Table of Contents

EMERGENCY POWER DOWN

IMMEDIATELY:

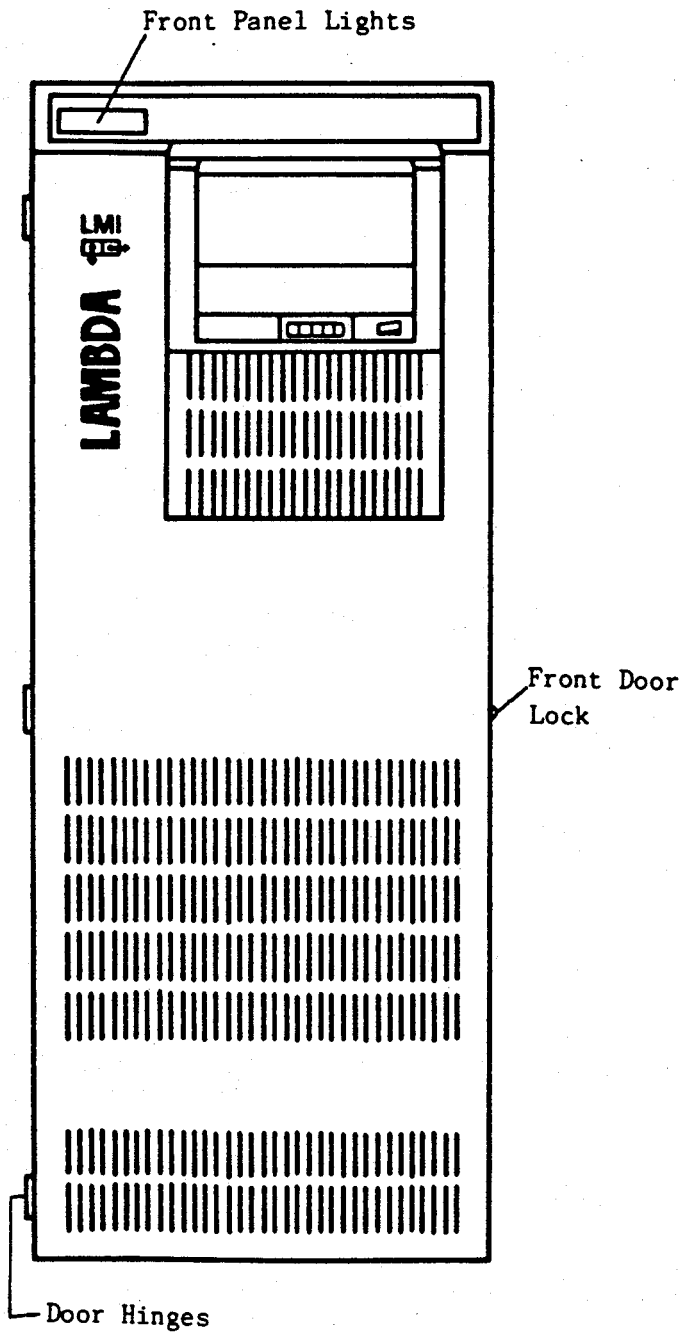
- Turn off (down) the main power switch on the back of the LMI Lambda, or
- Unplug the LMI Lambda's main power cord.

Important: Flipping the switch is faster, since the power cord has a twist-lock plug.

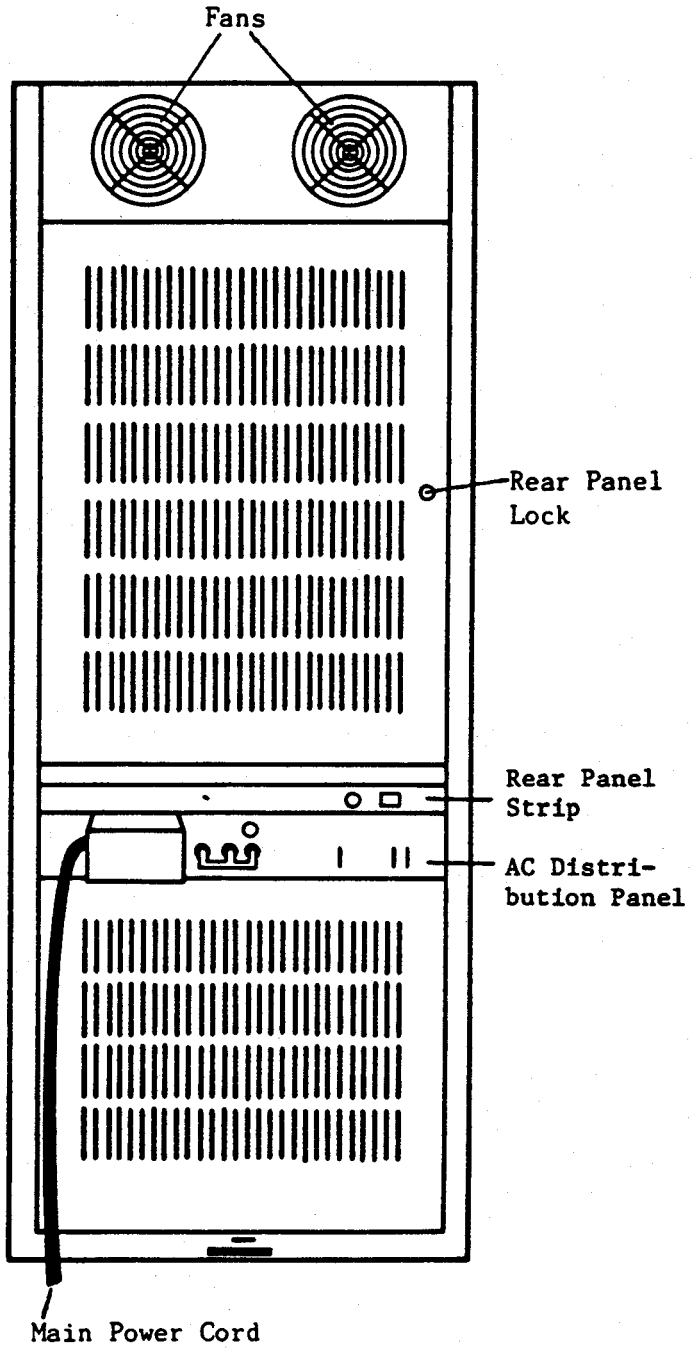
In case of an emergency, such as smoke, fire, or exposure to water, this procedure will minimize the chances of damaging your machine.

Once you have done this, call LMI.

FRONT VIEW OF LMI LAMBDA
WITH CIPHER 1/2" TAPE DRIVE



REAR VIEW OF LMI LAMBDA



Chapter 1

SOME DOCUMENTATION BASICS

1.1 Conventions in Documentation

By convention, phrases such as "type to (or at) the SDU" refer to operations that must be done using the Zenith Z29 console terminal or equivalent, before either LISP or UNIX is booted.

Unless otherwise stated, each command is followed by a <RETURN>. In multiple commands, each end-of-line signals a <RETURN>.

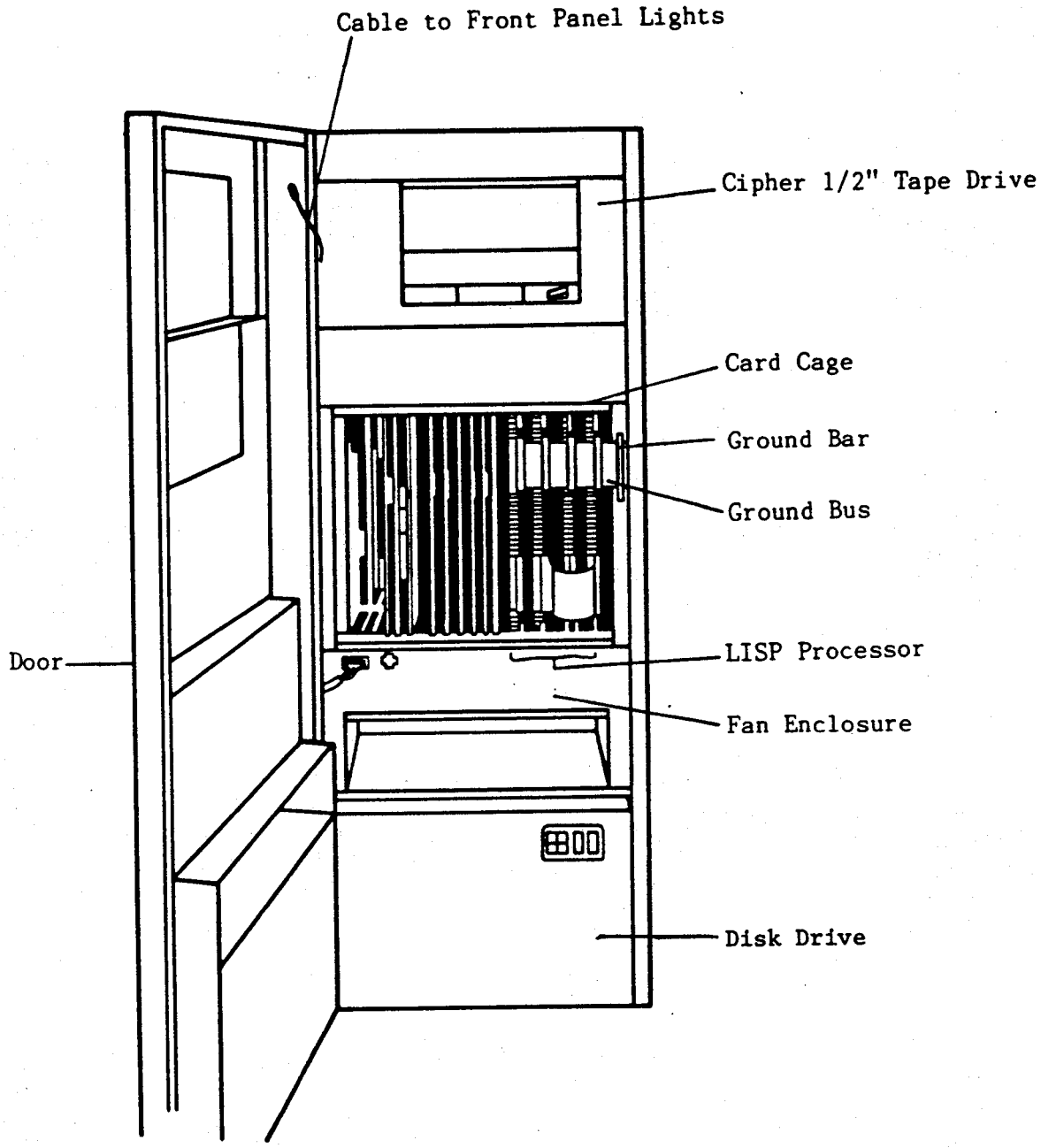
1.2 LMI Policy on Software Updates

LMI documentation is updated by revisions and expansions of hardcopy documentation, by continuing updates of online documentation, and by the LMI PROGRAMMERS' NEWSLETTER, sent free to all LMI Lambda and CADR owners.

This manual covers features that we anticipate will be changed in the next few months, particularly after the next release of the UNIX system. To keep up with current software changes, free updates of this manual are provided promptly to all customers.

Please feel free to call LMI's Customer Service Help Line, 617-876-6819, to get answers to your questions about the system.

FRONT VIEW OF LMI LAMBDA WITH CIPHER 1/2" TAPE DRIVE, DOOR OPEN



Chapter 2

INTRODUCTION

The LMI Lambda is a modular system with a standard hardware configuration consisting of a system cabinet, high-resolution display terminal, console terminal, and connecting cables.

System installation should be performed by LMI personnel only. All installation instructions contained in this manual are solely for customer reference.

This manual provides customers with site preparation instructions, which must be followed before a system can be installed, and with procedures customers may use for troubleshooting problems.

To facilitate its use, this manual has been divided into six sections, as follows:

- LMI Lambda System Components
- Site Preparation
- Installation
- System Operation
- Troubleshooting
- Preventive Maintenance

Chapter 3

LMI LAMBDA SYSTEM COMPONENTS

3.1 Major Components

The LMI Lambda System consists of a main cabinet, high-resolution monitor, console monitor, and Ethernet transceiver.

3.1.1 Main Cabinet

In addition to filters, fans and cabling, the main cabinet contains the following:

- AC Distribution Panel
- Rear Panel
- Disk Drive
- Card Cage
- Tape Drive

The monitors, keyboards, and mouse are attached to the main box by cabling.

3.1.1.1 AC Distribution Panel

This unit contains a main power rocker switch, main power cable, 10 Edison-style plugs, 3 remote control jacks, and a 250V, 1/8 A slow-blow fuse.

3.1.1.2 Rear Panel

This unit, directly above the AC Distribution Panel, holds the rotary switch and LMI Lambda RESET button.

3.1.1.3 Power Supplies

The two power supplies are plugged into the AC distribution panel. The smaller supply produces +/-12V DC power to the card cage and the larger one supplies the card cage with +/-5V DC. "DC OK" and "AC OK" signals are also produced by the supplies to protect the SDU's CMOS RAM against power loss.

3.1.1.4 Disk Drive

The Fujitsu Eagle Winchester M2351A Mini-Disk Drive plugs into one of the two sockets at the far right of the AC power distribution panel. Since it contains its own DC power supply, it does not draw power from the card cage power supplies.

3.1.1.5 Card Cage

Slots in the card cage are numbered from 0 at the right to 20 at the left, as viewed from the front of the machine. The card cage contains two busses, the industry-standard Intel Multibus and the NuBus. The Multibus, slots 13-20, has standard 16-bit data and 20-bit address space, and is used for connecting standard peripherals. The NuBus includes slots 0-15 on the backplane. To maximize user flexibility, slots 13-15 can be used for either the Multibus or the NuBus.

IMPORTANT: The private backplane for the wirewrap LMI Lambda LISP processor occupies slots 0-7 on the backplane. For PC board LISP processors, the backplane takes up slots 0-3. To avoid damage to the system, insert only LMI Lambda boards in these slots.

3.1.1.6 System Diagnostic Unit--SDU

This board, which occupies slot 15 of the card cage, is the source of the NuBus clock and is the bus connector between the NuBus and the Multibus. Features include the following:

- Intel 8088 microprocessor chip
- PROMs and monitor to run diagnostics, boot LISP or UNIX, and control the system's 1/4" cartridge tape streamer interface
- CMOS RAM and rechargeable battery
- Analog to digital converter
- Two RS232 ports

- Bus converter

3.1.1.7 LMI Lambda LISP Processor

The LISP processor consists of four boards, referred to as the RG, MI, CM and DP boards.

Together they form an independent processor for LISP, comparable to the 68000 processor board for UNIX. More information on the LMI Lambda LISP processor is available in the LMI Lambda Technical Summary.

3.1.1.8 Memory Board

The Memory Board contains a 1/2-megabyte memory bank as well as error correction circuitry (ECC), which corrects single-bit memory errors and detects 2-bit memory errors. The memory is capable of 2-, 4-, 8-, or 16-word transfers.

3.1.1.9 Multibus Boards

Multibus boards, which are mounted on triple-high Eurocards ("carrier cards") for standard size and ease of insertion, include the following:

Disk controller board
: The LMI Lambda uses Interphase Corporations's SMD 2181 disk controller. It runs the Fujitsu Eagle disk drive and plugs directly into the Multibus, serving as a bus master during data transfer.

  The board contains several dip switches and jumpers, which have been preset and should not be altered. Information on the dip-switch settings and jumpers is contained in Appendix C for reference only; users are requested to call LMI rather than reset dip switches or jumpers themselves.

  Resetting of dip switches should be necessary only during the installation of an additional disk controller, when a system is expanded to more than four disk drives.

Tape controller board
: Manufactured by Ciprico, the Tapemaster 1/2" magnetic tape drive controller is Multibus-compatible and controls up to eight 1/2", formatted, start-stop or streaming tapes. It functions with 16-, 20-, or 24-bit addressing,

single or multiprocessor.

Ethernet controller board
: Manufactured by 3-Com, it conforms to DEC, Intel, and Xerox 1.0 specifications.

3.1.1.10 Optional Boards

Many optional boards are available. Typical configurations might contain:

- The 68000 UNIX processor board, which occupies slot 11, containing one 68000 chip and a 4K word cache.
- A medium resolution color monitor board.
- Additional optional memory. Present memory boards have a capacity of 1/2 megabyte (128K words) of storage; 1- and 2-megabyte memory boards are under development.
- The Multibus carrier card for user-supplied peripherals, occupying one slot in the card cage for each designated peripheral. In most systems, Multibus carrier cards for the disk drive, tape drive, and Ethernet occupy three of the Multibus slots.

The Multibus is designed to accommodate a wide variety of user-supplied peripherals. However, LMI cannot guarantee proper function of peripherals not installed by its representatives. We suggest contacting LMI before installing peripherals.

The NuBus is also designed to accommodate user-supplied peripherals. At this time their production and distribution is limited to LMI and Texas Instruments; we welcome your interest in developing or interfacing user-supplied peripherals for the NuBus.

3.1.1.11 Tape Drive

The tape drive, Cipher Data Products Model F880 1/2" magnetic tape streamer unit (MTSU), is a dual-speed, 1600-bpi tape transport. The unit features dual-gap head and read-after-write capacity, with read/write, control, and formatting electronics incorporated in one printed circuit board (PCB).

The Quarterback 1/4" cartridge tape drive is a low-cost, high-performance option. The interface for this tape is included on the SDU board; thus an additional tape controller board is not required. Each tape drive provides up to 20 megabytes of user data storage and can record and read in streaming mode at 8,000 bpi.

3.1.1.12 Cabling

The following cabling, supplied with the LMI Lambda System, must be checked before installation for proper placement.

The MAIN POWER CABLE has a 30A twistlock plug.

The AC DISTRIBUTION PANEL has the following cabling:

- Two coaxial cables connect the AC distribution panel to the backplane.

- Power cables connect the AC distribution panel with the power supplies, fans, disk drive, and tape drive.

The BACKPLANE is cabled as follows:

- Regulated DC cable runs to the backplane from the power supplies. The power supplies plug into the AC distribution panel.

- If ethernet is ordered, a bulkhead cable connects the Ethernet controller to the Ethernet port at the back of the machine.

- Two gray ribbon cables connect the tape drive to the tape controller, at the back of slot 17.

- Two ribbon cables, one gray and one multicolored, connect the disk drive the disk controller, at the back of slot 16.

- A bulkhead cable connects the VCMEM board to the high resolution port at the back of the machine.

- The high-resolution monitor output cable runs from the paddlecard at the bottom of slot 8 to the connector on the backpanel. The R G & B cables for the medium-resolution color monitor (optional) run from the connector in the bottom of slot 13 to the backpanel.

Four cables are attached to the SDU paddlecard in the bottom connector at the back of slot 15. Each is keyed so that it cannot be plugged in upside-down. Still, it is fairly easy to skew these cables when inserting them, so check carefully. The cables go:

1. From the power supply sensors to the "P/S" plug on the SDU paddlecard.

2. From the front panel lights to the "Front Panel" plug on the SDU paddlecard.

3. From the rotary switch and the RESET button at the back of the machine to the "Rear Panel" plug on the SDU paddlecard.

4. From SDU ports A and B to the "Serial I/O" plug on the SDU paddlecard.

3.1.2 High-Resolution Monitor (RSD)

A Moniterm black-and-white, high-resolution monitor (800x1024 pixels), standard AI keyboard, and optical mouse are provided.

3.2 Medium Resolution Color Monitor (optional)

The LMI Medium Resolution Color Monitor is a Mitsubishi C-3910LP, fully integrated into the Lambda LISP environment through custom-written software.

3.3 System Console Monitor

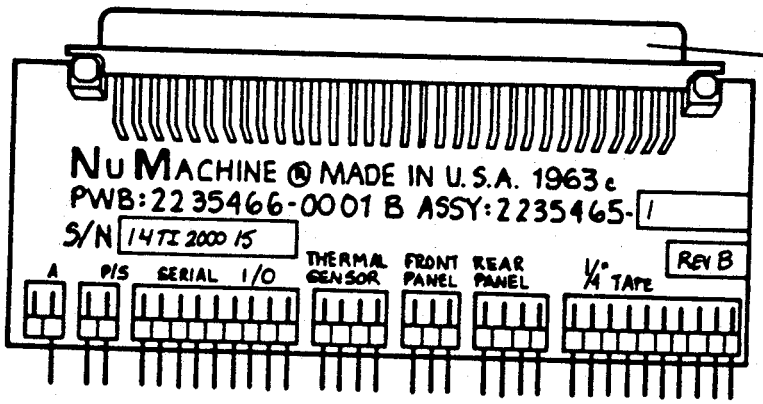
Currently a Zenith Z-29 is provided for system booting and debugging. In future releases this feature will drop out of the system.

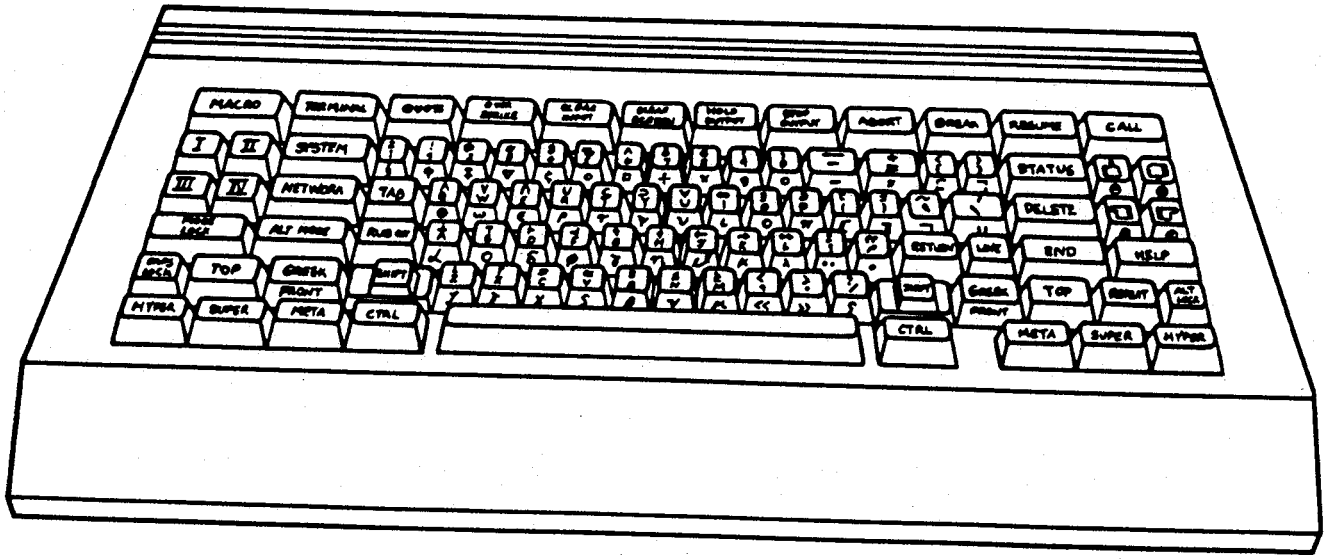
3.4 Ethernet Transceiver

This unit consists of an Ethernet transceiver, cabling, and connectors.

SDU PADDLECARD

(The Thermal Sensor Port is not yet used in the system.)





THE AI KEYBOARD FOR THE LMI LAMBDA

Chapter 4

SITE PREPARATION

General site requirements include cleanliness, uninterrupted air flow, sufficient space for access front and back, and suitable power. To maintain air cleanliness, designation of the installation site as a NO SMOKING area is highly recommended. The computer room should have "clean" power and good grounding to avoid crashes and system malfunctions. Air-conditioning should be sufficient to maintain the machine room at or below 75 degrees Fahrenheit.

To prevent line noise, the length of cabling from the high-resolution monitor to the main cabinet should not exceed 150 feet.

4.1 System Dimensions and Weight

Uncrated, the LMI Lambda Rack System main cabinet stands 60" high, 35 1/2" long and 21" wide. Crated dimensions are 70" x 44" x 36". Weight is approximately 500 lbs. crated, including the tape and disk drives. Uncrated, the monitor dimensions are 16" high, 13 1/2" long, and 11 7/8" wide. Crated dimensions are 24" x 20" x 20". The LMI Lambda office system stands 26" high, 21 1/2" long and 27 1/2" wide.

4.2 Ambient Requirements

4.2.1 Temperature

The LMI Lambda should be operated at between 13-33 degrees C (55-85 degrees F); temperature changes should not exceed 10 degrees C (18 degrees F) per hour.

4.2.2 Relative Humidity

Relative humidity should be 20-80%, noncondensing.

4.2.3 Power Requirements

Each LMI Lambda should be on its own dedicated line with separate ground. This line should be free of noise, spikes and surges. Each rack system should be on a 30A line; each office system requires a 20A line.

Input Voltage

120 VAC + 10%, -5%. 47-63 Hz single-phase.

Input Current

LMI Lambda, Lambda Plus: 15A nominal, 30A surge
2X2, 2X2P: 18A nominal, 30A surge
LMI Lambda/S: 9A nominal, 20A surge

Power Dissipation

LMI Lambda, Lambda Plus: 6400 BTU/hr
2X2, 2X2P: 7400 BTU/hr
LMI Lambda/S: 3700 BTU/hr

Socket Requirement

LMI Lambda, Lambda Plus, 2X2, 2X2P:
Hubbel 2613 30A 125V twistlock socket or equivalent
(NEMA reference L5-30R)

LMI Lambda/S: standard 20A 3-wire grounding straight blade socket (NEMA reference 5-20R)

Chapter 5

INSTALLATION

•

IMPORTANT: Your LMI Lambda will be fully installed by LMI personnel. The following information is supplied for your reference only. Additional disk drive installation information, including drawings of the machine and cabling, can be found in the Texas Instruments NuMachine Installation and User's Manual.

Note: Parts of the NuMachine Installation and User's Manual do not apply to LMI Lambda installation procedures. In all disagreements, use this manual as the definitive word on LMI Lambda installation.

5.1 General Installation

Unpack the LMI Lambda from its shipping crate. Save all shipping boxes, if possible, since that will facilitate returns in case of shipping damage or malfunction. Check for external damage in shipping, for instance dents or any other indication that the system has been dropped or roughly handled.

Open the front panel door and inspect for internal damage:

- Devices snapped from their mounts
- Broken boards
- Cables improperly attached.

5.2 Disk Drive Installation

For shipments within the United States, the Fujitsu Eagle Disk Drive is mounted in the rack. For shipments overseas, it is packaged separately. Please disregard the section on "Rack-Mounting the Disk Drive" if it is already mounted.

Detailed information about the Fujitsu Eagle and diagrams of it are in its manual, which you have received as part of your LMI Documentation Package. Please refer to it as necessary.

5.2.1 Rack-Mounting the Disk Drive

Installing the Fujitsu disk drive in its rack requires two or even three people. This is the recommended procedure:

1. Remove the disk drive from its shipping container.

2. The lower portion of the main cabinet contains slides on both sides. Pull these out until the latch mechanism activates.

3. Mount the disk drive onto these slides and push the slides toward the rear of the cabinet. Verify correct operation of the slide mechanism by pulling the drive in and out of the cabinet. CAUTION: Do not extend the disk drive and tape drive units simultaneously or the machine may fall forward.

5.2.2 Checking the Disk Drive for Shipping Damage

1. Remove the cover of the disk drive and inspect the inside.

2. Check for shipping damage, such as loose or missing parts or foreign matter.

3. Make sure that the printed circuit boards in the unit are properly seated and in the correct positions.

5.2.3 Installing the Disk Drive

1. If the disk was shipped in the machine, just move the machine to its final place. Remove the hardware bolting the disk back to the machine cabinet. Don't lose the bolting hardware; you will need it if the machine ever needs relocation. This allows the disk to be slid forward until it is fully extended in front of the machine. Make sure the cable restraints on the left rear of the disk are loose, so that the disk has enough slack in the cables to allow it to slide forward. Slide disk out until fully extended, remove the black cover (3 screws), and use an 8" Phillips head screwdriver to unlock the disk head. The disk manual explains how to unlock the disk (see "Rotary Actuator Unlocking/Locking," p. 2-4 of that manual).

2. Remove any foam blocks around the disk.

3. Plug in the two disk cables (one wide, one narrow) as shown in this manual. The narrow grey cable goes on the left rear connector on the disk, with the red stripe on the right. The wide multicolored cable goes on the connector just to the right of the narrow cable. The tan and brown twisted pair of wires goes on the right.

4. Plug the disk into the AC distribution panel at the back of the machine.

5. Attach the ground cable shorting plate to the terminals labelled "SG" and "FG2".

6. Replace the disk drive cover and reinsert its securing screws, as described in step 1.

7. Slide the disk drive back into the cabinet.

5.3 Installing Cards and Cables

All cards should be in the machine when it arrives; so should all cables. However, they should be checked for accurate placement and seating.

The four boards of the wirewrap LISP processor go in slots 0-7. The RG board goes in slot 0. The other three boards do not need to be in specific slots, but to avoid confusion we have adopted the following standard:

Slot	Board
0	RG
2	MI
4	CM
6	DP

The H-bus goes from the bottom connector on the front of the RG board to the bottom connector on the front of the CM board. The ground bus connects the top connectors of all four LMI Lambda boards and bolts to the right of the card cage.

All four boards should be pushed carefully into their connectors. These boards tend to warp a bit, so you may have to fiddle with them to push them in.

As with all boards, the component side is on the right.

Each LMI Lambda board has its name, number, and version written on it with a white pen.

The PC LISP processor goes in slots 0-3. The standard slots for the processor boards are:

Slot	Board
0	RG
1	CM
2	MI
3	DP

Although there is an H-Bus cable for the PC boards, there is no ground bus.

The rest of the boards are likewise inserted with the component side on the right.

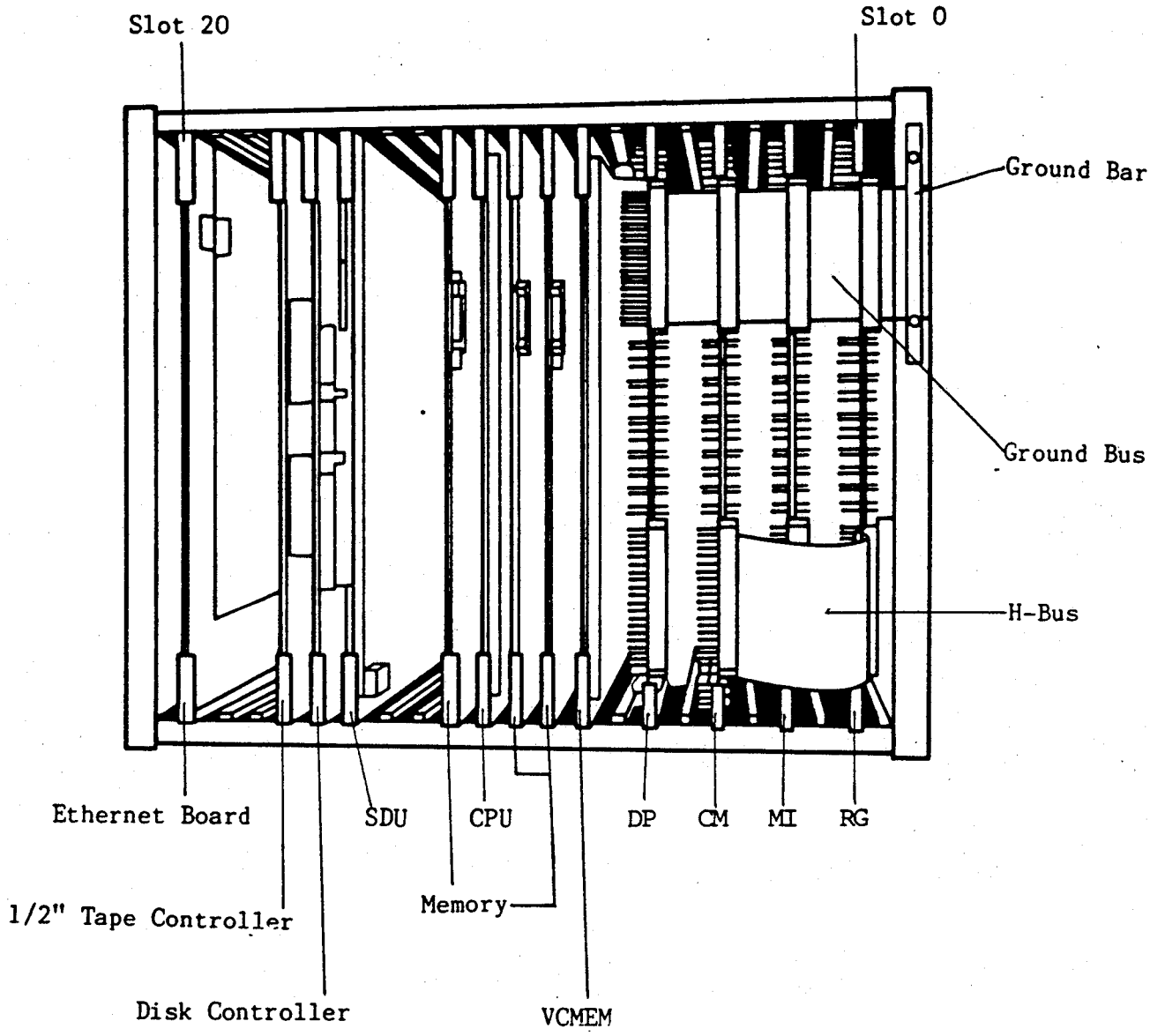
Slot	Board	Distinguishing Features
8	VCMEM	Metal bar on front. 3 PROMs together at top
9	Memory 2	Says "Memory" on front edge
10	Memory 3	Says "Memory" on front edge
11	CPU	Only if UNIX system ordered. Metal bar on front; one 68000 microprocessor chip
12	Memory 1	Says "Memory" on front edge
13	Med Res Color	(Optional)
14	Memory 4	Says "Memory" on front edge
15	SDU	Has battery in corner
16	Disk controller	On disk carrier card. One wide, 1 or 2 narrow cables
17	Tape controller	On tape carrier card. Two wide cables
18	Unused	
19	Unused	
20	3-Com Ethernet	On carrier card. Very narrow flat cable

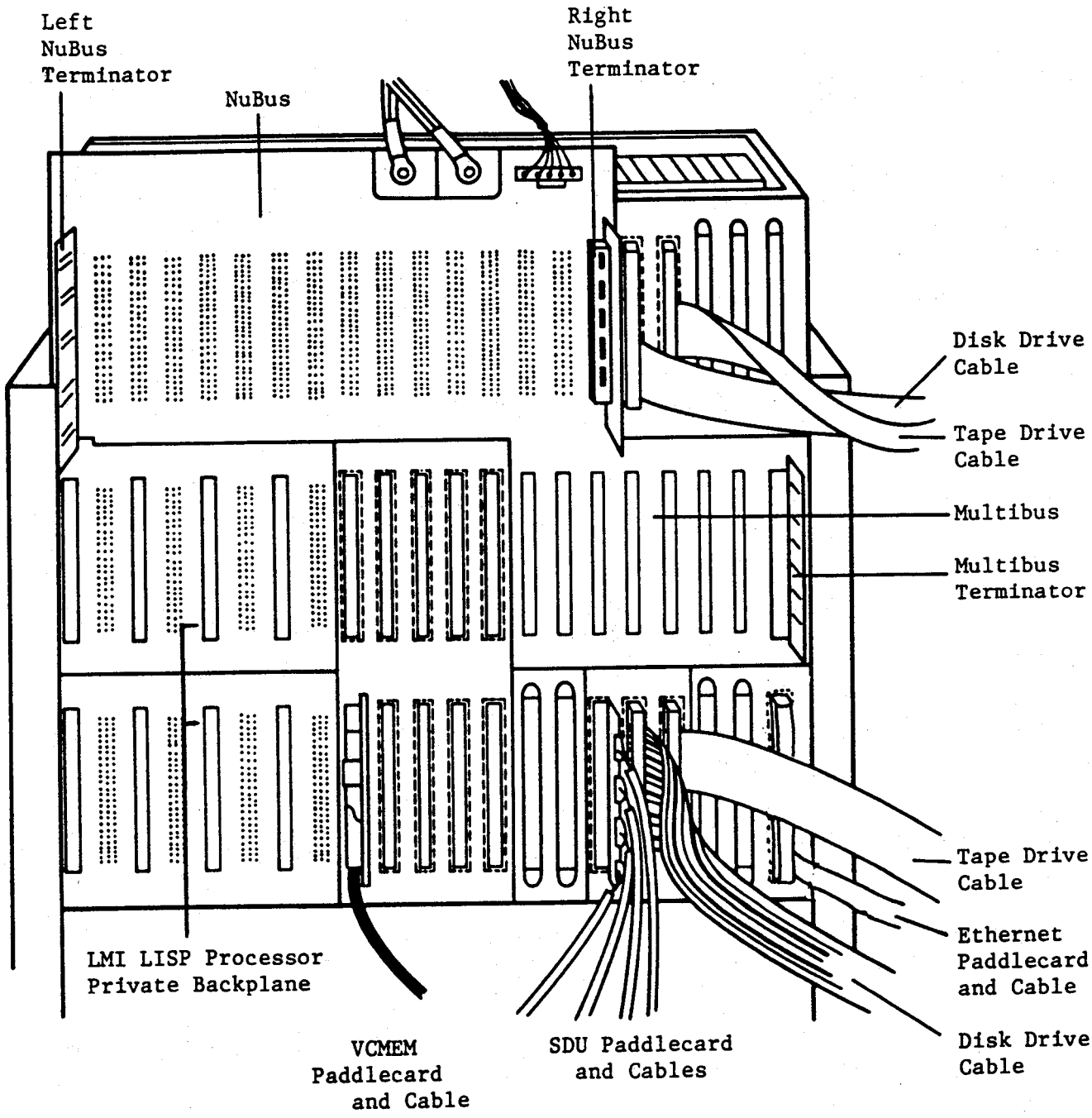
5.4 Installing Paddlecards and Terminators

The left NuBus terminator goes in the top connector in slot 0 at the back of the machine. The right NuBus terminator goes in the top connector in slot 15 at the back of the machine. The only difference between the left and right NuBus terminators is that the left has at its bottom discrete resistors that are absent from the right terminator. Be sure not to confuse these two terminators; although doing so will not damage the equipment, it will make the LMI Lambda function improperly.

The Multibus terminator goes in the middle connector of slot 20, at the back of the machine.

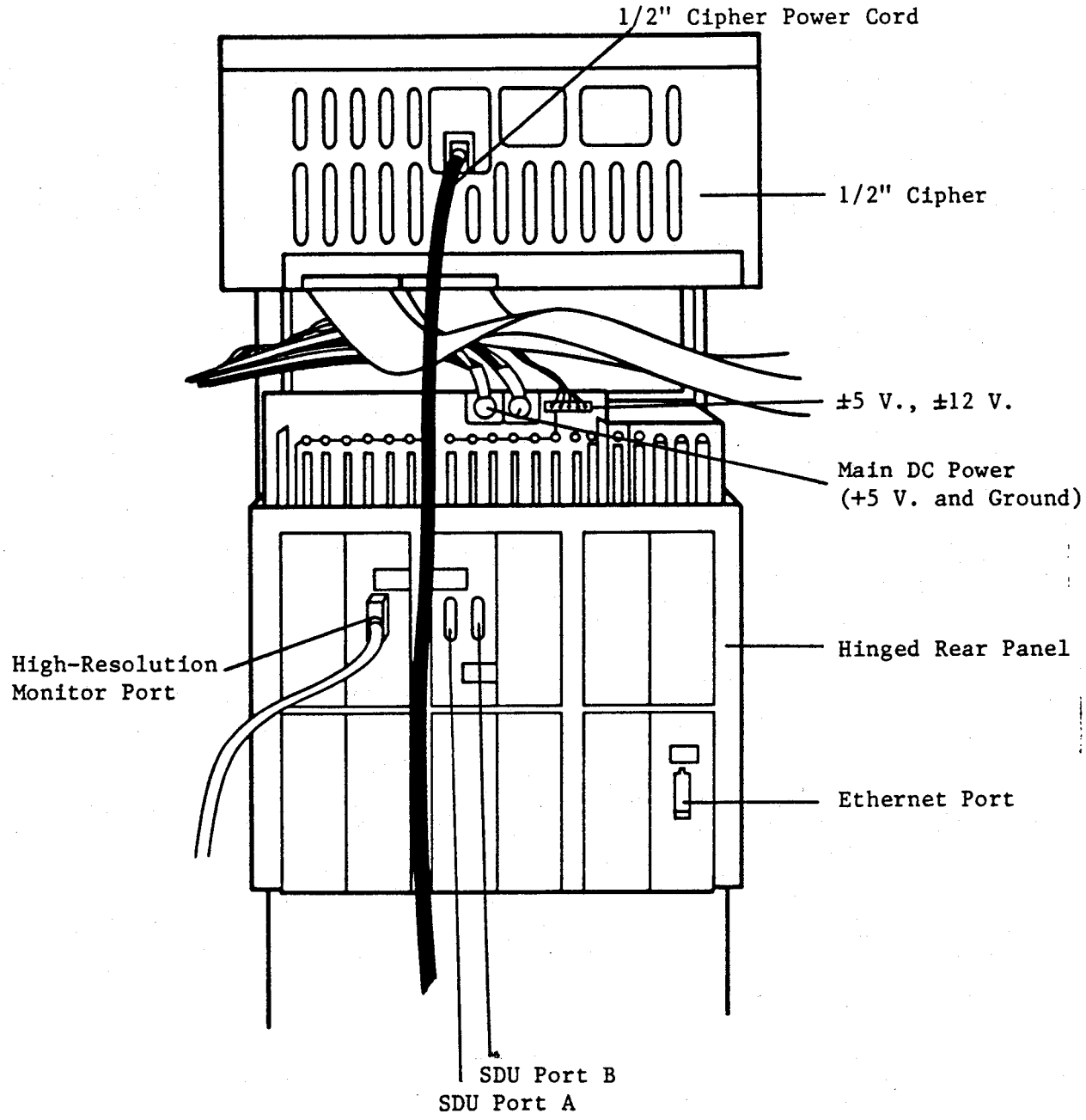
LAMBDA CARD CAGE

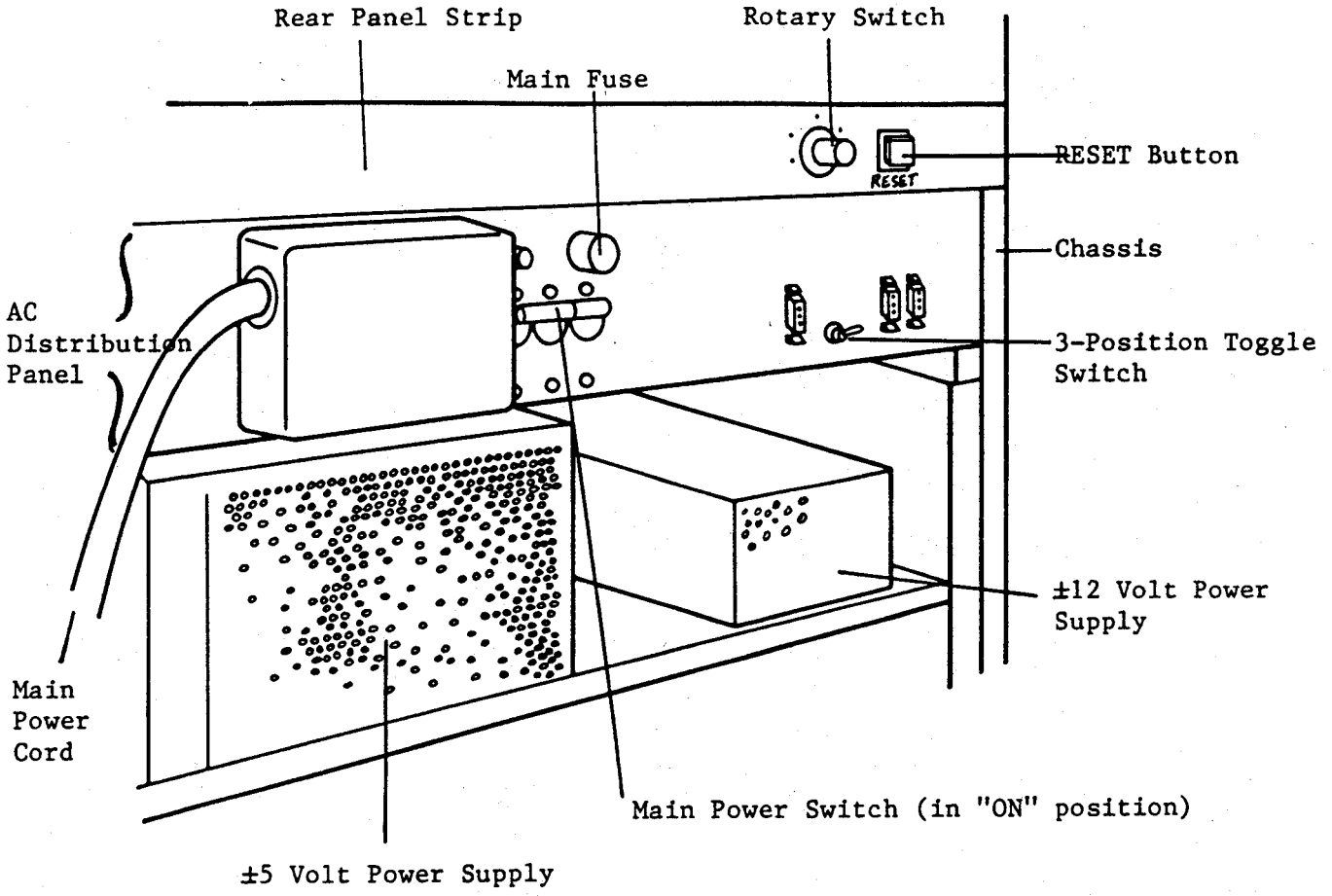




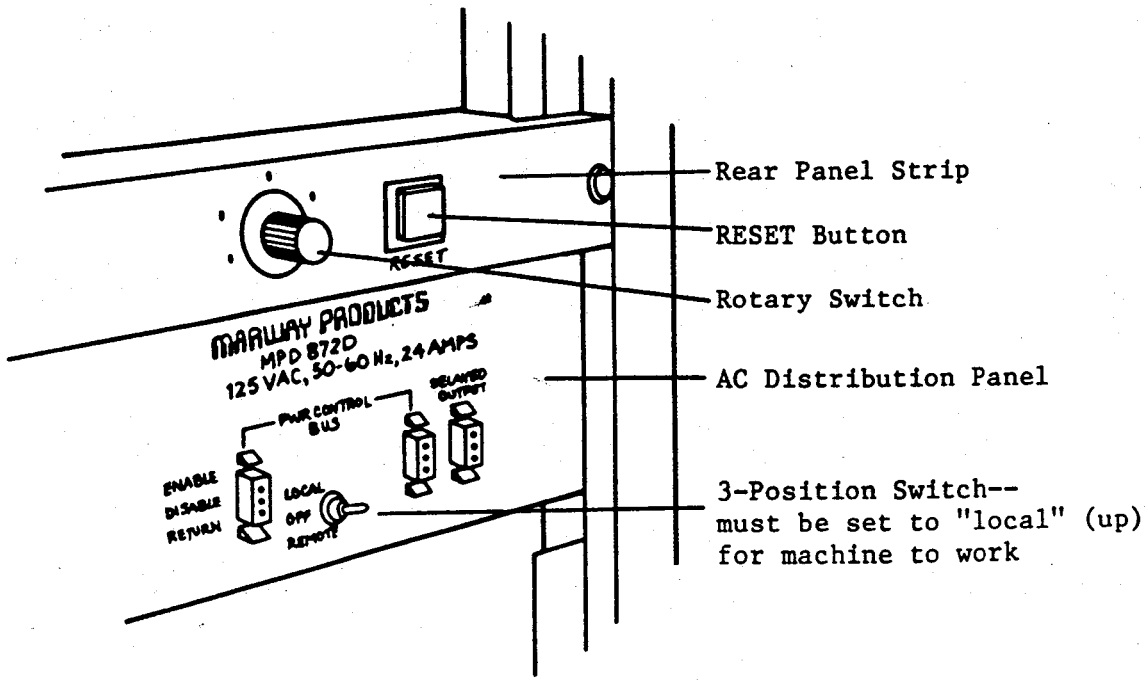
TI BACKPLANE (NUBUS AND MULTIBUS) AND LMI PRIVATE BACKPLANE

REAR VIEW OF LAMBDA WITH REAR PANEL REMOVED





REAR PANEL STRIP AND AC DISTRIBUTION PANEL--DETAIL



The VCMEM paddlecard goes in the bottom connector in slot 8, at the back of the machine. One cable runs from here to the high-resolution monitor cable connector on the back panel.

The color monitor (optional) output plug inserts in slot 13's bottom connector. From top to bottom, the three coax cables are Red, Green and Blue. These cables solder to female BNC connectors on the back panel, the center wire soldered into the center hollow prong of the BNC and the outer shielding connected to the solder lug on the perimeter. Looking at the back panel BNC connectors from the outside, Red is at the top left, Green is at the top right, and Blue is directly below Green.

5.5 Installing Monitors

To hook up an SDU console terminal, connect the top connector at the back of the Z29 to "SDU Port A", the male RS232 connector labeled on the back of the machine, using a standard, straight-through RS232 cable. Make sure the rotary switch on the back of the machine (just above the back of the power distribution panel, toward the right) is set to 1. The Z29 should be set up in Auxiliary mode at 9600 baud.

Connect the high-resolution monitor to the labeled "Hi Res" port to the left of SDU Port A.

Another terminal (normal resolution and not a console) can be hooked up to SDU Port B. If it is another Z29, it should be SETUP as a Normal mode port at 9600 baud. The cable to this second Z29 should plug from the bottom connector located on the back of the terminal to SDU Port B, the female RS232 connector next to SDU Port A. SDU Port B is the standard port for a printer.

5.6 The AC Distribution Panel

The AC distribution panel contains 10 Edison-style sockets.

The power cables from the card cage power supply, disk drive, and tape drive all plug into the AC distribution panel.

The three-position toggle switch on the right of the panel must be set to "Local". If it is set to "Off" or "Remote", it will cut power to all panel sockets except for the two sockets at the extreme right of the panel.

5.7 The Rear Panel

The rotary switch should be set to "1" to use SDU Port A as the 9600 baud console.

The LMI Lambda RESET button is the square button at the right of the panel.

5.8 Installing the Tape Drive

First, make sure the tape controller and the cables are firmly plugged in.

5.8.1 Installing the Cipher Tape Drive

On a 1/2" Cipher tape drive, one cable (part number 2338206) goes from P1 on the left rear of the tape drive to the bottom connector on the tape controller. The red line on the cable is on the bottom and the cable is flush against the top of the connector. The other cable, part number 2338207, goes from P2 on the left rear of the drive to the top connector on the tape controller. The red line is still on the bottom; the cable is flush against the bottom of the connector.

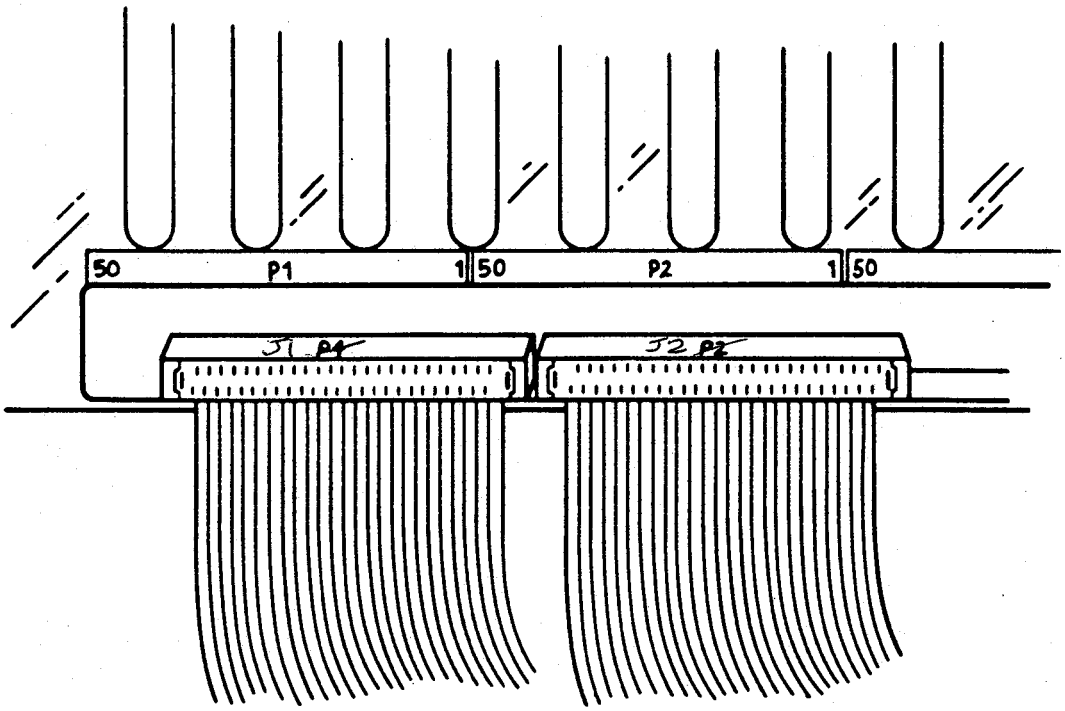
The Cipher Tape Drive is mounted in the main cabinet when the system is delivered. To install it, the following procedure must be followed:

1. Remove the tape that holds the top covers and front door in place. Open the top cover by lifting the sides directly behind the front panel.

2. Remove the retaining pin from the cover stay, lift the two black Ny-Latch fasteners, and raise the cover. A slot is provided to hold the stay.

3. Pull the tachometer away from the hub and remove and discard the foam cushion. Carefully replace the tachometer assembly on the takeup hub.

4. Inspect visually for damage, especially for dents or any parts that have obviously been broken off or bent. Examine the tape path, reel hub, tachometer, and other parts for



CLOSEUP OF BACK OF 1/2" CIPHER TAPE DRIVE (NOTE "P4" AND "P2" LABELS)

foreign matter.

5. The voltage is preset for standard US or optional European requirements according to the order. No voltage adjustment should be necessary. For reference, instructions on adjusting the voltage are provided in Appendix F.

6. The power cord supplied is for a 120V outlet. Plug it into the AC distribution panel.

7. If necessary, clean the tape path using 1,1,1 trichloroethane. For details on cleaning, see "Preventive Maintenance", Appendix A.

8. Power up the unit and verify that the UNLOAD indicator lights up. Allow for a two-second delay while front panel lights flash, indicating the performance of a self-diagnostic.

9. Open the front-panel door by pressing down gently on the center top of the door. Observe that both the top cover and the front-panel door lock automatically during all tape-loaded functions.

10. Insert the tape into the front panel of the unit with the write-enable ring side down.

11. Close the front-panel door.

12. Actuate the LOAD/REWIND switch, which locks the access doors. The LOAD/REWIND indicator blinks while operating and remains lit after the LOAD sequence is completed.

13. Depress the ON-LINE switch several times and verify that it is alternately lit and extinguished.

For further details on the operation of the tape drive, see the Cipher Tape Drive Manual.

5.8.2 Installing the Optional Kennedy Tape Drive

On the Kennedy reel-to-reel tape drive, the cable (#2338207) from the top connector on the tape controller goes to J1 at the top of the Kennedy's formatter board. The cable (#2338206) from the bottom connector on the tape controller goes to J5 at the bottom of the Kennedy's formatter board.

Make sure the DIP switches 1-3 on the right DIP switch box on the Kennedy formatter card are set to 75 ips:

speed	1	2	3
25 ips	1	1	0
75 ips	0	1	1

The switch on the front left of the drive must be set for 1600 bpi.

5.9 Inspecting the Card Cage

Inspect for damage, such as broken cards, loose cables, or any signs that brackets have been twisted or stressed.

5.10 The Optical Mouse

Your new LMI Lambda is equipped with an optical mouse manufactured by Mouse Systems Inc. Setting up the mouse for operation is a bit different than it was with the mechanical mice some of you may be familiar with from CADRs. The mouse box should contain a mouse with a MODULAR-JACK (RJ-11C) connector, a mouse pad, and a technical manual from Mouse Systems. The box may have empty spaces for other equipment; these are for accessories not needed for your LMI Lambda.

The metal object with blue and green lines printed on it is your MOUSE PAD. The mouse runs on it and will not work correctly unless it is on the pad. Put the pad on your desk in a comfortable position, vertically oriented like a sheet of paper.

The connector on the mouse should be inserted into the matching socket on the right side of the monitor. There is only one appropriate socket.

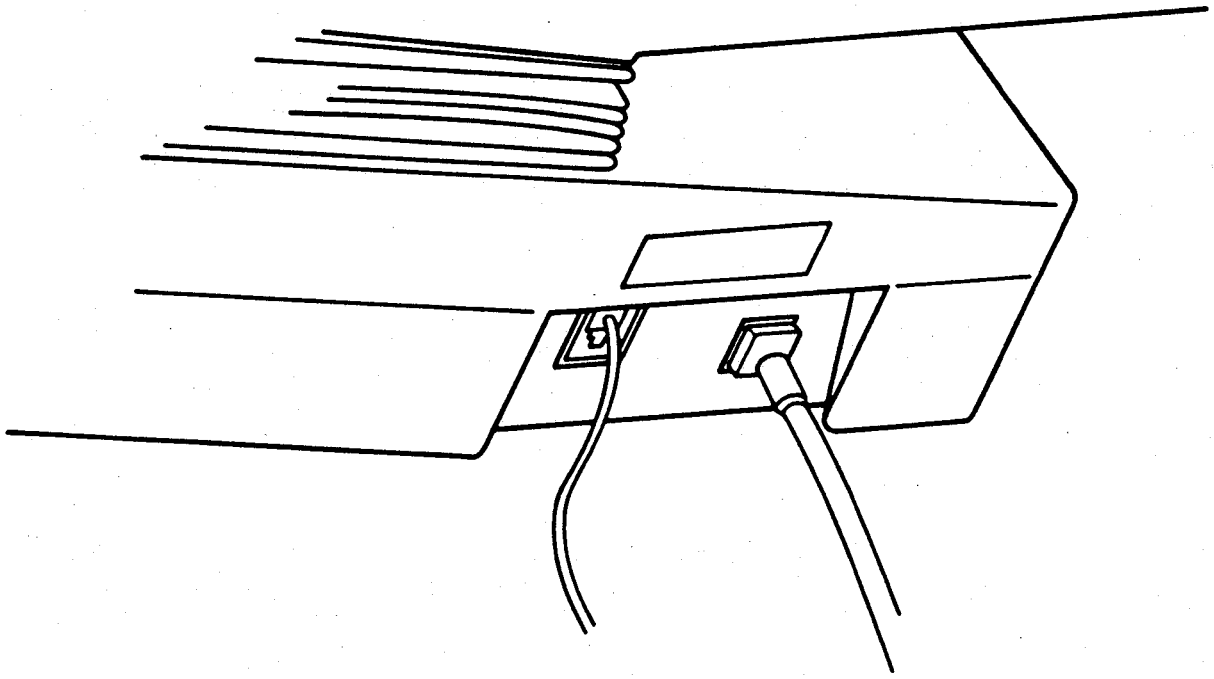
The mouse receives its power from the monitor and loses power each time the monitor is turned off, or when it is disconnected from the monitor.

If you have a problem with the mouse, first make sure the LMI Lambda software has not crashed. Then try power-cycling the mouse by pulling its connector out of the monitor for a few seconds and recalibrating. This will usually clear up any problems. If trouble persists, turn the mouse over and look at the LEDs that shine through holes there. Normally only one of the two LEDs will be on and will glow continuously. If no LEDs are lit, try pushing

any of the buttons on the front of the mouse. If both LEDs light the mouse is not calibrated; try the calibration process again. If no LEDs are lit the mouse probably is not receiving power. If one or both LEDs are flashing, there is some sort of error detected by the internal mouse processor. The Mouse Systems technical manual describes how you can correct most of these. If this doesn't succeed, contact LMI for a replacement mouse.

If you have a black mouse, it must be calibrated each time it is powered up. To calibrate the mouse, move it around in large circles at a comfortable speed until the mouse cursor starts moving, then click any mouse button once. Also on the black mouse, two of the DIP switches inside the mouse are changed from their standard settings. Switch 4 is turned off; this sets the pad orientation to suit our screen shape. Switch 6 is turned on. Beige mice contain no DIP switches.

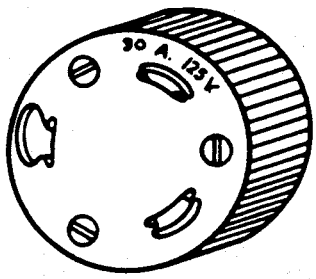
KEYBOARD AND MOUSE CONNECTIONS TO BACK OF MONITOR



Keyboard

Mouse

HUBBEL 30A 125V TWISTLOCK PLUG, #2611



Chapter 6

SYSTEM OPERATION

6.1 Powering Up

Before connecting power, make sure that all switches are off.

1. Insert the main power cable into a twistlock female connector (Hubbel 2613) on its own 30 Amp. circuit and twist it clockwise. If it does not catch, pull it out, turn it one-third and reinsert.

2. Make sure the rotary switch is set to position 1.

3. All cards and cables should already be installed. Make sure that the main circuit breaker on the machine is off, the circuit breaker on the right rear of the disk is on, the disk drive, power supplies and any fan and tape plugs are plugged in, and the Start switch on the disk front panel is off.

4. An additional terminal currently is necessary to boot the LMI Lambda. Almost any RS232 full duplex terminal that can be used as a DCE device can be used. LMI supplies a Zenith Z29 terminal for this purpose. The Z29 terminal should be SETUP as a 9600 baud, Auxiliary mode port and plugged into SDU port A, the male RS232 port on the back of the Lambda.

5. Verify that the orange indicator light on the AC power distribution panel is on. Also, make sure the 3-position toggle switch on the right center of the power distribution panel is up (in the "Local" position). Important: If this switch is turned to "Off" or "Remote", it will turn off power to the LMI Lambda.

6. Turn on the disk drive circuit breaker. It is located on the back right of the disk drive.

7. Turn on the main rocker switch to power the card cage. "On" is the up position.

8. Check to make sure the fans are operating and that there are lights on the front panel and on some of the cards.

9. As a precautionary measure, check for any crackling sound or smell of smoke. In case of either, turn off the machine IMMEDIATELY by turning off the main circuit breaker or pulling out the plug; then call LMI.

10. Turn on the tape drive using the switch on the right front of the drive.

Within a few seconds after you power up the LMI Lambda, the message "Monitor Version 7" will appear on the SDU console screen, followed by a SDU prompt (>>). The power light on the disk front panel will be on. (If not, make sure that the disk is plugged in and its circuit breaker is on.) Turn on the Start switch on the disk front panel. The START light will go on. In forty seconds the READY light on the disk front panel will go on. Make sure the Protect (write protect) switch on the disk front panel is off.

Once all three lights--POWER, START, and READY--are on, the disk is "spun up" and ready for use.

If the red SETUP light on the LMI Lambda front panel comes on and stays on, you need to set up the CMOS RAM. See the section on "SDU Operations" in chapter 7, "Troubleshooting by Area", for instructions.

Anytime you power up the machine or press the square RESET button on the back of the machine, you will have to type the following command on the console:

```
init
```

Whenever this command is running, the lights on the front panel will go on in the following sequence:

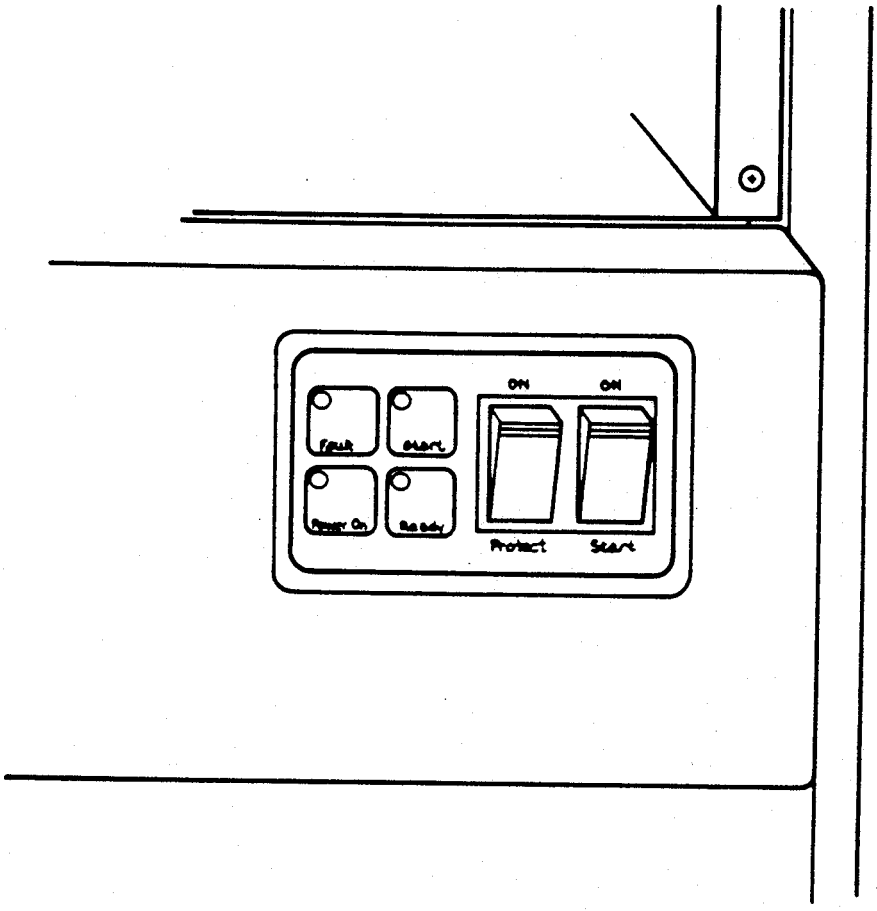
1. Initially the red ATTN and SETUP lights will go on.

2. The ATTN light will go off.

3. If the CMOS RAM on the SDU is valid, the SETUP light will next go off.

4. The green RUN light will come on, indicating that the system is ready for use.

The "init" command enables the NuBus and Multibus. If you forget to type "init", the next command you type will probably return a "NuBus and Multibus timeout" error message. If that happens, just type "init" and start again.



FUJITSU EAGLE DISK DRIVE FRONT PANEL

It is also necessary to type "init" after various errors. If you get an error message from the SDU or if you return to the SDU by typing CTRL-C to exit a program, it is a good idea to type "init".

6.1.1 Configuring the System

The SDU program "config" locates boards, allocates memory and sets the speed of the LISP processor boards. It uses information in the "/disk/monitor/conf/std" file to determine the setup of the UNIX disk partitions, I/O ports, log port and log channel. The system will need to be reconfigured only if you need to change the number of memory or processor boards, change the LISP board speed, add another board to your system, or make another such change. This is not something that needs to be done every time you power up the system.

The program works whether the machine contains one or more processors. It puts information in the file "/disk/lambda/sh-config.1" which is accessible by both the UNIX and LISP processors. In most cases, the default configuration will be sufficient for your needs. In special purpose machines (for example, if you are developing support for new Multibus boards), you may have to configure some options yourself.

To configure your system, type at the SDU

```
config
```

The program will prompt "Create default configuration? (y/n)". The configuration defaults to a speed of 3:3, which is sufficient for at least the first boot after installation. Type a "y" followed by a RETURN and config will configure the system. It prints how much memory is available and how much it is allocating to LISP and to UNIX. Next, it asks "Do you want to change anything? (y/n)". Answer "n" followed by a RETURN and the configuration will complete.

If you want to change the LISP processor speed, you can type "y" followed by a RETURN when asked if you want to change anything. You then get a menu of possible commands, and a "cmd:" prompt. In this case, type

```
lambda
```

You will be prompted as to whether you want to change the Lambda's speed. Answer "y" followed by a RETURN and you will be prompted to supply the source and execute cycle speeds. PC board sets will work at full speed (1:1). Wirewrap boards should be set to 2:1, 2:2, or 3:3 according to their performance capability. After selecting the speed, you will again see a "cmd:" prompt. Write out the configuration and exit by typing

write

If you begin the config program by saying that you do not want the default configuration or that you want to change something, online documentation in the config program will tell you what to do. If you have any special confiruration questions, call LMI.

Note: The SDU "config" program replaces the "memsetup" and "setspeed" programs as well as the editing of the file "/disk/monitor/conf/std".

6.1.2 Running a STREAMS machine on 4 memory boards

If a STREAMS machine has only 4 memory boards, "config" must be told to divide them equally. The config program defaults to using 3 boards for LISP and 1 for UNIX, but UNIX really needs 2 (and LISP can run with 2, though slowly). This configuration needs to be done only when the board configuration changes.

Type at the SDU

config

When asked "Create default configuration? (y/n)" answer "y". It will print the configuration, giving 1.5Mbytes of memory to LISP and 512K to UNIX. It will say "Do you want to change anything? (y/n)" and you should respond "y" (ordinarily, "n" is the proper response here).

It will give you a list of possible options, and will issue a "cmd:" prompt. Type

mem

at this prompt to edit the memory configuration. It will ask for the amount of memmory to give the Lambda processor in slot 0, and you should type

1m

It will respond that 1024K (1M) is left for the 68000 processor, and will give another "cmd:" prompt. Since everhthing is now set up properly, type

write

at this prompt to write out the configuration and exit the program. From now on (until you run "config" again), 1MB will go to each of the two processors.

It is also necessary to type "init" after various errors. If you get an error message from the SDU or if you return to the SDU by

typing CTRL-C to exit a program, it is a good idea to type "init".

6.2 Booting LISP

Note: The LISP boot procedure and related functions documented here are still being improved, and therefore are subject to change. These instructions are current for "LMI Release 1". Check with LMI if you have questions.

Type on the console

```
lboot
```

This boot program will run for under a minute, ending with "Running bootstrap microcode...done". At this time you should see "run bars" flashing on the high-resolution monitor screen. These bars show that the LISP processor is cold-booting. They appear about four inches above the lower right corner of the screen.

NOTE: While LISP is booting, do not type on the keyboard, move the mouse, or detach the mouse or keyboard. Neither type on nor detach the console keyboard. If you do, the machine will probably crash.

After a few minutes of run bar motion the screen should be cleared, and soon afterwards the LMI Lambda will finish booting. Again, make sure that there is no keyboard or mouse input from anywhere while the machine is booting.

After LISP has finished booting, the word "keyboard" will be displayed at the bottom center of the high-resolution monitor screen and it is safe to type on the keyboard. It is also safe to detach the console from SDU port A, if one is connected there. This terminal will not be necessary again until the next time you are out of LISP.

If you will be using tapes on the LISP machine, you must first initialize the tape controller. Type in LISP

```
(fs:tm-init)
```

to do so.

Since this needs to be done only once each time LISP is booted, it is a good idea to do it just after booting LISP so it will already have been done by the time someone needs to use tape.

Note: Nothing will be harmed if the tape controller is initialized more than once; it is just unnecessary.

As a precautionary measure: Every two weeks, and whenever LISP has been halted with files open, type to the LISP Listener after LISP is booted

```
(fs:lm-salvage)
```

This will take ten minutes or less, and will make sure the LISP file system is consistent.

6.2.1 Possible Problems While Booting

The first time you try to boot the machine after power-up, the memories might not be initialized. The machine might stop booting and print "NuBus timeout" on the SDU monitor. If this happens, type "init" and "lboot" again. If the problem recurs three or four times (for instance, if the machine consistently stops at the same point, or at random times during the boot procedure), call LMI.

If LISP crashes during booting, you will be able to tell by the lack of activity on the screen. The run bars will stop moving and not start again. A crash after LISP is already up will cause the cursor to stop flashing and the time, in the lower left-hand corner, to stop updating. Call LMI if LISP continually crashes.

If booting stops repeatedly with a "timeout" (Nubus or Multibus) or if the message "no Lambda processor" appears even though there is a LISP processor in the machine, there is most likely something wrong with the board placement file. Run the SDU program "config" to reconfigure the board placement. Try again, and call LMI if booting continues to fail; a memory board may have failed.

6.3 Booting UNIX

Type on the console

```
init

uboot
```

UNIX should boot. It will print a line or two of text, then will prompt with a number sign (#). Next type CTRL-D to boot UNIX the rest of the way. (Now it will be able to connect to other terminals besides the console.)

The first thing UNIX will do after the CTRL-D is to run the program "/etc/fsck" to make sure the file system is consistent. It will then display the date and put a login prompt on all terminals that are connected and enabled in the "/etc/ttys" file.

(For an explanation of "/etc/fsck", see CRASHES IN UNIX.)

If nothing happens for about a minute after you type "uboot", type CTRL-C. The message "Command exited with value 1" should appear on the console, followed by an SDU prompt. Type

```
init

uboot
```

to try it again.

If you get UNIX booted most of the way--if it prints a line or two, but it won't print the prompt #--wait about 30 seconds, then type CTRL-C and try again, using "init" and "uboot".

If, while trying to boot UNIX, you get one of the messages, "Command exited with value 647," "NuBus timeout," "Multibus timeout," or "Command got both NuBus and Multibus timeout," type "init" on the console and try again. (Remember which error you got.) If you get the same error again, call LMI; one of the cards may be bad. (Most likely the CPU or memory board.)

In Release 1, "uboot" sometimes causes a screenful of obscure error messages to print, with UNIX failing to boot. If this happens, type "init" then "uboot" again. If the problem persists, boot with "superboot" instead. LMI regrets this inconvenience.

6.4 Booting LISP and UNIX concurrently (STREAMS)

Type on the console

```
init
```

and then

```
superboot
```

LISP will boot, followed by UNIX. When UNIX boots, bring it up multiuser by typing CTRL-C on the console.

Note: it is not necessary to type

(fs:tm-init)

to initialize the tape controller if LISP has been superbooted.

6.5 Setting Terminal Type in UNIX

To tell UNIX what type of terminal you have, type while in multiuser UNIX, using the "C shell" (see CREATING NEW UNIX DIRECTORIES):

setenv TERM h19

on a Z29, or

setenv TERM rsd

on a high-resolution monitor.

Then, on either terminal, type

stty newcrt

Important: If you type the "setenv" command (or "stty") in singleuser instead of multiuser UNIX, the system will crash.

6.6 Switching from LISP to UNIX and from UNIX to LISP

The STREAMS software will now allow you to switch easily from the ZetaLISP-Plus environment to the UNIX environment, and even use LISP from UNIX and call UNIX tools from LISP.

The LISP Processor and the MC68000 Processor running UNIX can run concurrently, sharing various system resources. Both processors may access the disk through a common interface. Memory on the NuBus gets divided among the host processors based on certain system configuration information.

There are two ways to use the UNIX-STREAMS software: directly using the three dedicated stream links or through a virtual CHAOSNET connection.

6.6.1 Direct Connection

The easiest way to use a dedicated link is by typing <SYSTEM> U to get a UNIX window. This behaves just like any other Lambda window (such as a ZMACS or LISP window) except that it contains a UNIX process. As such it does not interact with other processes.

There are three direct stream links between the LMI Lambda and the UNIX system. From UNIX these are called:

```
/dev/tty10
/dev/tty11
/dev/tty12
```

These are character special devices that act just like terminal lines. By default, they are in raw, no echo mode, but you can modify this. The UNIX initializer only accepts logins on /dev/tty10, so this is the line that the routine called by SYSTEM U uses. /dev/tty11 and /dev/tty12 are not enabled in the UNIX file /etc/ttys, so the initializer doesn't set up login listeners listeners for them. If you want to use them with login capability you must change the initial 0s (zeroes) to 1s (ones) for the appropriate entries in /etc/ttys.

From LISP, the three relevant symbols are:

```
*unix-port-0*
*unix-port-1*
*unix-port-2*
```

These are bound to objects to the type si:unix-stream. You must send the :setup message to these streams before use, and after a warm boot. I/O operations to these streams cause data to be sent or received from the corresponding UNIX devices.

6.6.2 CHAOSNET Link

The CHAOSNET link provides a general communication mechanism allowing each host processor to behave like any other CHAOSNET host including the use of file transfer (CFTP) and remote terminal (SUPDUP) protocols. (Use of CHAOSNET on the LISP Machine is described in Chapter 23 of the "LISP Machine Manual".) The following assumes a familiarity with the use of CHAOSNET with both UNIX systems and LISP Machines.

You can use the new STREAMS software even if your Lambda doesn't have an associated UNIX processor. Just specify the host address of a UNIX host on your local area network. That the associated UNIX host is not a part of the local system is immaterial except that the overhead required to use the physical network may degrade

performance slightly.

Both the UNIX system and the LISP processor have CHAOSNET servers which respond to the contact name "EVAL". The UNIX system's EVAL server passes input from the connection to the UNIX shell and the standard input of any commands and passes output back through the connection. The Lambda's EVAL server reads LISP forms from the connection, evaluates them and prints the results back. Using these servers either system can take advantage of the resources and software of the other.

6.6.3 A Typical Application

The following code provides an example of how you might use the STREAMS LISP/UNIX connection. It uses CHAOSNET.

The problem: how to write an extension to ZMACS that would allow you to do complicated regular expression substitution. UNIX already has such a facility called "sed". Using STREAMS you could pass the substitution command and the text from the buffer to the UNIX host; the UNIX tool sed would perform the work and shoot the results back to ZMACS. This is much easier than rewriting the sed facility in LISP.

6.7 Logging In

To log into LISP, type the LISP function

```
(login "name")
```

where "name" is your username. You will then be in a LISP Listener, from which you can access any part of the system.

To log in to UNIX, type your username at the prompt "login:". UNIX will prompt you with "password:". Type your password at the system. With the UNIX set up as shipped from LMI, you can log in as "root", with no password, if you don't have a login ID for UNIX.

6.8 Creating New LISP User Directories

Make a directory for a user ("naha," for example) by typing in LISP

```
(fs:create-directory "naha;")
```

Once the directory is created the user can use ZMACS to create an init file (called "lispm.init") in that directory.

6.9 Creating New UNIX User Directories

"/etc/passwd" describes users to the system. Each user is described by a line consisting of seven fields:

- A login name
- An encrypted password
- A user ID
- A group ID
- A field containing information for accounting purposes
- The full pathname of the user's login directory
- A program name

If the last is null, /bin/sh is invoked; those who prefer the "C shell" should use /bin/csh. (You must use /bin/csh for terminal type setting to work.)

Login as root, then edit the file /etc/passwd to add an entry for the user being added. A typical entry for a privileged user would be

```
gjc:j9qRm:117:100:George J. Carrette:/usr/gjc:/bin/csh
```

Next, make the directory by typing

```
mkdir /usr/gjc
```

and change its owner from Root to the user by

```
chown gjc /usr/gjc
```

If a password is desired, set it with the "passwd" program. This can be done by the user himself or by the superuser:

```
passwd gjc
```

The user will be prompted for the old password and then twice for the new password. The password is stored in "/etc/passwd" in an

encrypted form.

For more information, consult the UNIX Programmer's Manual, volume I.

6.10 Installing LISP Sources

Use the LMI Lambda distribution tape. Type to LISP (if necessary)

```
(fs:tm-init)
```

Put in the desired tape and type

```
(fs:restore-magtape ':query nil)
```

This will take a bit over an hour. When it is done, the bottom of the screen will say "keyboard", and you can rewind the tape by typing

```
(fs:mt-rewind)
```

or by "manually" rewinding it from the tape drive's front panel.

6.11 Backing up LISP Directories

To backup a LISP directory onto tape (e.g., the LISP directory for the user "mwt"), mount a blank tape (with a write ring) and type while in LISP

```
(fs:copy-directory "mwt;" "mt:" ':query nil)
```

This should be done, as often as makes you comfortable, for each directory that has been modified. More than one directory can be vput on the same tape, merely by repeating the above command, substituting the next directory name in place of "mwt".

When you finish writing directories onto the tape, type

```
(fs:mt-write-eof)
```

to write an end-of-file mark onto the tape. (If you forget this, the tape will work fine except that you'll get an error at the end, when the machine doesn't see an eof mark.) Take out the write ring and put the tape somewhere safe. When you want to restore it, you can put it in the drive and make sure it is at the

beginning of the tape (i.e., the LOAD light is on). Then you can type

```
(fs:restore-magtape ':query nil)
```

and then

```
(fs:mt-rewind)
```

6.12 Backing up UNIX Files

There are two ways to backup UNIX files. One is via the "tar" command, which is often used for backing up a particular file or directory and can be restored also using the "tar" command. The other is via the "dump" command. Files on dump tapes can be restored using the "restor" command.

6.12.1 Backing up UNIX Directories

To backup a UNIX directory onto tape (e.g., the UNIX directory for the user "mwt"), mount a blank tape (with a write ring) and type while in multiuser UNIX

```
cd /usr
```

then

```
rmt4 10, it says

tar cvfb /dev/rmt4 10 mwt
```

Take out the write ring and put the tape somewhere safe. You cannot append files to the end of the tape--if you try, the files at the beginning of the tape will be written over, and the only accessible files will be the new ones you just wrote. To put more than one directory onto the tape, issue the directory names (e.g., "mwt" and "zrm") in the same command line

```
tar cvfb /dev/rmt0 10 mwt zrm
```

If you need to continue onto another line with directory names, either set your terminal for "line wrap" and keep typing, or type a backslash (\) before typing the carriage return and then continue the command line:

```
tar cvfb /dev/rmt 10 mwt zrm\
jqr
```

(A backslash turns off the special meaning of whatever character follows it - in this case, the carriage return.)

NOTE: Keep in mind that since UNIX doesn't have version numbers, any files extracted from tape will write over files with the same names, already on the disk. To prevent this, you should rename files on the disk before extracting the tape. For example, if "foo" and "bar" are the two files in the directory "mwt", you should type in multiuser UNIX

```
cd /usr/mwt

cp foo foo.old

cp bar bar.old
```

It is now safe to extract the tape, since "foo" has been copied into "foo.old" and "bar" to "bar.old". Type

```
cd /usr

tar xpvf /dev/rmt0
```

You can rename disk files at any time with the "mv" or "cp" command if you wish. The "cp" command preserves all the links to the file, and so is probably more useful than "mv". (See the UNIX Programmer's Manual, Vol. 1, for more information.)

6.12.2 Full and Incremental Dumps: Backing up UNIX Filesystems

"dump" backs up a filesystem at a time, in a more compact format than "tar" uses. It can be used both for full dumps and for incremental dumps. Refer to the Nu Machine UNIX Programmer's Manual, Vol. 1 for documentation on the "dump" command. Below are some examples of "dump" command use, but you should refer to the UNIX Programmer's Manual, vol. 1, to learn about the command before using it.

To see what filesystems are mounted on what disk devices, type in unix the command

```
cat /etc/rc
```

This will print the "rc" file on the screen. The "rc" file is a shell script executed when you bring up mutiuser UNIX. While running it mounts ("/etc/mount") each filesystem on its disk device. "Root" is already mounted on /dev/dk0a as soon as UNIX is booted. The usr filesystem is mounted on /dev/dk0e by convention.

To minimize the possibility of errors, dump filesystems while no one is using the system. In this case, it is safe to backup the

"raw" devices.

Note: If you backup the system while it is in use, be sure to use the non-raw devices (/dev/dk0e, in the above case). This will take longer, but will decrease the risk of inconsistencies.

6.12.2.1 Example: The Full Dump

To backup the usr filesystem, mount a 2400 foot long tape with a write ring and type in UNIX

```
dump 0uf /dev/rmt0 /dev/dk0e
```

The message "dumping /dev/dk0e to /dev/rmt0 will appear, with more messages as the dump continues. You will be notified when either another tape is needed or the dump is complete.

6.12.2.2 Example: The Incremental Dump

To backup all the files in the usr filesystem that have changed since the last dump, do a "level 9" dump instead of the "level 0" dump done in the example above. To do do, type similarly to above,

```
dump 9uf /dev/rmt0 /dev/rdk0e
```

Tapes smaller than 2400 feet can be used as long as you can specify the length a an option in the command line. Consult the UNIX manual for this.

6.12.2.3 Restoring a Dump Tape

To restore all or part of a dump tape, use the "restor" command. The procedure for using "restor" is documented extensively in the UNIX Programmer's Manual, vol. 1. Read the UNIX manual and make sure you thoroughly understand the "restor" command before using it. AS with "dump", the "f" option should be used and the first argument should be /dev/rmt0, to specify the tape drive to be used.

6.13 Making a LISP Microcode and Band Tape

LMI's standard format for band tapes is two microcode bands followed by one LISP band. Load a blank tape (with a write ring) and type, assuming "lmcl" is a good microcode band,

```
(si:copy-disk-partition 0 "lmc1" "mt" "")
```

Note that here there is no colon after the letters "mt"; it is not "mt:".

Repeat this, maybe using a different microcode band (e.g., lmc2 in place of lmc1). Next, do it again but with a LISP band (e.g., lod1) instead of a microcode band. When all this is done, write a few extra eof's on the tape by typing

```
(fs:mt-write-eof)
```

three or four times. As always, take out the write ring and put the tape somewhere safe.

6.14 Making a Root Image Tape (at the SDU)

A "Root Image" contains all the information necessary to boot either LISP or UNIX. If you want a copy of it on tape, type at the SDU

```
copy $disk tape 8500
```

IMPORTANT: Be very careful not to confuse the order of "$disk" and "tape". If you do, random garbage will be written all over what used to be the root image on the disk.

6.15 Restoring a Root Image Tape (at the SDU)

To restore the root image onto the disk, type at the SDU

```
copy tape $disk
```

Notice that no length--e.g. "8500"--is necessary.

Restoring the root image is a destructive procedure. It replaces the current root image on the disk.

6.16 Logging Out of LISP

In the LISP Listener, use the function

```
(logout)
```

If another user logs in, you will automatically be logged out.

6.17 Logging Out of UNIX

You can log out of UNIX by typing "logout" (when you are in multiuser UNIX) or typing CTRL-D at the "%" prompt.

6.18 Leaving LISP Gracefully

To leave the LISP environment, first save any work you have done and logout using the (logout) function. Make sure no one else is reading or writing files over the network from another machine. You can tell that someone is doing so if there is a file name in the bottom right corner of the screen. Warn other networked users that you are about to halt the machine and wait for them to finish using any files they have open. If you cannot find the person who has the file open (or if the other machine "forgot" and left the file open), type in the LISP Listener

```
(chaos:reset)
```

and then

```
(fs:close-all-files)
```

and finally halt the machine by

```
(si:%halt)
```

This will cause the LISP processor to halt. If you are using STREAMS, leave UNIX gracefully (next page) before returning to the SDU. Once this is done, press the SDU reset button on the back of the machine. Type at the SDU

```
init
```

before trying any other operations.

6.19 Leaving UNIX Gracefully

If you are in single-user UNIX, you can leave by merely typing

```
sync

sync
```

and then pressing the RESET button. After you get a prompt from the SDU, type

```
init
```

If you are in multiuser UNIX, make sure the other users have saved their work and know you are about to stop UNIX. Login as "root" or become a superuser by typing

```
su
```

You will get a different prompt (#) than you had as a multiuser. Type

```
sync

sync

kill -15 1
```

You will immediately get a prompt. Wait about 20 seconds. When you get another prompt, it means you are in single-user UNIX. Type

```
sync

sync
```

If you are using STREAMS, leave LISP gracefully (previous page) before returning to the SDU. When this is done, press the RESET button. When you get a SDU prompt, type

```
init
```

6.20 RESET Button

Whenever you want to leave UNIX or LISP when the machine "hangs" and you are unable to regain control, you must use the RESET button. It is the square black button on the right rear of the machine, mounted on the panel above the AC distribution panel. Pressing RESET momentarily grounds the machine's reset line, causing the SDU to regain control. Since this also resets the NuBus and Multibus, you must always type "init" to the SDU after you reset the machine.

This is not a preferred method for exiting LISP or UNIX, since it will forcibly halt processes in mid-operation. Whenever possible, you should leave LISP or UNIX gracefully. Still, there are times when no other option is feasible. Call LMI if you are concerned.

6.21 Restoring Disk from Tape

IMPORTANT: This is a destructive procedure, to be used only when absolutely necessary (on LMI's instructions). In most situations you will want to us only the "Installing LISP Microcode and Band" instructions.

If the disk must be completely reloaded, you must borrow a 68000 CPU from another machine or from LMI. This is because the LISP microcode and band can be loaded only in LISP (which isn't on a new disk) or in UNIX.

6.21.1 Systems with 474MB Fujitsu Eagle Disk And 1/2" Tape Drive:

A 68000 CPU is necessary to completely redo the disk software, as are at least 3 memory boards. Call LMI to borrow a CPU and memory for the duration of this operation.

If you need additional boards (CPU and memory), power down the system and install them, then power back up. Setup the SDU's CMOS RAM so that it can work with the Streams software you are about to load. Mount the "LMI Diagnostic Tape" and type at the SDU

```
init

/tar/setup clear eagle sp shell
```

```
init
```

Mount the "Streams Release 1 root image" tape. To install it, type

```
copy tape $disk
```

This will take about 20 minutes.

6.21.2 Configuration

A configuration program has been added to the system. This program, run from the SDU, locates boards, allocates memory and sets the speed of the LISP processor boards. It uses information in the "/disk/monitor/conf/std" file to determine the setup of the UNIX disk partitions, I/O ports, log port and log channel. The system will need to be reconfigured only if you need to change the number of memory or processor boards, change the LISP board speed, add another board to your system, or make another such change. THIS IS NOT SOMETHING THAT NEEDS TO BE DONE EVERY TIME YOU POWER UP THE SYSTEM.

The program works whether the machine contains one or more processors. It puts information in the file "/disk/lambda/sh-config.1" which is accessible by both the UNIX and LISP processors. In most cases, the default configuration will be sufficient for your needs. In special purpose machines (for example, if you are developing support for new Multibus boards), you may have to configure some options yourself.

To configure your system, type at the SDU

```
config
```

The program will prompt "Create default configuration? (y/n)". The configuration defaults to a speed of 3:3, which is sufficient for at least the first boot after installation. Type a "y" followed by a RETURN and config will configure the system. It prints how much memory is available and how much it is allocating to LISP and to UNIX. Next, it asks "Do you want to change anything? (y/n)". Answer "n" followed by a RETURN and the configuration will complete.

If you want to change the LISP processor speed, you could type "y" followed by a RETURN when asked if you want to change anything. You will then get a menu of possible commands, and a "cmd:" prompt. In this case, type

```
lambda
```

You will be prompted as to whether you want to change the Lambda's speed. Answer "y" followed by a RETURN and you will be prompted

to supply the source and execute cycle speeds. PC board sets will work at full speed (1:1). Wirewrap boards should be set to 2:1, 2:2 or 3:3 according to their performance capability. After selecting the speed, you will again see a "cmd:" prompt. Write out the configuration and exit by typing

```
write
```

If you begin the config program by saying that you do not want the default configuration or that you want to change something, online documentation in the config programM will tell you what to do. If you have special configuration questions, call LMI.

After running "config", type

```
init
```

Boot UNIX by typing

```
superboot
```

The "superboot" program boots first LISP and then UNIX. Even if LISP does not boot for some reason, UNIX boots after LISP fails. In this case, "superboot" will be unable to boot LISP because the microcode and band are not loaded, but will happily boot UNIX anyway.

Important: the following step should be done only once, when the system is first installed. It is a destructive procedure that wipes out the data in the existing UNIX /usr filesystem. Check the consistency of the "/"and "/sdu" filesystems by typing

```
/etc/fsck /dev/dk0a
```

then

```
/etc/fsck /dev/dk0g
```

Make the UNIX usr filesystem by typing

```
csh makedisk eagle
```

It will say

```
Making filesystems...
        isize=25600
        m/n=4500
```

It will then try to link disk-dependent files, but in the Eagle's case the files already exist so the message "link already exists" will be printed for each of the 4 files it tries to link. Next, it will check the filesystems by running the file consistency checking program "/etc/fsck".

Last, it will run the program "/etc/sethostname" to determine the network name of the UNIX system. A screenful of possible names will print on the screen. A screenful of possible names will be prompted to choose one. The convention is to name the UNIX system similarly to its LISP counterpart. For example, the first Lambda LISP system at a site should be called "LAMA", and the associated UNIX system should be designated "UNIX-a". The "sethostname" program does not check for valid names; therefore, be careful to type in a valid name when prompted.

When this is done, bring up multiuser UNIX as always by typing CTRL-D. The file system consistency check (fsck) program will run again, since it always runs before booting multiuser. At the prompt, log in as "root".

Mount the microcode and band tape and type

```
/lambda/load68
```

It will print a "Disk loader>" prompt and await your command. Type

```
printlabel
```

The program will prompt you for the type of disk you are using. As it suggests, you should respond

```
1
```

because you are using an Eagle disk drive. It will print out the disk label.

Note: even if you know there is no disk label because this is a newly installed system, you must type "printlabel" before initializing the label. Otherwise, the "minilabel" (which is on the root image and holds the position of the LISP disk label) will not be read in, and the program will give an error.

If there is no disk label because this is a new system, or if you do not care about any bands or files on the disk, initialize the label by typing

```
initlabel
```

When asked if you really want to initialize the label and instead use the existing one, you will see microcode and band names in the comment fields. If you already loaded the new microcode and band, you will see them on disk partition comment fields. If you did not already load them, note the location of available microcode and band partitions and type

```
load
```

The program will prompt you for the type of tape drive you are using: As it suggests, type

```
1
```

because you are using a Cipher 1/2" tape drive. The program will then prompt for which microcode and band partitions you want loaded. The microcode loads in about a minute; the band takes close to an hour. Refer to the LMI Lambda Field Service Manual for detailed instructions on how to install the microcode and band. When this is done, get out of UNIX, as always, by typing "sync" twice and pressing the RESET button. Then, type "init" as usual.

Now, boot UNIX and LISP concurrently by typing

```
superboot
```

This is now the command you will usually use when booting. Type CTRL-D to bring up multiuserr UNIX, as usual.

If UNIX has been ordered with your system, install the UNIX usr files. Mount the "LMI UNIX usr files" tape and type in UNIX

```
cd/usr
```

to change into the /usr directory, and then

```
tar xvpf /dev/rmt0
```

to extract the usr files from the tape. This will take almost an hour.

Next time you want to boot, merely type

```
init

superboot
```

and both LISP and UNIX will boot. If this is a new system with no LISP files you want, make the LISP filesystem by typing

```
(fs:initialize-file-system)
```

Load the LISP sources tape as usual. Next, you must "reconfigure for customer site". This was formerly done with a separate tape; now, however, the appropriate program is included with the sources and so is already on the disk. Follow the standard customer site reconfiguration instructions ("Reconfiguring for Customer Site").

6.21.3 Systems With 169MB Fujitsu Micro Disk And 1/2" Tape Drive:

Setup the CMOS RAM by typing

```
/tar/setup clear micro169 sp shell
init
```

After copying the root image, boot UNIX by typing

```
uboot -C /disk/monitor/conf/std.micro169 tiunix
```

When it boots singleuser, type

```
/etc/mount /dev/dk0g /sdu
cd /sdu/monitor/conf
rm std
sync
sync
cd /etc
/etc/umount /dev/dk0g
rm rc fstab checklist
sync
sync
cd /
csh makedisk micro169
```

This mounts the sdu monitor filesystem so that you can remove the configurationfile linked to support the standard Eagle drive. After unmounting that filesystem, you change into the /etc directory to remove the Eagle-linked disk dependent files there. Then, after changing into the root directory, you use "makedisk: to link the disk dependent files correctly and check the filesystem consistency. There is no UNIX usr filesystem on the Micro169 disk, because it is too small to support more than LISP and a small, debugging UNIX for our use.

Type CTRL-D to bring up multiuser UNIX, then install the microcode and band with load68 as usual. When using the load68 program you should respond

3

for the type of disk because you are using a Micro169 disk drive.

Type "sync" twice and press the RESET button to exit UNIX. Use "config" the usual way, and from then on you can merely use the command "superboot" to boot the system.

No LISP sources can be on a system with a 169 MB disk, because the disk is too small. Either a customer site reconfiguration tape or the equivalent networking instructions will be provided at the time of installation to enable you to network the 169 MB disk to a fileserver with sources.

6.21.4 Systems Using 474 MB Fujitsu Eagle Disk And 1/4" Tape:

To load the CMOS RAM, type

```
/tarq/setup clear eagle sp shell
init
```

To install the "test Streams root image" tape, type

```
copy quart $disk"
```

In load68. specify the tape drive you are using as "2".

To extract the UNIX usr files, type

```
tar xvpf /def/rqt0
```

6.21.5 Systems Using 169MB Fujitsu Disk and 1/4" Tape:

Set up the CMOS RAM by typing

```
/tarq/setup clear micro169 sp shell
init
```

Install the "test Streams root image" tape by typing

```
copy quart $disk
```

Make the UNIX usr filesystem by typing

```
csh makedisk micro169
```

When load68 prompts you for the disk type, respond

3

because you are using a Micro169 disk drive. When it prompts for the type of tape drive you are using, respond

2

because you are using a Cipher 1/4" tape drive.

Install the UNIX usr files by typing

tar xvpf /dev/rqt0

No LISP sources can be on a system with a 169 MB disk, because the disk is too small. Either a customer site reconfiguration tape or the equivalent networking instructions will be provided at the time of installation to enable you to network the 169 MB disk to a fileserver with sources.

For Non-Streams Systems

Now that the disk is restored, power down the machine and remove the CPU and any extra memory added for the installation. Use "config" to set up for the new board configuration.

When this is done, you can "lboot" as usual.

6.21.6 Reconfiguring for Customer Site

All LMI Lambdas at LMI expect the machine "LMI-Lambda-3" ("LAM3") as the fileserver, or system host. To break the dependence of a machine on LAM3, you must reconfigure for customer site. This can be done after the Release 1 sources have been loaded.

To reconfigure, boot LISP. Type

```
(login "lispm" t)

(si:set-pack-name "LAMA")
```

and then type

```
(disk-restore)
```

and type "yes" when prompted. When the machine reboots, type

```
(load "lama:release-1.customer-site;reconfigure)
```

and after this loads, type

```
(si:reconfigure-for-customer)
```

You will be prompted for the pack name. If this is your first machine, type "LAMA". If it is your second, call it "LAMB", and so on. The name will change as appropriate, and the reconfigured site files will be saved to the current band. From now on, the machine is independent of LMI's network and expects "LAMA" as system host.

6.22 Restoring the Root Image

Reloading the root image is a destructive procedure that replaces the current root image on the disk, including LISP boot files and the UNIX kernel. It does not affect the LISP or UNIX user files, nor the LISP disk label. If you hav done any work on the UNIX root on your machine, check with LMI to determine the safest course of action to tke before reloading the root. In any case, it is a good practice to backup your LISP and UNIX user files before making any major changes (such as this one) to the disk.

Mount the desired root image tape. Type at the SDU (after typing "init" as usual)

```
copy tape $disk
```

This will take about 20 minutes. When done, it will state that 8704000 bytes were copied and you will get a prompt. At this point, you can boot LISP and UNIX as before.

IMPORTANT: Don't mix up the order of "tape" and "disk" or the SDU will attempt to write the contents of the entire disk onto the tape, which will take a long time and create garbage. Also,, do NOT type either "initmini: or "initlabel" in the SDU or UNIX load program (which is what you would do if reloading the disk completely from tape). If you type these commands, the information about what LISP software is on the disk will be removed from the disk label and you will have to edit the disk label in LISP to restore this information.

6.23 Restoring UNIX User Files (assuming the user filesystem exists)

Mount the "LMI UNIX usr files" tape and type in UNIX

```
cd /usr
```

to change into the /usr directory, and then

```
tar xvpf /dev/rmt0
```

to extract the usr files from the tape. This will take about 20 minutes.

6.24 Warm and Cold Booting from the AI Keyboard

Booting from the AI keyboard is implemented as part of the Release 1 software upgrade. As long as the LISP processor is still performing memory cycles, you can warm and cold boot LISP from the AI keyboard. Unlike booting from the SDU, booting from the keyboard has no affect on any concurrently booted processors.

6.24.1 Warm Booting:

To warm boot LISP, hold down the keys CTRL-META-CTRL-META-RETURN on the keyboard. This will warm boot LISP, reloading the LISP band but not the LISP microcode.

A warm booted machine may have corrupted internal data structures. It will probably halt when it accesses one of these, so you should consider the machine to be in a very delicate state. Try not to run any more programs than the minimum needed to save your files. Get out of LISP and cold boot when possible.

If you are brave you can keep using the warm booted machine. If so, be careful, and be especially wary of a sudden crash.

6.24.2 Cold Booting:

If the LISP processor is still doing memory cycles, cold boot it by holding down the keys CTRL-META-CTRL-META-RUBOUT on the AI keyboard. This will reload the LISP microcode and the LISP band, with run bars appearing on the screen just as if you had booted with "lboot".

If you get no response for several seconds after attempting to boot from the keyboard, the processor is not doing memory cycles and you must reboot from the console.

In this case, you should gracefully take down any concurrently booted processors and then reboot in the usual way from the console.

6.25 Installing LISP Microcode and Band

When LMI sends you an updated microcode and band tape, or if you wish to install a new LISP microcode and band on your machine, you will not want to destroy information already on the disk. Likewise, you may wish to solve LISP software problems by reloading software, but without resorting to the drastic measures outlined in the "Restoring Disk from Tape" section of this manual. Whenever possible, you should avoid destructive commands such as the SDU "copy tape $disk" and especially the commands "initmini", "initlabel" and "(fs:initialize-file-system)".

The following procedures should be your first recourse when the microcode and band need reloading. The first procedure outlined is for use in UNIX; the second, for use in LISP.

6.25.1 In UNIX

If LISP won't boot or run reliably, you will have to use UNIX. Boot UNIX, load the tape containing the desired microcode and band, then type in UNIX

```
/lambda/load68
```

When you get the "Disk Loader>" prompt, type

```
printlabel
```

Look at the disk label and see where there are unused microcode and band partitions (with nothing in their comment fields). If none of the appropriate partitions are free, decide which are unnecessary and can be loaded over. When you have this information, type

```
load
```

The program will read the tape and will tell you which type of partition it is, microcode ("ulambda") or band (e.g., "1.52"). If it is a microcode partition and you know from the disk label that it should go on the partition "LMC4", type when prompted

```
lmc4
```

Similarly, if it is a band you want to put on LOD3, type

```
lod3
```

Continue in this way for whichever partitions in the tape you want.

To switch to this new software (so that it will be used when LISP is next booted), set the current microcode and band to these partitions (see "Setting Current Microcode and Band in UNIX").

6.25.2 In LISP

Once LISP is booted, load the LISP microcode and band tape and issue the (fs:tm-init) command if necessary. Type

```
(print-disk-label)
```

Look at the disk label and see where there are unused microcode and band partitions (with nothing in their comment fields). If none of the appropriate partitions are free, decide which are unnecessary and can be loaded over.

Suppose you have a tape containing two copies of a microcode you want loaded onto microcode partitions LMC4 and LMC5, followed by a LISP band you want loaded onto band partition LOD3. Type

```
(fs:restore-magtape)
```

This will read the tape header to see what microcode and LISP bands are on it, and will prompt you for where to load them. Follow the instructions as prompted.

If restore-magtape tells you it got an error reading the tape header, rewind the tape or try again. If it happens again, there may be a bug in your tape software. Restore the tape by typing (and answering "yes" when prompted).

```
(si: copy-disk-partition "mt" "" 0 "lmc4")
```

then

```
(si:copy-disk-partition "mt" "" 0 "lmc5")
```

and finally

```
(si: copy-disk-partition "mt" "" 0 "lod3")
```

To switch to this new software, set the current band and microcode to these partitions and reload the partitions. (See "Setting Current Microcode and Band in LISP".)

6.26 Setting Current Microcode and Band in UNIX

You can set the current microcode and band for LISP by using the UNIX band loading program. Type in UNIX

```
/lambda/load68
```

To see what microcode and band partitions are currently in use, type at the "Disk Loader>" prompt

```
printlabel
```

to look at the disk label. The third line of text says which partitions are the current microcode ("microload") and band. There is also an asterisk to the left of the current microcode and band partitions on the disk label.

To change to the band on LOD3, type at the "Disk Loader>" prompt (still in the /lambda/load68 program)

```
setband lod3
```

and to change the current microcode partition to LMC4, type

```
setmload lmc4
```

The next time LISP boots, it will do so using the microcode in LMC4 and the LISP band on LOD3.

6.27 Setting Current Microload and Band in LISP

To see what microcode and band partitions are currently in use, type

```
(print-disk-label)
```

to look at the disk label. The third line of text says which partitions are the current microcode ("microload") and band. There is also an asterisk to the left of the current microcode and band partitions on the disk label.

To change to the band on LOD3, type

```
(set-current-band 3)
```

To change the current microcode partition to LMC4, type

```
(set-current-microload 4)
```

These commmands set up the environment so that the next time LISP cold boots it will do so on the partitions just set.

If you changed the current band partition but not the current microcode partition, you can start using the new band, with the old microcode, without rebooting LISP. Before doing so, take all the precautions usually taken before halting LISP: save all your work, make sure no one is using your machine as a fileserver, and warn anyone who might decide to start using it. Then type the command

```
(disk-restore)
```

The machine will ask you if you really want to reload the partition; you should say "yes". A "disk restore" is like a cold boot except that it does not reload the microcode. When you again see the "keyboard" prompt at the bottom of the screen (as usual, when booting completes), the machine will be booted on the partition you just set.

6.28 Enabling and Disabling Terminal Lines

The UNIX file "/etc/ttys" contains information used by UNIX about which terminal devices are enabled when UNIX is up multiuser. The Beta Streams Release 1 Root image has the following contents:

```
04rsd
17ttya
07ttyb
17tty10
   07tty11
   17ttyp0
   17ttyp1
   17ttyp2
   17ttyp3
```

The first digit is a "1" for every line that is enabled, a "0" for disabled lines. Each enabled line will have UNIX "getty" process running on it when UNIX is booted multiuser.

The second digit is the getty parameters for the line. For each normal resolution 9600 baud terminal added to the system, this should be "7".

The "rsd" entry stands for "raster scan device", the high resolution monitor. This is disabled in a Streams system so that LISP instead of UNIX can control the high resolution monitor.

The "ttya" and "ttyb" entries refer to SDU ports A and B, respectively.

The "tty10" - "tty12" entries (the second to last character is an "L", not a "one") are logical terminal lines on which the LISP processor can have UNIX getty processes.

The "ttyp0" - "ttyp3" entries are "pseudo-ttys", for using "supdup" to access another machine on the network.

A printer should not have a getty on it (that is, it should not receive a login prompt). Lines on which printers are connected must be disabled. SDU port B (ttyb) is appropriately disabled here.

To have a UNIX window on the high resolution monitor (by saying SYSTEM-U, for example), line "tty10" must be enabled. If it is not, LISP will hang with a message that it is waiting for a 68000 login process.

If this happens, log in as "root" on a normal resolution terminal (a Z29, for instance) and edit /etc/ttys to enable line tty10. Then, type

```
kill -1 1
```

to send a "signal 1" to process 1. This starts the initialization process again, using the new /etc/ttys file. LISP will now continue as it should.

6.29 Initializing the Tape Controller

In Release 1, if you boot with "lboot" you will have to initialize the tape controller. To do so, type in LISP

```
(fs:tm-init)
```

This command replaces the command (fs:tapemaster-initialize) used in previous releases. To check the status of the tape drive (as the "tapemaster-initialize" command used to), type in LISP

```
(fs:tm-status)
```

The "tm-init" command is not necessary if the machine is booted using the "superboot" command. The "superboot" command

initializes the tape controller during the booting process.

6.30 Freeing a LISP-owned Tape Drive

In certain circumstances, an aborted LISP tape operation can leave LISP in control of the tape drive even when this is no longer necessary. Such a case will case the message "Tape drive owned by processor in slot 0" to be printed if the drive is accessed through UNIX.

If this happens, type in UNIX

```
/etc/freemt
```

to release the tape drive from LISP's control. After this program runs, either processor should be able to access the tape drive.

6.31 Powering Down the Machine

Leave LISP or UNIX gracefully (see the sections on LEAVING LISP GRACEFULLY and LEAVING UNIX GRACEFULLY). Spin down the disk by turning the Start switch to off. (Off is the down position.) Wait about two seconds. Turn off the main rocker switch on the AC distribution panel (this is the main circuit breaker).

Keep in mind that any peripherals plugged into the AC distribution panel will also get powered down when you turn off the main circuit breaker. In particular, make sure that you have rewound and unloaded any tape in the tape drive. If you power down a loaded tape, it will read "unloaded" but will really be locked into the drive when you next power on the drive. If this happens, press "unload" and the tape will unload normally.

Turn off the SDU console and the high-resolution monitor.

If desired, unplug the console monitor and LMI Lambda.

6.32 Emergency Power Down

In case of an emergency, such as smoke, fire, or exposure to water, the following procedure will minimize the chances of damaging your machine.

IMMEDIATELY:

- Turn off (down) the main circuit breaker on the back of the LMI Lambda, or
- Unplug the LMI Lambda's main power cord.

Note: The main power cord has a twist-lock plug to prevent accidental power loss, so will NOT pull straight out of the wall. Flipping the switch is faster.

Once you have done this, call LMI.

6.33 Running Diagnostics

Diagnostics for the various boards can be run from tape. Mount the "LMI diagnostic tape including setup and lam" and then type at the SDU (after typing "init") any of the following commands, followed by a RETURN.

CARD	
Memory slot 9	/tar/ram -tvsS 9
Memory slot 10	/tar/ram -tvsS 10
Memory slot 11	/tar/ram -tvsS 11
Memory slot 12	/tar/ram -tvsS 12
Memory slot 13	/tar/ram -tvsS 13
Memory slot 14	/tar/ram -tvsS 14
VCMEM	/tar/vcmem -tvsS 8
CPU	/tar/cpu -tvsS 11
Tape subassembly	/tar/tape -tvs
Disk subassembly	/tar/2181 -tvs
LISP Processor	/tar/lam

Memory and VCMEM tests take about an hour to run. However, you can type CTRL-C at any time to stop the test. This way, you can at least know how the first several tests come out (they go quickly).

The CPU diagnostic is extremely memory-intensive. If you get the message "out of memory, can't run cpu", type "init" and try again. If you forget to specify the slot number of the CPU, the diagnostic will fail.

If you have a problem with the tape subassembly, running a diagnostic from tape will probably be useless. The tape diagnostic is also on the disk. To run this diagnostic, type at the SDU

```
/disk/monitor/diag/tape -tvs
```

If you suspect a tape drive problem but none is found by the tape diagnostic, use the more extensive diagnostic option. To do so, mount a scrap tape with a write ring and type

```
/disk/monitor/diag/tape -tvsD
```

This runs the "destructive" tape diagnostic, named because it writes on the tape (thereby destroying any data on the tape). This test takes about 10 minutes to complete.

Chapter 7

Troubleshooting by Area

The following sections summarize some symptoms of malfunctions that may appear in your LMI Lambda, and ways in which these can be quickly corrected.

This manual divides troubleshooting problems into two chapters. Use the first if you are reasonably sure where the problem lies. The second chapter details problems for which you may be unsure of the cause, or which may have several causes.

In all cases, the manual details the simplest debugging procedures first. Try these before going on to more complex ones.

When in doubt, or when these procedures don't work, please call us. LMI's Customer Service Help Line is open to serve you from (9-5 (Eastern Time), Monday through Friday, at (617)876-6819. This service is free to all LMI customers.

7.1 SDU Operations

Symptom — Nothing lights on the front panel when you power up the machine.

Response — Either the front panel lights or the SDU are unplugged. Reconnect whichever is appropriate, and call LMI with any questions.

Symptom — The red SETUP light on the front panel goes on and stays on when you power up the machine.

Response — The battery that backs up the CMOS RAM has run down. (Since the CMOS RAM on the SDU contains essential configuration information, the SDU will not operate correctly without it.) The CMOS RAM needs to be set up. At this point, your machine is set to have the console on SDU Port A at 300 baud, so you must set your console terminal to 300 baud before continuing.

Use the LMI "tar" diagnostic tape. Load it and put it online, then type at the SDU (after typing "init", if necessary)

/tar/setup clear eagle sp shell

In a minute or two, you will get a prompt. Type

init

When you type "init", both red lights will come on; then they will go off and the green RUN light will go on. The console will now be set up for the usual 9600 baud, so you must change the console terminal to 9600 baud to see the prompt. Everything should now be back to normal.

If something does go wrong (for instance, a multibus timeout occurs even though "init" has been typed), type

/tar/setup

This will print out the contents of the CMOS RAM. If everything is zero, try again typing

/tar/setup clear eagle sp shell

init

and look at /tar/setup again. It should have numbers other than 0, and should have entries for the disk. If not, something more serious is wrong. Call LMI.

The battery recharges itself whenever the power is on, so you need not worry about replacing it.

Symptom

Nothing happens when the machine is powered up.

Response

Plug in the machine, and plug into the AC distribution panel all power cords inside the machine.

Make sure the main circuit breaker is on and the machine is plugged into a socket that has power. Make sure the 3 position switch to the right of the main breaker is on "local".

If there is still no response, check the fuse in the power distribution box.

Symptom

Nothing appears on the console screen once the machine is powered up.

Response

Make sure the console monitor is plugged in and turned on, and check its fuse.

Symptom

A cursor is flashing on the console screen, but no SDU message or prompt appears on the screen. The green RUN light is lit on the machine's front panel.

Response

Make sure the console cable is connected to SDU Port A. Make sure the rotary switch on the back panel is set to position 1. If it is not, switch it to 1 and hit the RESET button. If your console is a Z29, make sure the console cable is attached to the top port at the back of the console, and the console is setup to 9600 baud and "Auxiliary Mode". If you still have no luck, try changing your Z29 setup to 300 baud. Power cycle the console, then the machine.

If that doesn't fix it, one of the boards may be wedging the bus. Power down the machine, remove all boards except the SDU, power up the machine and see if a prompt appears. If it does, try to determine which board is at fault by using the following procedure:

Since it is very unlikely that the memory boards cause this sort of problem, power down the machine, put the memory boards back where they belong, power up the machine, and see if there is a prompt. If not, power cycle the machine, removing one memory board at a time, until the bad one is isolated.

If there is a prompt, continue power cycling the machine, inserting one additional board from those belonging in slots 8-20 each time, until the prompt fails to appear. Whichever board stops the prompt is the culprit. If the prompt appears when all boards in slots 8-20 are in place, the trouble is with the LISP processor. Power down the machine, insert the RG, CM and DP boards (i.e., all the LISP processor boards except the MI), power up the machine and verify that there is a prompt. The MI board is the only one of the LISP boards that can be a bus master, so it is the only one that should be able to wedge the bus. When a prompt appears, power down, insert the MI board, power up, and verify that the MI board causes no prompt to appear.

In the unlikely case that one of the RG, CM, or DP boards is the problem, power down the machine and remove all four LISP boards. Insert the RG board, power up, and look for a prompt. If there is none, the RG board is bad. If there is one, power down, insert the CM or DP board, power up, and try again. If this does not isolate the bad board, power down, insert whichever board (CM or DP) was not inserted last time, power up, and try again. Verify that this last board causes the problem.

Symptom — The "ls" command from the SDU does not list the name of a file that "ls" in UNIX shows.

Response — This is a bug in the software for the SDU "ls" command. Although annoying, it is harmless. If you are concerned that a UNIX file seems to be missing, boot UNIX to look for it instead of relying on the SDU.

Symptom — The SDU command "init", or nearly any other command other than "reset" and "help" causes the machine to hang.

Response — The bus is wedging. Very likely, the Multibus jumper (connecting pins A31 and B31 on the back of the slot 15 Multibus connector) is missing. First check to see if that jumper is missing, and if so, replace it. This will solve the problem.

Most commands use either the Nubus or the Multibus, and the "init" command to verify that the wedging occurs as soon as the buses are enabled. Press the RESET button, then type

```
reset
```

to reset the machine. Next, type the command

```
enable
```

When the machine hangs, press the RESET button and type "reset" again. This time, try to enable only the Multibus. To do so, type

```
enable -m
```

If this does not work, something is wrong with the Multibus. Doublecheck to make sure the Multibus jumper mentioned above is installed properly, as this is by far the most common cause of this symptom. If there is still a problem, power down the machine and remove all boards in slots 16-20. Power up the machine again and type "enable -m" again. If it works this time, one of the Multibus boards was wedging the bus. If it still does not work, something is wrong with the Multibus backplane.

If "enable -m" works, try to enable the Nubus by typing

```
enable -n
```

If this does not work, something is wrong with the Nubus. Power down the machine, remove all the boards in slots 0-14, then type "reset" and "enable -n" again. If it works this time, one of the boards just removed was wedging the bus. If it fails again, something is wrong with the Nubus backplane.

7.2 Problems in the SDU/UNIX Environment

If the system gets so wedged that typing CTRL-C on the console doesn't cause a prompt, press the RESET button. It is the square button just to the right of the rotary switch on the back of the machine. RESET should cause a prompt from the SDU.

If the console seems unable to communicate with the SDU, check whether it is plugged in and is on, then check the SETUP mode. The console should be in "Auxiliary Mode" and set at 9600 baud, the monitor should be off, and it should be on line.

7.3 Disk Drive Operations

Symptom	The ON light on the disk drive does not come on, although there is power to other parts of the machine (fans, card cage).
Response	Turn off the main circuit breaker. Make sure the disk drive is plugged in and the circuit breaker on the lower right rear of the drive is on. Next, check the fuse on the disk drive.
Symptom	The FAULT light comes on.
Response	Reseat the 1" wide multicolored ribbon cable on the right rear of the drive near the drive's main circuit breaker. If the FALT light remains lit, a cable under the disk enclosure needs reseating. Call LMI.
Symptom	The ON light is on, but the READY light doesn't come on when you press the START switch to spin up the disk.

Response — Turn the READY switch off and try again. Next, make sure that the two-position toggle switch on the right rear of the disk drive is switched to "Local". If not, switch it and the drive should spin up.

Symptom — The ON light on the disk drive is on, but as soon as you turn on the START switch, the circuit breaker on the back of the disk drive shuts off.

Response — Almost certainly a large power surge has burned out some power transistors on the disk drive power amplifier board. The board will need to be sent back to Fujitsu for repairs. Call LMI.

Symptom — The disk drive is working (the READY and ON lights are on), but a "can't access disk" error message appears when you start the SDU.

Response — First make sure that the PROTECT switch is not on. (The PROTECT switch is located on the front panel next to the START switch.) Next, open the front panel and check visually to make sure that the disk controller board is in the correct slot in the card cage (Slot 16). You do not need to power down to do this.

If the board is in the wrong slot, power down the machine and remove and reinsert the board.

Check to make sure that the two ribbon cables (60-pin and 26-pin) that connect the disk controller to the disk drive are firmly connected at each end--at the left rear of the disk drive and at the top and bottom connectors at the back of Slot 16.

Next, check at the back of the card cage to see if the pins of the DIN connectors have been bent or are touching each other in any way.

If none of the above items is the source of the problem, call LMI.

The SDU will notify you of disk drive problems by printing a terse error message. It will function, but will not be able to access the disk drive.

7.3.1 Reformatting the Disk

This is a destructive procedure that will indiscriminately destroy all information on the disk--the Root image, UNIX, LISP, and all of the user files. Although LMI has backup copies of the Root, UNIX, and LISP, only you have tape backup copies of your files. ONLY FILES THAT HAVE BEEN MANUALLY BACKED UP ON TAPE WILL EXIST AFTER THE DISK IS REFORMATTED.

Information in the SDU's CMOS RAM determines how the disk will be formatted. The SDU must be set up for the 474MB Fujitsu Eagle disk drive or the 169MB Micro disk drive, as appropriate, before reformatting.

For the 474MB disk, mount the LMI diagnostic tape in a 1/2" tape drive and type at the SDU

```
/tar/setup clear eagle sp shell

init
```

For the 169MB disk, mount the LMI diagnostic tape in a 1/2" tape drive and type at the SDU

```
/tar/setup clear micro169 sp shell

init
```

If your system has a 1/4" tape drive in place of a 1/2" tape drive, type "tarq" instead of "tar" when running the setup program.

Reformat the disk (it will take about an hour) with the command

```
/tar/2181 -tvsFD
```

To be safe, run the disk diagnostic after this is done. Sometimes the disk controller (or formatting program, or something) doesn't manage to format some tracks. Type

```
/tar/2181 -tvs
```

At the end of the diagnostic, any bad tracks will be printed. Reformat these tracks. If only one track is bad (for instance, track 1234), type a range of tracks around that one:

```
/tar/2181 -tvsFD -A 1232-1236
```

If several tracks in a range are bad, reformat that range (for instance, tracks from 3000 to 3025) with

```
/tar/2181 -tvsFD -A 3000-3025
```

(All track numbers are decimal, as the SDU expects.)

Run the diagnostic again for just those reformatted tracks:

```
/tar/2181 -tvs -A 1234

/tar/2181 -tvs -A 3000-3025
```

7.3.2 Running Memory Tests

In cases where it seems memory boards cannot be accessed, try using the SDU to do writes and reads to the boards in question.

The writes and reads on the SDU work only with memory boards that are listed in the configuration file. Boards that are not in this file will get timeouts. To see which boards are in the file, type at the SDU

```
cat /disk/monitor/conf/std
```

Each memory board in the file will have a line like:

```
ram     FC000000     200
```

where FC000000 is the slot number, in hexadecimal (so this example is the slot 12 memory board).

If this file shows memory boards for only "FC000000" and "FA000000", only SDU writes and reads to slots 12 and 9 will work; boards in slots 10, 11, 13 and 14 will get timeouts.

The SDU Manual explains about write and read commands. Briefly, to write a word containing the number 400 (decimal) to the zeroth location of the memory board in slot 12, type at the SDU

```
w -w FC000000 400
```

and to read that word, type

```
r -w FC000000
```

Note: You must always do a write before the read in this test.

7.4 Tape Drive Operations

The SDU console can test the tape. You will be most likely to encounter tape drive problems in the form of an error message from the SDU, or from UNIX or LISP when you try to use the tape with one of these processors.

NOTE: In LISP you must type

```
(fs:tm-init)
```

before using the tape if you booted with "lboot" and not "superboot".

First, make sure the tape controller and the cables are firmly plugged in. The tape controller should be in slot 17, the slot to the left of the disk controller.

On a 1/2" Cipher tape drive, one cable (part number 2338206) goes from P1 at the left rear of the drive to the bottom connector on the tape controller. At P1, the red line on the cable should be on the right. At the tape controller, the red line on the cable is on the bottom and the cable is flush against the top of the connector. The other cable, part number 2338207, goes from P2 at the left of the drive to the top connector on the tape controller. At P2, the red line on the cable is again on the right. At the controller, the red line is still on the bottom; the cable is flush against the bottom of the connector.

On the Kennedy reel-to-reel tape drive the bottom connector on the tape controller plugs into J5 on the drive and the top connector plugs into J1 on the drive. The red line on the cable should be on the LEFT on both J1 and J5 on the Kennedy. (Note that this differs from the orientation on the Cipher.) Note also that the drive and formatter board must be set to 75 ips (see "Installing the Optional Kennedy Tape Drive" earlier in this manual).

After you've checked that everything is plugged in, see if the tape works from the SDU. Insert the diagnostic tape and say

```
/tar/2181
```

This will start a disk diagnostic, which will tell you whether the tape works. You don't have to keep running the diagnostic; stop it by typing CTRL-C when you've seen as much as you want.

Next, type at the SDU

```
init
```

Then type

```
/disk/monitor/diag/tape -tvs
```

Write down any error messages you get and what was being done when you got them, and call us at LMI to describe the problem.

If the tape works from the SDU, try it with LISP. In LISP,

```
(fs:tm-status)
```

will print information about the tape drive and controller.

Exercise the system by loading a tape that you know works and restoring it with

```
(fs:restore-magtape ':query nil)
```

If it still doesn't work, the problem is likely to be a hardware failure; try replacing the tape controller, cables, or tape drive.

A common way for tape drives to fail is for the machine to think the drive is offline although it is actually online. If this appears to be the case, run the tape diagnostic at the SDU to see if the problem shows up there. If so, make sure the cables are plugged in solidly. If the problem persists, there is a problem with either the tape controller or the Cipher tape drive (probably with its interface board). Try swapping the controller first, then call LMI to arrange for the replacement or repair of the drive.

If the drive passes the SDU diagnostic, try it with UNIX and with LISP to see if it fails with either or both. Call LMI and report.

Although the tape controller and the SDU are the two boards that most commonly cause tape problems, such problems can also be caused by the UNIX or LISP processor or the disk controller. Historically, most problems have been failures with the cables, tape controllers, and tape drives.

If you suspect a tape drive problem, you should run a more extensive tape diagnostic test. To do so, mount a scrap tape with a write ring and type

```
/disk/monitor/diag/tape -tvsD
```

This runs the "destructive" tape diagnostic, named because it writes on the tape (thereby destroying any data on the tape). This test takes about 10 minutes to complete.

7.5 Cabling Operations

Four types of cabling problems generally occur.

Symptom	The piece of equipment in question won't function.
Response	The cabling is improperly seated or not plugged in. Plug in the cable.
Symptom	Cabling is plugged in, but piece of equipment still won't function.
Response	Cabling is plugged in upside down, or is defective. Most cabling for the LMI Lambda is keyed to prevent improper insertion. However, the ribbon cables for the disk drive and tape drive could come loose or be inserted upside down. Turn off the machine, leave it off, and call LMI if you suspect problems. Cables plugged in upside down can conceivably damage the machine, although this is unlikely.
Symptom	Equipment does not work, or works but gives incorrect responses.
Response	Cabling is plugged into wrong slot, or is defective. This can cause shorts, resulting in possibly severe damage to the machine. If any cabling is disconnected, i.e. in shipping, avoid problems by playing it safe: DO NOT POWER UP THE MACHINE until you have spoken with LMI.
Symptom	Cable is frayed or pinched.
Response	Cable may be shorted or broken. Check all cables carefully before powering up the machine. Send any suspicious ones back to LMI for replacement rather than risking a short that could seriously damage the equipment.

7.6 Monitor, Keyboard and Mouse Operations

Symptom — The monitor display is comprehensible but distorted; i.e., everything is shifted to right, out of proportion, etc., or screen has funny lines in background or contrast problems, but screen still contains recognizable data.

Response — The VCMEM board may be bad. Call LMI for a replacement VCMEM board first, unless the problem is obviously with the monitor.

The monitor may be out of adjustment. See the monitor manual for monitor adjustment. Monitor adjustment should be attempted by skilled personnel only.

DANGER: The monitor contains extremely high voltages. Plastic tools should be used to protect both you and the equipment.

Symptom — The monitor doesn't light up after you plug it in and turn it on.

Response — Check fuses.

Check cabling. All external monitor cabling is keyed to prevent improper insertion.

If possible, attempt to use the monitor with a different monitor cable and the cable with a different monitor, to determine which part causes the problem.

Symptom — The monitor works but the keyboard does not.

Response — Wait about one minute and try again. Next, press the keyboard's RESET button and wait again. Some of the keyboards may contain defective microprocessors, which take too long to warm up. LMI will replace any such keyboard.

If there continues to be no response from the keyboard, there may be a problem with the VCMEM board, cable or monitor. Call LMI.

Symptom — The screen is suddenly full of garbage while the machine is in operation.

Response — The probable cause is a voltage surge or brownout (accompanied by dimming of the lights).

First press the CLEAR SCREEN key. If the screen clears but does not refresh, try CTRL-L to renew the display.

If the CLEAR SCREEN key and CTRL-L do not work, check the monitor. If it is at fault, you can disconnect it and reconnect another monitor to a live keyboard. If there is still a problem, the VCMEM board may be defective; call LMI for a replacement.

Symptom — Mouse doesn't function.

Response — First check mouse cabling to make sure that it is plugged in. (Mouse cabling is keyed, so improper insertion should not be an issue.)

If the mouse cabling is properly plugged into the monitor, there may be a problem with the mouse, the VCMEM board or cables, or the internal monitor cabling. Call LMI.

7.7 The "Why" Program

If LISP crashes, we might ask you to run the "why" program to give us more information. After a LISP crash, get back to SDU command level (by typing CTRL-Z if booted with "lboot:, or exiting UNIX gracefully then pressing the RESET button and typing "init" if using Streams) and type

```
why
```

When the screen is full, type CTRL-S to stop the text from scrolling off. You can restart the scrolling by typing CTRL-Q. This program will give some information about the LISP processor's state, and will suggest a cause of the crash if possible (a bus timeout, for instance). The program prints just over one screenful of text.

Do not run the program more than once. It will give valid data only the first time it is run after a crash, and if run again will no longer represent the state of the machine at crash time.

7.8 Crashes in LISP

Whenever LISP has been halted with files open, type to the LISP Listener after LISP is booted

```
(fs:lm-salvage)
```

This will take ten minutes or less, and will make sure the file system is consistent.

This should also be done every two weeks, as a precautionary measure. (See in Chapter 6 the section "Booting LISP.")

7.9 Crashes in UNIX

This procedure is also useful the first time UNIX is brought up after power on.

Do "uboot" in the usual way. While still in single-user mode (before typing CTRL-D), type

```
/etc/fsck -y
```

This runs the "file system consistency check" program. With the "-y" option, it will automatically do any appropriate cleaning up, if necessary, to restore for you a consistent UNIX file system.

If you do not use the "-y" option, if UNIX finds an inconsistency it will ask you whether it should clean it up. Generally it is safe to say yes. If you are concerned about whether to answer yes, call the LMI help line. Information about what happened to make the file system inconsistent (accidental powerdown, crash, etc.) will help us understand the problem.

If "fsck" finds no errors, you will simply get a prompt. Type CTRL-D to bring up multiuser UNIX. Automatically, "/etc/fsck" is run after you type CTRL-D.

If "fsck" does some cleaning up, it may say "BOOT UNIX--NO SYNC" when it is done. Press the RESET button on the back of the machine to leave UNIX in this case. (Don't type "sync" before pressing RESET.) Once you are back at the SDU, type "init" and boot UNIX in the usual way. This will give you a consistent file system.

7.10 Exercising UNIX

If you suspect something may be wrong with your UNIX system, try testing disk and CPU operation using the "exercise" command.

Boot UNIX singleuser. Type

```
exercise
```

Every 30-60 seconds, a message will print to let you know the test is running. This test is an infinite loop, and so will not stop until you instruct it.

To stop this test, type CTRL-C. This will halt the exercise, and both the resulting number of CPU and disk operations run and the number sucessfully completed will be displayed.

7.10.1 Exercising LISP

If you suspect something may be wrong with your LISP system, you might want it to run a test until it crashes. To perform a test of the LISP boards, memory and disk operation without risking a programmer's work, type

```
(do-forever (apropos ""))
```

Then, type TERMINAL Om (that means you should depress the key labelled TERMINAL, then the "zero" key, then the m) to turn off "more processing". This way, text will scroll off the screen without waiting for you to type a space at the end of the screenful.

This will print all the known symbols in the LISP world whose printed representations have the null string (" ") as a substring, forever, until you stop it. To stop, type CTRL-ABORT.

Chapter 8

Troubleshooting by Symptoms

Symptom

The SDU command "init", or nearly any other command other than "reset" and "help", causes the machine to hang.

Response

The bus is wedging. Very likely, the machine is missing the Multibus jumper (connecting pins A31 and B31 on the back of the slot 15 Multibus connector). First check to see if that jumper is missing, and if so replace it. Press the RESET button, then type

```
reset
```

to reset the machine. Next, type the command

```
enable
```

When the machine hangs, press the RESET button and type "reset" again. This time, try to enable only the Multibus. To do so, type

```
enable -m
```

If this does not work, something is wrong with the Multibus. Doublecheck to make sure the Multibus jumper mentioned above is installed properly, as this is by far the most common cause of this symptom. If there is still a problem, power down the machine and remove all boards in slots 16-20. Power up the machine again and type "enable -m" again. If it works this time, one of the Multibus boards was wedging the bus. If it still does not work, something is wrong with the Multibus backplane.

If "enable -m" works, try to enable the Nubus by typing

```
enable -n
```

If this does not work, something is wrong with the

Nubus. Power down the machine, remove all the boards in slots 0-14 then type "reset" and "enable -n" again. If it works this time, one of the boards just removed was wedging the bus. If it fails again, something is wrong with the Nubus backplane. Call LMI and report the problem so that we can replace the defective backplane.

Symptom — You have just powered up the machine, but don't get any ">>" prompt on the console.

Possible Causes: Terminal is set at wrong baud rate.

Rotary switch on back of machine is in wrong position.

Terminal RS232 cable isn't plugged into right port (male port A in machine to female "DCE" port on Zenith).

SDU paddle card, cable, or backplane is malfunctioning.

SDU is bad.

What to Do: Check the cables and connections. Set the baud rate to 300 and the rotary switch to position 0. Hit the RESET button on the back of the machine. If you still don't get a prompt, then probably the SDU paddle card, cable, or backplane has a loose connection. Try wiggling and tightening all connections. This will take some patience, and you should not give up until you have tried for at least five minutes.

If you still have no luck, type

```
init
```

to the terminal. If the two red lights on the front door go on, then you can be sure that the problem is a loose connection, since the machine is receiving input from the terminal, but isn't able to send output successfully.

If you have extras, replace the SDU paddlecard and its cables. Try swapping with another machine's SDU.

Supposedly rotary switch positions 3 and 4 allow the high-resolution monitor and port B to be used as the system consoles, instead of port A. This feature has never actually been seen to work.

Symptom — You are able to talk to the SDU monitor at 9600 baud and "init" works. However, when you type "ls /disk", an error message appears.

Possible Causes: The SDU has lost its CMOS RAM information.

The disk is not spun up.

The disk has not been formatted.

There is no Root file system on the disk.

The "Disk protect" switch is on or the disk heads are locked.

You have a bad SDU.

What to Do: First make sure the disk drive is on and spun up.

If the red SETUP light on the front panel is on, reload the CMOS RAM (with /tar/setup) and that will fix the problem.

Try typing CTRL-C and then "init".

If you suspect that the disk needs to be formatted, check by loading the diagnostic tape and running "/tar/2181 -tvs -A 0-100", which will check the format of the first 100 tracks on the disk.

More likely, though, the Root file system has not been copied to disk. Call LMI.

Symptom — "ls /disk" works when typed to the SDU monitor, but "uboot" doesn't work (for instance, it returns a "NuBus/Multibus timeout error".

Possible Causes: The "config" program has not been run.

The 68000 board is bad.

A memory board is bad.

The configuration file, or one of the other files needed by UNIX (/etc/rc, /etc/fstab, /etc/checklist) is missing.

The Root file system may have been damaged.

You forgot to type "init".

What to Do: Type "init" and try "uboot" again.

List the contents of the /disk/monitor/bin directory, and make sure there is a file named "uboot" there.

Run "config", then try to boot.

If the other files (/etc/rc, /etc/fstab, /etc/checklist) are missing, then you will have to reload the Root file system. This is a destructive procedure, so call LMI first to see if anything else can be done. If not, put the LMI Root tape in and type

```
copy tape $disk
```

It takes about 20 minutes.

If there happens to be another configuration file on the disk (this file is sometimes named /disk/monitor/conf/std.old), you can have "uboot" use this file instead of the standard one by typing

```
uboot -C /disk/monitor/conf/std.old
```

instead of just "uboot".

Run memory board diagnostics.

Try swapping 68000 boards.

Symptom "uboot" seems to be working--it printed out a welcome message and version number--but it never prints out a "#" prompt or accepts terminal input.

Possible Causes: This is a random software bug, which happens on rare occasions. It is not destructive, just inconvenient.

What to Do: Hit the back panel RESET button, type "init", and "uboot" again.

Symptom You get flaky, unreliable, and inconsistent behavior of the SDU monitor or UNIX. (The system hangs or crashes while running innocuous programs, or when you are typing at the keyboard; when you run disk diagnostics you get many random errors; programs behave inconsistently and illogically.)

Possible Causes: Loose or bad cable connections.

Bad SDU board or 68000 board.

What to Do: Try running the NuMachine diagnostics from the SDU to see which board, if any, is bad.

Check cabling.

Swap SDU or 68000 boards.

Symptom You get "bad track" errors when formatting the disk.

Possible Causes: Nothing is really wrong; bad tracks are common (although usually there are fewer than ~20). If you seem to be getting substantially more, make sure the disk and tape cables are securely connected.

What to Do: Reformat the bad tracks. It is usually best to reformat some of the surrounding tracks also. Thus to reformat track 1012, type

```
/tar/2181 -tvsDF -A 1010-1014
```

If this fails, then try again; sometimes it takes two or three tries before a good track is found.

When you have finished formatting the bad sectors, it is a good idea to check the entire disk again by typing

```
/tar/2181 -tvs
```

Symptom You have problems booting UNIX.

Possible Causes: A disk or cable problem, or a software problem.

What to Do: Try booting LISP to see if it works. In most cases, if LISP boots it is a UNIX software problem instead of a disk problem. The LMI Root Image will need reloading. This operation requires the reloading of all UNIX files and of the entire LISP system, so be sure to make a tape backup of all LISP user files before reloading the root.

Symptom The AI keyboard works in the LISP world but not in multiuser UNIX.

Possible Causes: Defective VCMEM board.

What to Do: Replace the VCMEM board.

Symptom LISP doesn't boot. Typing "lboot" results in an error message; "lboot" gets an error while executing the dribble file; or the run bars have appeared on the high-resolution monitor, but the system craches before it finishes.

Possible Causes: You forgot to type "init" after power-up and before running "lboot".

The "lboot" program isn't on the disk, or it hasn't been linked to the file /disk/monitor/bin/lboot, or it is damaged.

The memory board positions haven't been specified properly via the "memsetup" command.

The processor speed hasn't been set via the "setspeed" command.

The current band and microcode haven't been loaded or set.

The boards are being run too fast.

One of the LMI Lambda boards has some sort of a hardware problem (for instance, bent pins or a wiring problem).

Appendix A

Preventive Maintenance

The Lambda LISP Machine is designed to require hardly any maintenance. The card cage and boards, power supply, power controller, and fans do not require preventive maintenance.

A.1 Filters

The filter is located on a sliding tray under the fan. To clean it, simply slide out the tray. The filter (Fujitsu part number is B90L-0400-0303A) should be replaced as needed.

A.2 Cipher Tape Drive

Preventive Maintenance Schedule

Maintenance Operation	Frequency
Clean head	daily [if in frequent use]
Clean guides	daily [if in frequent use]
Clean tachometer roller	daily [if in frequent use]
Clean tape cleaner	daily [if in frequent use]
Clean reel-hub pads	daily [if in frequent use]
Replace reel motors	when worn [at least 5,000 hrs.]
Clean filter	1,000 hrs.

Cleaning of Tape Path Components

Clean the tape path using 1,1,1 trichloroethane. Use a moistened cotton swab to wipe the tachometer roller and guides, head, and tape scraper blades and housing. When cleaning tape cleaner, be careful to avoid damaging the sharp edges of the tape cleaner blades.

IMPORTANT: Do not use alcohol or other solvents. Do not allow solvent to come into contact with the tachometer disk, located in the tachometer housing. Permanent damage to the tachometer assembly may result.

Housing

Clean the dust door and control panel with Miller Stephenson Chemical Co. MS-260, Windex, or an equivalent, commercial-grade plastic cleaner.

Reel Hub Pads

Wipe away any debris clinging to the reel hub pads with a cotton swab moistened with tape path cleaner (trichlorofluoroethane).

Tape Drive Filter

1. Disconnect the transport from AC power source.
2. Place the drive in access position:
    1. Release the rack retaining mechanism located on the lower left-hand side of the front panel.
    2. Hold the front panel firmly and withdraw the drive on its slides until the first lock engages.
    3. Holding each side of the front cover, raise the cover to its locked, retained position.
3. Slide the air duct tube forward by depressing the side of the tube near the front panel and disengaging the retaining tab.
4. Release the Ny-Latch fastener securing the filter adapter to the front of the power supply assembly and remove the filter adapter.
5. Remove the filter from the filter adapter.
6. Blow off loose dirt with compressed air.
7. Replace air filter in filter adapter and replace filter adapter to the front of power supply assembly and replace

the air duct tube.

Once every two years, schedule time (a maximum of two hours) for an LMI Service Representative to perform a routine Preventive Maintenance on your equipment. The dates and hours should be scheduled in advance.

Appendix B

Disk Controller Switches and Jumpers

B.1 SMD 2181 Dip Switch Settings

The factory set positions and functions of the two 10-pole dip switches S1 and S2 are as follows:

Pole	S2	S1
	1 2 3 4 5 6 7 8 9 10	1 2 3 4 5 6 7 8 9 10
Closed, ON, 1:	. . . . .	. O . . . . .
Open, OFF, O:	O O O O O O O O O O	O . O O O O O O O O
Function:	X * F E D C B A 9 8	7 6 5 4 3 2 1 O X X

X: Reserved
*: 8/16 bit I/O Addressing (ON-16, OFF-8)
O-F: I/O Address, Positive Logic (On-1, OFF-O)

The I/O Base Adress of the disk controller is 100 octal.

*: Factory settings

S2	Reserved
	S2-1
S2	8/16 Bit I/O addressing select
	S2-2
	ON 16 Bit I/O Addressing
	OFF 8 Bit I/O Addressing
S2	Most significant I/O address byte (positive logic)

S2-3,4,5,6,7,8,9,10

ON (1) S2-3 is the MSB and S2-10 is the LSB

OFF (0)

S1 Least significant I/O address byte (positive logic)

S1-1,2,3,4,5,6,7,8

ON (1) S1-1 is the MSB and S1-8 is the LSB

OFF (0) (S1-7 and S1-8 should be OFF)

S1 Reserved

S1-9, 10

B.2 Jumpers

The SMD 2181 contains seventeen jumpers labeled "W1" through "W19".

W1 -5 Volt Source (Located below U5)

*C-1 P1-9 and P1-10 supply -5 volts from the bus.

C-2 The on-board regulator supplies -5 volts.

W4 Interrupt Level Select (Located below U72)

C-0,1...7 Selects the desired interrupt level (connect only one). Jumper C to 4.

*NO CONNECT Status-driven systems should leave all disconnected.

W5 -5 Volt Regulator Source (Located below U74)

C-1 P1-79 and P1-80 supply -12 volts to the regulator.

C-2 P1-77 and P1-78 supply -10 volts to the regulator.

*NO CONNECT If W-1, C-1 is installed, W5 should be disconnected.

W6 Sector Detect + (Located between U3 & U4)

*C-1 "SECTOR" flags the beginning of a sector

C-2 "ADDRESS MARK FOUND" flags the beginning of a sector

W7 Sector Detect - (Located between U3 & U4)

*C-1 "SECTOR" flags the beginning of a sector

C-2 "ADDRESS MARK FOUND" flags the beginning of a sector

W8 Unit B3 (Located below U3)

C-1 SMD drive is unit 8 through unit 15

*C-2 SMD drive is unit 0 through unit 7

W9 Unit B2 (Located below U3)

C-1 SMD drive is unit 4 through unit 7 or 12 through 15.

*C-2 SMD drive is unit 0 through unit 3 or 8 through 11.

W10 Priority Out (Located below U35)

*C-1 SMD 2181 supplies BPRO active low (normal bus)

C-2 SMD 2181 supplies BPRO active high

W11 BCLK/Origin (Located below U61)

1-2 SMD 2181 supplies a 10 MHz BCLK/ to the bus

*NO CONNECT another board supplies BCLK/ to the bus

W12 Reserved

W13 Buffer Page 2 (Located above U31)

*C-1 2K to 4K buffer, 24 pin RAMS in U31 and U45

C-2 16K buffer, 28 pin RAMS in U31 and U45

W14 Buffer size (Located above U30)

1-2 4K or greater buffer, 2K or greater RAMS in U31 and U45

*NO CONNECT 2K buffer, 1K RAMS installed in U31 and U45

W15 I/O Address 0 decode (Located below U63)

*C-1 SMD 2181 Normal Mode

C-2 Reserved

W16 I/O Address 1 decode (Located below U63)

*C-1 SMD 2181 Normal Mode

C-2 Reserved

W17 Processor RAM size (Located below U38)

1-2 2K RAM installed in U51

*NO CONNECT 1K RAM installed in U51

W18 EPROM size (Located above U29)

*C-1 4K EPROM 2732 installed in U38 (normal mode)

C-2 2K EPROM 2716 installed in U38

W19 Priority In (Located below U70)

C-1 SMD 2181 is not to be the highest priority device or this is a parallel priority system

*C-2 SMD 2181 is to be the highest priority device in a serial priority system.

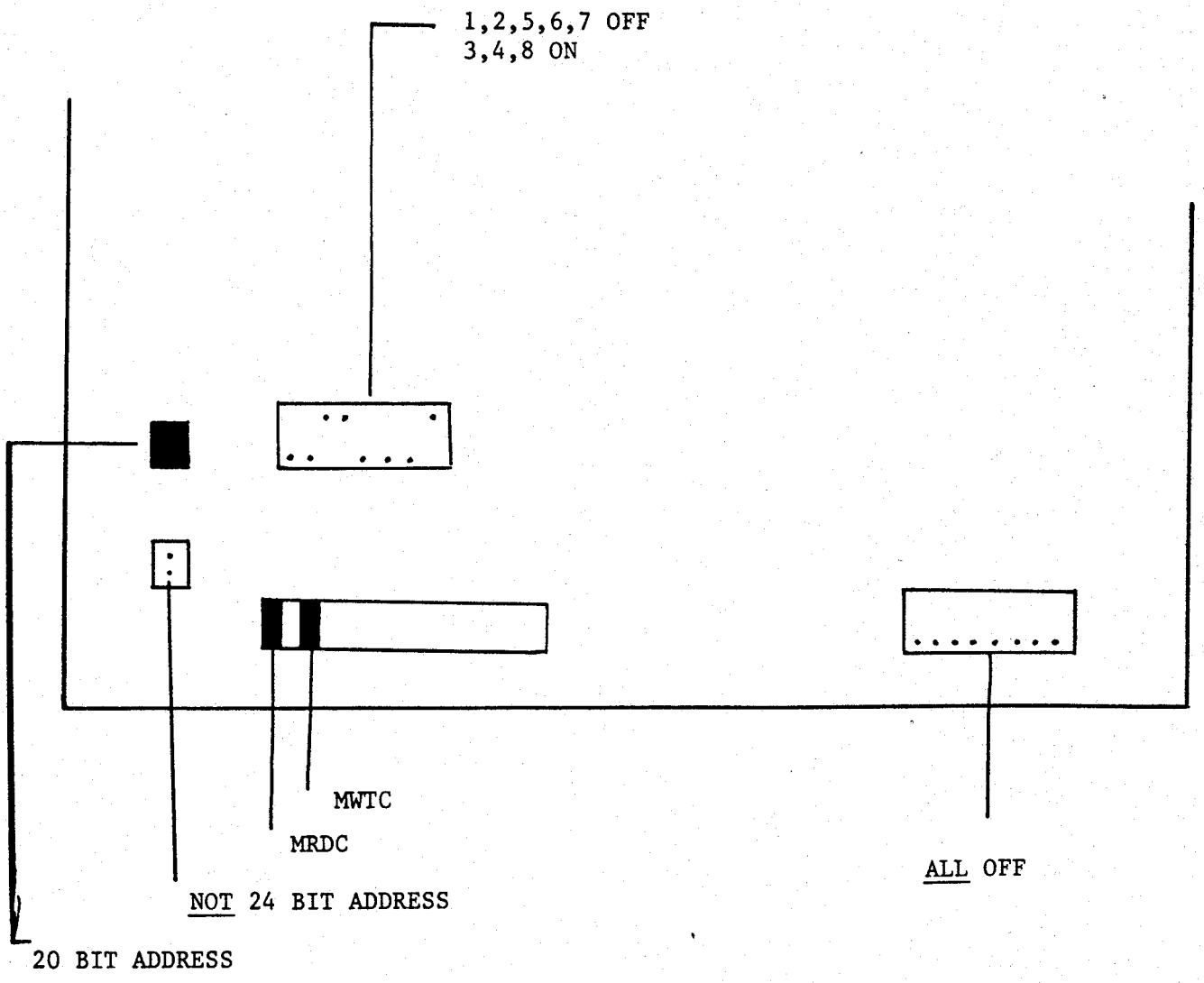
Appendix C

Tape Controller Board Dip Switches and Jumpers

Pole	S2	S1
	1 2 3 4 5 6 7 8 9 10	1 2 3 4 5 6 7 8 9 10
Closed, ON, 1:	O O O O O O O O O O	O . . O O O O O O O
Open, OFF, O:	. . . . .	. O O . . . . .

Appendix D

Jumpers and Switches for 3Com Ethernet Card



Appendix E

Selecting Disk Drive Operating Voltage

The voltage selection panel is labeled TRM1 and is located on the left hand side of the back of the disk drive (see Figure E-1). Change the taps on the terminal according to the desired voltage and frequency of AC input power as follows:

Vac (Hz)	Tap Number (A) - (B)	(C) - (D)
100 (50/60)	2 - 4	6 - 8
120 (60)	1 - 4	6 - 7
220 (50)	2 - 3	5 - 7
240 (50)	1 - 3	5 - 7

(For more information on the disk drive DC power supply unit, refer to the Fujitsu M2351A/AE Mini-Disk Drive CE Manual, pp. 3-1 to 3-3).

NOTE: If your exact voltage is not available, ALWAYS set the drive for the slightly higher voltage. For example, disk drives powered with 110 VAC, 115 VAC, or 120 VAC should all be set for 120 VAC--never for 100 VAC. This will protect the drive against voltage spikes and surges. If your drive is more than 10% from one of the drive's settings, use a transformer.

Appendix F

Selecting Tape Drive Operating Voltage

1. Make sure the transport power is OFF and the power disconnected.

2. Release the rack retaining mechanism located on the front panel at the lower left.

3. Hold the front panel firmly and withdraw the transport on its slides until the first lock engages.

4. Hold the front cover by its sides and raise it to its locked, retained position.

5. Release the first lock retainer on the right slide.

6. Withdraw the tape drive firmly on its slides until the second lock engages.

7. Use a screwdriver to loosen the two spring-loaded screws on the top of the top-plate casting.

8. Grasp the two lower corners of the front panel and lift the top plate to its maximum upright position. When the unit is lowered approximately one inch, the latch mechanism will engage automatically.

9. Insert a tool or bolt through the holes provided in both members of the top-plate support to prevent injury due to accidental release of the lock mechanism.

10. Releasing the Ny-Latch fasteners near the printed wiring board (PWB) stiffener, carefully lower the printed wiring board. Grasping it by the sides, lift it carefully out of the slots in the brackets at its rear corners and lower it it to the bottom of the chassis assembly.

11. Remove the two 6-32 Phillips' head screws that secure the power supply cover and slide the cover forward to remove it from the assembly. Note the position of the chassis ground cable.

12. Remove the four 6-32 Phillips' head screws at the corners of the power supply printed wiring board and lower the board.

13. The voltage selection card is on J9 on the power supply printed wiring board. Select the correct voltage by repositioning the key in J9 to correspond to the slot on the card as follows:

Voltage	Selection Card	Amperes	Frequency
85-110	100	3.0	50/60 Hz
102-132	120	3.0	50/60 Hz
187-242	220	1.5	50/60 Hz
204-264	240	1.5	50/60 Hz

14. Carefully replace the power supply printed wiring board, routing the cables so as not to interfere with the blower motor.

15. Replace the power supply cover and the Phillips' head screws and replace the printed wiring board in the slots of its brackets, making sure the chassis ground cable is connected to the power supply assembly.

16. Lift the transport to its maximum upright position and lower it smoothly to horizontal position.

17. Reverse Steps 1 through 5.

18. Make sure that all screws and retaining pins are replaced.

Appendix G

1Heath-Nu Machine Cable Configuration, RS232C

Nu Machine Male (DCE)			Heath Female (DTE)		
Wire	Signal	Pin	Pin	Signal	Wire
R	(TD)	2	2	(RD)	R
RB	(RD)	3	3	(TD)	RB
W	(RTS)	4	4	(RTS)	W
WB	(CTS)	5	5	(CTS)	WB
G	(SG)	7	7	(SG)	G
GB	(DTR)	20	20	(DTR)	GB

SEE ATTACHED FIGURE.

Appendix H

System Diagnostic Unit RS232 Port Pin and Signal Assignment Table

25-Pin, D-Shell Male Connector (Remote Channel)		25-Pin, D-Shell Female Connector ((Local Channel)	
Pin No.	Signal Name	Pin No.	Signal Name
1	Chassis Gnd	1	Chassis Gnd
2	TXD	2	RXD
3	RXD	3	TXD
4	RTS	4	CTS
5	CTS	5	RTS
6	--	6	Pull-Up (+12V)
7	Logic Gnd	7	Logic Gnd
8	DSR	8	DTR
9	--	9	--
10	--	10	--
11	--	11	--
12	--	12	--
13	--	13	--
14	--	14	--
15	--	15	--
16	--	16	--
17	--	17	--
18	--	18	--
19	--	19	--
20	DTR	20	DSR
21	--	21	--
22	--	22	--
23	--	23	--
24	--	24	--
25	--	25	--

Appendix I

Using LISP with No Ethernet Board

Soon there will be a SDU program that can tell the LMI Lambda when it has an Ethernet board. Until then, you have to follow the procedure below:

Put a working Ethernet board in the system and boot it. Then type

```
(login "lispm" t)

(si:set-sys-host "lm")
```

Now remove the board from the system variables:

```
(ethernet:remove-ethernet-board)
```

then type

```
(print-disk-label)
```

to look for a free "lod" band. You should see one with a number between 1 and 5 that has "" in its comment field. (The comment field is at the end of the line.) Type

```
(disk-save N)
```

where N is the number of the "lod" band (e.g., disk-save 2 if "LOD2" is the free band you wish to use), to write out to the disk a band that doesn't need Ethernet. (disk-save N) will ask you to type a comment; type "no ether". If it says something like "1.96 June 13 no ether" is too long for the comment field, retype the comment as "1.96 no ether".

Now the machine will look as if it is cold-booting. Wait until it comes back, then type

```
(set-current-band N)
```

where N is the same as above. Now turn off the power, remove the Ethernet board, and try booting the system again.

If you run into any problems while doing this, call LMI.

I.1 Reinserting the Ethernet Board

To go back to using the Ethernet board, type in LISP

```
(ethernet:install-ethernet-board)
```

I.2 Preliminary Theory of Operations

A "monitor" is the actual screen on which communications appear. The entire monitor, keyboard and cable assembly together is called a "terminal".

A "monitor" is also a program that runs on a computer as the first thing you see. It provides a path via which you can access the operating systm and the disk.

The root image contains basics with which the UNIX and LISP systems can be built. It contains all SDU software except that stored in the 3 SDU proms and the SDU CMOS RAM. It contains the file "/unix", which is the UNIX operating system (the UNIX "kernel") itself. It contains the files necessary to set up the LISP environment that the LISP processor can begin to run its own code, instead of relying on the SDU. It contains the UNIX device driver nodes. It contains information on what terminal lines are enabled, who is allowed to login on the system, and how the UNIX disk is partitioned.

When you first power on the Lambda, a prompt appears on the SDU console. This can happen because there is information in the SDU proms and the CMOS RAM that lets the SDU know that there's a console out there. This information says that if the rotary switch is in position 1, the console is a normal resolution terminal at 9600 baud attached to SDU port A. If the switch were in position 0, the console would still be attached to SDU port A but it would be at 300 baud.

Switch position 0 is when the switch is completely open. So, if the switch is disconnected or if oneof the wires for a particular position is broken, the console will default to SDU port A at 300 baud. Also, since the CMOS RAM contains the information on how to read the switch, if the CMOS RAM is invalid (SETUP light on) the console again is SDU Port A at 300 baud.

The SDU proms contain the program that runs the SDU "monitor". PROM software is copied into SDU RAM at power-up, where it runs even before the disk is accessible. It provides access to the disk by providing commands such as "ls", "cat", "copy", and software that can run programs from the disk.

A new version of SDU Proms, "Monitor Version 8", will be released in the next month or so. This release will know how to use the high resolution monitor with our AI keyboard as a console. With this development, we will stop shipping Z29s on the system. The default (open) switch position will be interpreted as the high resolution monitor. This will be the case even if the CMOS RAM is invalid.

The SDU prom change that makes this possible is a new keyboard driver for the SDU. A "driver" is the software link between the operating system and the piece of hardware that needs to be "driven". Our keyboard sends different "keycodes" than a standard ASCII does; when you type an "A" on the standard ASCII keyboard, a different byte (eight bits of information) is sent to the computer than when you type an "A" on our AI keyboard.

The Lambda keyboard driver understands both keyboards. On initialization the ASCII keyboard sends an identifiable code to the computer, while our keyboard sends nothing. We conditionalized the keyboard driver so that if it receives the correct code, it assumes it is using the ASCII keyboard. If it does not receive the code, it assumes our AI keyboard is attached. From then on, it interprets keycodes it receives according to a table for each of the two keyboard types. Very much the same thing has been done for the SDU proms.

After the prompt appears, you can communicate with the SDU monitor program via the console. At this point, you type either "reset" and "enable" or "init". In most cases, "reset" and "enable" is sufficient. "Reset" resets the memories; "enable" separately enables both the Nubus and the Multibus. The command "init" does both, but first grounds the reset line on the busses. This is a hardware reset, which is more thorough than that done by typing "reset". The only differences the user will see is that "init" is less to type, but takes longer and causes the SETUP and ATTN lights on the front panel to light (like the RESET button does). When software problems wedge boards, "init" helps but "reset" and "enable" are ineffective. It is always good practice to type "init" and then try again before giving up.

In extreme cases, even "init" isn't good enough, and you must press the hardware RESET button. This button grounds the reset line but does not enable the busses. Its main advantage is that it does not depend on a working console to be effective, so you can reset the machine and start over even if the SDU monitor crashes.

Several commands reside in the SDU proms, and so can be executed even without the disk. These include "init", "reset", "enable", and "help". The "help" command lists the SDU resident commands. The SDU proms also contain enough information so that an SDU with an invalid CMOS RAM can reload the CMOS RAM from tape. The CMOS RAM tells the SDU how the disk is partitioned, and so the SDU can't use the disk to run a program to reload an invalid CMOS RAM.

The "tape" driver from the SDU is in RAM (copied there from the PROM), and so works regardless of CMOS RAM state. If the CMOS RAM needs reloading, all you need in the machine is an SDU and a tape controller. With the 1/4" tape, all you'll need is an SDU, since the 1/4" tape controller is built into the SDU.

The "/tar" at the beginning of a command tells the SDU to use the "tape" driver to load the first file from the tape (the "bootstrap"), to name that file the "tar" driver, and then to run the program specified after "/tar". The SDU "tape" driver can do nothing except access the first file from the tape; the "tar" driver, loaded from tape, must do the rest of the work. For instance, "/tar/setup" runs the bootstrap from the beginning of the LMI diagnostic tape (the bootstrap works the same for all tapes, and for all monitor versions so far), then runs the "setup" program.

Because of the bootstrap program needed at the beginning of a tape to be run on the SDU, a standard UNIX "tar" format tape ("tar tape") cannot work on the SDU. Running a program in UNIX from the SDU would require putting the bootstrap program on the beginning of the tape.

Suppose you want to make a tape of an SDU program called "load" that can be run from the SDU on tape instead of disk. You must get a copy of the bootstrap program which you will then be able to use in all such cases.

Mount the LMI diagnostic tape. Copy the bootstrap from the beginning of the tape into the disk file "/bootstrap" by typing

```
dd if=/dev/rmt0 of=/bootstrap bs=2b
```

To write a tape of "load" to run at the SDU, mount a tape with a write ring, change into the directory containing "load", then type

```
dd if=/bootstrap of=/dev/rmt4 bs=2b

tar cvfb /dev/rmt0 2 load
```

This writes the /bootstrap file onto a tape in 1KB blocks, and will not rewind the tape when done (the tape device /dev/rmt4 does not rewind when done). Then, it will write the load program from the current directory onto the tape in 1KB blocks, and will rewind when done (the tape device /dev/rmt0 does rewind when done). When

the tape device closes, two EOFs are written, then the tape backs up over the second one. If you rewind now, there are two EOFs at the end, as required to mark the end of the tape. If this is the non-rewinding tape device and you write another file onto the tape, the second EOF is written over and one EOF is left between files.

The "dd" command has the arguments "if" (input file), "of" (output file), and "bs" (block size). "if=foo" means the input file (from which the blocks will be read) is the file "foo". "of=bar" means the output file (the file to which blocks will be written) is the file "bar". If "bar" is replaced by the name of the tape drive (e.g. "dev/rmt4"), blocks from the input file "foo" will be written onto the tape.

The "bs" option tells "dd" what size blocks are in the input file. Data should be written onto the tape in 1024-byte blocks. There are two ways to specify this. One is "bs-2b", where "b" is "2" is a multiplier (yielding 2*512 = 1024). The other is "bs"1k, which is more straightforward. Each works equally well on our system, and either can be used.

The "tar" options "c" (create, or write) "f" (file), "v" (verbose error messages) and "b" (block size) are specified on the command line. Since the "c" option is being used, the "f" file is taken as the output file, the file to be written on, the order of "f" and "b" are very important with respect to the rest of the command line. With the "tar" options in this order ("f" before "b"), first thing after the options is taken as the output file. The second thing is therefore taken as the blocksize. (If "b" was typed before "f", the first argument after the options would be interpreted as the blocksize and the second as the output file. Blocksizes are specified as multiples of the basic UNIX block (512 bytes).

A "block" in our UNIX is 512 bytes. Our disk, though is set for 1KB (1KB = 1 Kilobyte = 1024 bytes) blocks. This means that everytime a program requests a block from the disk, it receives a chunk of 1K bytes. Data can be written to the disk in multiples of 1K.

In UNIX, devices such as the disk (or logical sections of the disk), tape drive and terminals are all treatd as files. To see a list of all UNIX devices on our machine, you can list the /dev directory in UNIX. Some of the files there correspond to hardware that may not be installed on your machine. If you open one of those (e.g. using the "cat" command), it is likely to crash UNIX.

The null device, "/dev/null", is used for skipping over files. For instance, if you want to skip over the first file on a tape and use the second file, copy from the first file into /dev/null using the non-rewinding tape device. You will then be positioned at the second file on the tape.

The SDU can run C programs, but not exactly the same programs as the 68000 CPU can run. The chip running programs on the SDU is an Intel 8088, whereas the CPU uses a Motorola 68000 or 68010. Programs must be compiled differently for the 8088 than for the 68000, and so we have two different C compilers for these two different purposes. Programs in the "/sdu" directory and many in the "/" directory in the UNIX filesystem can be run from the SDU. Programs that can run on either board can do so because they have a "switch" telling them to use the version compiled for the 8088 if being called by the SDU, and the version compiled for the 68000 if being called by the CPU. The final, 68000- or 8088-compiled versions of such a "switched" program still need a different name for each processor's version.

The LMI diagnostic tape was written with the above techniques. A bootstrap file is followed by tar format files. The programs are compiled for the 8088 so that they can run on the SDU. Therefore, "/tar/setup clear eagle <or micro169> sp shell" can run on the SDU and write into the CMOS RAM the necessary information to setup the CMOS RAM for use with an Eagle <or Micro169> disk drive.

After this program is run, you must type "init" (or press the RESET button) before the new CMOS RAM setup is used. This is because the SDU doesn't look at the new information in the CMOS RAM until it runs its initialization code. Only after you do this will the red SETUP light go off and the green run light come back on. On the monitor version 7 proms, the console will now revert to SDU port A at 9600 baud.

A root image tape is created by typing at the SDU

```
copy $disk tape 8500
```

The SDU must be SETUP in order to do this, since although the "copy" command is in the SDU proms, an SDU with an invalid CMOS RAM does not know how to access the device "$disk" (the raw disk). The above command line copies FROM the first argument ($disk) TO the second argument (tape). It copies the number of blocks specified (8500), starting at the beginning of each of the specified devices. It has been determined both mathematically and experimentally that 8500 blocks is sufficient to contain the entire root image.

The "copy" program does not interpret what it reads as separate files. It simply copies the first 8500 blocks, whatever they contain, onto the tape. At the end of the copy, two EOFs are written on the tape. This is always done just before the tape device is closed after a write. Nothing prints until the copy is complete (or an error occurs), at which time a message will print saying how many bytes were copied. If all 8500 blocks were copied, the message will be "8704000 bytes copied". This is because:

8500 blocks * 1024 bytes/block = 8704000 bytes

This root image can be installed by typing at the SDU

copy tape $disk

the copy program copies this image from the tape onto the first 8500 blocks of the disk. It stops copying when it reaches the EOF mark at the end of the image. This is actually two EOF marks in a row, since there is already one at the end of each file, plus one at the end written by the tape driver just before closing the tape device.

Once the root is installed, many programs can be run from the disk. At this point, the UNIX kernel is on the disk, and can be run as long as memory and other boards are in the right slots. All the programs needed to set up the machine the rest of the way for LISP and UNIX are now on the disk.

The disk is logically separated (partitioned) for the SDU, LISP and UNIX. The program that allows the SDU to read the UNIX filesystem is in the first 22 blocks on the disk. Block 22 is called the "minilabel". This "minilabel" holds the offset of the LISP disk label. The LISP disk label (referred to merely as the "disk label") is just beyond the end of the root image (past block 8500), at cylinder 100. It is the first thing in "LISP territory", that section of the disk logically belonging to LISP instead of UNIX. LMI has given UNIX the first 100 cylinders of the disk to the SDU and to UNIX for the root image and the UNIX "usr" (user) filesystem.

The first time LISP software has to be put on the disk, it must be done using either the UNIX or the SDU (since the first choice, LISP, doesn't exist yet). Currently, the SDU software cannot install a LISP microcode and band tape. That leaves UNIX as the operating system of choice for initially installing the LISP microcode and band.

The UNIX program "load68", in the /lambda directory, can write a minilabel or a disk label, print it on the screen, set the current microcode and band, and install a new microcode and band from tape. It contains information on the disk partitioning for the eagle, micro169 and T302 disk drive, and can access both the 1/2" and 1/4" tape devices. It is completely oblivious to the part of the disk inhabited by UNIX, except for reading and writing the minilabel in block 22. The disk label it writes is in LISP disk label format, which means the product of the load68 program is a disk label that can be accessed by LISP just as if the LISP machine had written the label itself.

"load68" is the SDU "load" program, but compiled for the 68000 instead of the 8088. Everything you can do with "load68" in UNIX can be done with "load" from the SDU, except for loading a

microcode and band from tape. This cannot be done from the SDU because of limitations in the SDU tape driver, not because of the "load" program. We are working on rewriting this SDU program so that it can load these tapes, but this is not done yet.

The SDU limitation is that the SDU can read tapes only with 1KB blocks, while UNIX can read blocks that are multiples of 1KB. The LISP band is very big, and is written in 4KB (4096 byte) blocks. It does not fit on one 2400 foot (big) tape if it is written in 1KB blocks. UNIX can read 4K blocks fine, but the SDU cannot at present.

Before LISP and UNIX can be booted, they must know where to access the boards in the machine. Each board inn a slot has a certain range of addresses associated with it. NuBus slot 13 is a range of addresses that differ from those in either NuBus slot 14 or Multibus slot 13. In the previous release, memory board locations were set in UNIX by the file "/disk/monitor/conf/std" and in LISP with the "memsetup" command. Both are documented in Chapter 6.

Also in the old release, the RG board always had to be in slot 0 (although the order of the other LISP boards didn't matter), the VCMEM had to be in slot 8 (for use by either LISP or UNIX), and the 68000 CPU in slot 11. UNIX memory location was set by editing the "std" file; for LISP, the "memsetup" command wrote the selected memory board slots into a "memories" file which was accessible to both the user (to read when checking memory configuration) and the "lboot" program (to know the locations of the boards it must initialize).

In the new release, much of this is taken over by the "config" program. The 68000 CPU still must be in slot 11, since this location is compiled into the UNIX kernel. The VCMEM is no longer controlled by UNIX; and since LISP is more flexible about board slots, as long as LISP is controlling the VCMEM board its NuBus-only slot is unimportant. The RG board can be in any appropriate NuBus slot (not necessarially 0), because the "config" program and the LISP microcode now support a flexible RG board location.

The last thing that must be done before booting the LISP processor is to set the LISP processor speed. In the old release, this was accomplished using the "setspeed" program which is very similar in construction to the "memsetup" program. In the new release, this is done by the "config" program. Setting the processor speed really tells the LISP processor whether to use only one standard length source cycle and one standard length execute cycle (1-1), or to lengthen the time per source and execute cycle by using multiples of these standard units (2-1 and 3-3, for example).

Once the root image is installed, the disk label is initialized, the microcode and band are installed, and the machine is configured (with speed and board location), everything necessary

for LISP to boot is in place. When you type "lboot" from the SDU, the boot sequence begins.

In the old release, you would either type "uboot" to boot UNIX or "lboot" to boot LISP. Typing "uboot" would look in /monitor/conf/std" for the board locations and UNIX disk partitions, and would use the program "/monitor/bin/uboot" to boot the UNIX version in the file "/unix".

Also in the old release, typing "lboot" would run the program "/monitor/bin/lboot", which in turn would use the file "boot.dribble" in the /lambda directory. "lboot" would check the "speed" and "memories" files, then the dribble file script would initialize the NuBus memory to enable LISP to boot.

In the new release, the dribble file has been replaced by a compact C program that simply initializes the NuBus memory to contain the same values as if booted with the dribble file. The C program runs much faster than the dribble file used to. The result is that the SDU's role in the boot process takes one minute instead of five.

The bootstrap microcode is the LISP microcode that must run before the LISP processor can manage to run its own microcode., It is in a file in the /lambda directory, "c.sdu-uload" (in the old release, this was clled "sdu-uload"). This file is analogous to a prom that contained the bootstrap microcode on the CADRs. Because of this, the old version of lboot said "Loading prom" when this software was being loaded, and "waiting for the Lambda to halt" while it was running. The new release says "running bootstrap microcode" while it is running.

The "prom" or bootstrap microcode, loads the LISP microcode from the selected "lmc" partition into Lambda control memory. When it finishes, the microcode starts running. The LISP microcode initializes the on-board memory of the LISP processor. At this point, run bars appear on the high resolution screen. These bars show that the microcode is running, and is copying the band from the selected "lod" partition into the "page" partition in the LISP world. When the run bars mov to the bottom center of the screen, the band is actually starting to run.

Besides the boot program itself, the root contains the timing ram files, the bootstrap microcode and the csm (cache state machine) files for the LISP machine. So, when a root image is replaced, the new one may initialize the Lambda processor differently because of any one of these.

When UNIX boots, it comes up in "single-user" mode. This means only the console is enabled, and anyone who types on the console is the "superuser", or privileged user. The superuser has or can get access to any UNIX file, and can change, delete, or change the access permissions of every file.

At this point, only part of the system is running. In the old release, "uboot" had now initialized the NuBus memories in the "std" file and VCMEM board, started an "iomsg" (input-output messages for the CPU) process, enabled the console (/dev/tty0), and mounted the root filesystem (mounting the files in "/" on /dev/dk0a). uboot then continued running on the sdu, to keep the iomsg process running.

In the new release, "uboot" initializes the NuBus memory it is given by the "config" program, disregarding what the "std" file says. It leaves the VCMEM board for LISP to initialize. Instead of uboot running "iomsg", a background process starts on the sdu just before the boot procedure runs this and uboot exits when it finishes booting UNIX. Once uboot exits, control returns to the sdu.

When you type CTRL-D to bring up multiuser UNIX, several things happen. The shell script i.e., program in "/etc/rc" runs(made up of commands that could be typed directly at the terminal and run by the UNIX shell, rather than being written in C and needing to be compiled). It mounts the other filesystems listed in the script command lines starting "/etc/mount". In the old release, the "usr" filesystem was the only other one mounted (on /dev/dk0e) because all the rest of the files were in the root.

In the new release, the root has been broken into two parts. One (/sdu) consists of files run mainly from the SDU (files extending the SDU monitor) and not concerned with the running of UNIX, while the other is made up of files needed to run UNIX.

In the new release, the "rc" file mounts both the /sdu filesystesm (on /dev/dk0g) and the usr filesystem (on /dev/dk0e). Then, the file system consistency check (/etc/fsck) program looks in the file /etc/checklist to see whcih filesystems to check, and then checks them. When this check is finished, all the terminal lines beginning with "1" in the /etc/ttys file are enabled, and a UNIX login process ("getty") is started on each one. Lines that are disabled (/etc/ttys entries starting with "0") can be used as lineprinters, since all a getty does is cause a "login" prompt and start a shell running on the terminal. The "shell" is simply the process to which a user types -- it is the thing that prompts the user for input, and then causes typed commands to be executed as necessary by the CPU.

The "disk devices" mentioned above refer to the logically separate sections of the disk. It is still one disk, but broken into separate "minor" devices. The raw disk itself is a major device, and is actually major device #0. The root filesystem is major device #1 (since it is part of the "cooked", or block device, disk), and minor device #0. Other disk devices have other minor device numbers (for instance, the usr filesystem gets mounted on /dev/dk0e, which is minor disk device #5). The tape drive device we usually use is major device 13, with different minor device

numbers depending on whether the device rewinds (minor device #0, /dev/rmt0) or does not rewind (minor device #4, /dev/rmt4). Each major and minor devide specification and the associated device name is the /dev/ directory, and can be seen by typing "ls -l /dev" in UNIX.

One other file contains disk dependent information. This is the file "etc/fstab". It is used by "df", a program that prints out how much of each filesystem is free ("df" = "disk free"). This command is useful when you want to make sure your filesystem is not getting too full. This sometimes becomes a problem if users keep many copies of very large files around -- ten versions of ten "/unix" files saved on the root with different names, for instance. In such cases, a system hacker deletes the useless copies until there is sufficient space.

The sdu/monitor/conf/std file, mentioned in the discussion about memory boards earlier, also contains disk partition information. The rightmost column in the "disk" entry lines in this file contains the offset (in hexadecimal) of the disk partitions (logical subsections, or disk minor devices, if you prefer). The next column to the left contains the length in bytes (in hexadecimal) of the partitions.

The /sdu/monitor/conf/std file from the old release is in Chapter 6. The only difference between that and the one for the new release is that we have increased the size of the usr file system. It used to be 16E3000 hex bytes long (that's 23436 decimal blocks), and it has been increased to 2710000 hex bytes (40000 decimal blocks) in this release.

Although the std file has the hex byte lengths of the disk partitions, the decimal block members are used by the "/etc/mkfs" program when making the filesystems. The number "23436" for the usr filesystem is familiar, since it was used in the old relese whenever the systesm installer created the usr filesystem. The usr filesystem fills the space between the end of the root and the beginning of LISP (the LISP disk label, actually), and so is not contained in the root image tape. In the new release, there is a shell script called "makedisk" that makes the appropriate filesystems. This eliminates the chance of human error while making the filesystems, besides cutting down on the amount of typing the installer must do. To see what "makedisk" does, you can look at the shell script by typing "cat makedisk" while you are in the root (/) directory in UNIX.

Recently, we began to support the micro169 169MB disk drive. Below are the calculations done to determine the appropriate "std" file and "makedisk" filesystem numbers to support that disk. The precise format of the std file would be the same as for the Eagle drive.

Many of the files mentioned above can be read by a user to get a better understanding of the system. All shell scripts and some other files are in human-readable form, and can be viewed with the "cat" command. If the file is too big to fit on the screen, type CTRL-S to stop it from scrolling. Type CTRL-Q to allow it to proceed. The files /etc/ttys, /etc/checklist, /etc/fstab, /etc/rc, /makedisk, and /sdu/monitor/conf/std are all readable; /etc/rc and /makedisk are both shell scripts.

The old and new releases use different setup tapes for the SDU's CMOS RAM. There are a few factors determining what setup tape is needed for an SDU in a system. The first factor is the monitor version in the SDU proms. Currently, monitor version 7 is in use. Soon, that will change to monitor version 8 and we will need a new setup tape.

Another factor is the root image. More precisely, it is the SDU's UNIX file system driver resident on disk (in the first 22 blocks). The CMOS RAM determines what gets loaded from the disk monitor into SDU RAM (onboard memory). If the SDU is setup with the old release diagnostic tape but the new root is being used, the error "errno 6, can't run_____" will print for anything typed that uses the disk (in other words, for any commands that are not resident in SDU prom). Setting up the CMOS RAM with the new setup tape fixes this.

Another change between the old and new releases is the name of the rewinding and non-rewinding tape devices. We always use the "raw" tape devices, which deal in character-by-character transfers instead of waiting for block transfers. In the old release, the rewinding raw tape device was called "dev/rmt1". In the new one it is called "/dev/rmt0". The old non-rewinding tape device was "/dev/rmt", while the new one is "/dev/rmt4".

In the new release, when the system is booted, we use either "lboot", "uboot", or "superboot". Only "superboot" starts by running "starter", a program that starts running the sharing disk driver (capable of letting both UNIX and LISP use the disk, with the SDU arbitrating) and starts the UNIX "iomsg" background process running. Then, "lboot" checks the "config" programs shared array (the file containing the LISP and UNIX configuration info), and boots LISP in the manner previously explained.

"uboot" boots UNIX as described previously. The "superboot" program runs "lboot" and then "uboot". "lboot" must run before UNIX so that it can initialize its NuBus memory before UNIX initializes its. For some reason, the LISP memories do not get initialized correctly if UNIX goes first.

One interesting and unforeseen development with the new releas is the many ways it is incompatible with older LISP board versions. The newest LISP software (greater than or equal to microcode 127 and the associated band), with the new root image (containing new

timing ram files, csm files, prom files (bootstrap microcode) and boot sequence), is correct for the newest PC boards. These PC boards cannot work with any older microcode (e.g. 94, which was used nicely for a few months). Very old wirewrap boards (V2.n) cannot run with these newer microcodes, and cannot even run on the old microcodes with the new root. They can, however, run the old-fashioned way, with the old root image (the 12/5/83 Root, standard until shipments at the end of May), booted on the pre-Streams software, (microcode 94, for instance). These incompatibilities arise from boot-time and microcode differences, not from the LISP bands. The bands contain higher level software, and run on all boards.

Certain bands are made to run with certain microcodes. If they do not go together, the band does not have the error table for that microcode saved into it; consequently, it searches over the network for the machine that it expects to have the error table. Needless to say, this makes the machines useless until they can be loaded with a consistent band and microcode.

Appendix J

Streams Release I

This is a summary of the differences between this release and the pre-Streams release. This document is meant for in-house use, and as a reference for interested customers.

This release is based on TI Distribution 3.1 software. The diagnostic tape, root image and UNIX usr files are all affected by these changes. Whether or not a machine is usiong STREAMS, the Release 1 software is the standard and the documentation below is pertinent.

J.1 The LMI Diagnostic Tape

The diagnostic tape from TI contains the setup program for the SDU CMOS RAM, plus diagnostics to be run on the NuMachine boards. The "lam" LISP board diagnostic program was added to the tape, and another setup program ("usetup") was added. "usetup" will not be used in general practice, and is on the tape for future compatibility.

An SDU setup with the pre-Streams setup (on the old diagnostic tape) will not run the new (Streams) root image. The converse is also true.

J.2 The Root Image

The new root image is somewhat of a departure from TI's root. The SDU filesystem and the UNIX root have been combined into one root image tape. These changes are user-transparent at the install; we still say "copy $disk tape 8500" to write out the root image, and "copy tape $disk" to install it.

In the old release, the prefix "/disk" was used with any file on the root ("/") to see that file from the SDU. Now, typing "/disk"

from the SDU gives access to files in the "/sdu" directory, which is the UNIX name for the monitor. To see the files in "/" (on the UNIX root), preface the filename with "/uroot" at the SDU.

The lengths and offsets of the UNIX logical disk partitions have been changed in this release, much as was done in the previous release. The root, sdu, swap space and raw disk are the same offsets and lengths, and the length of the "/usr" filesystem has been changed to fit within the first 100 cylinders on the disk. This usr filesystem is larger than it was in the old releas -- it is now 40000 blocks long instead of 23436 blocks (1KB blocks). As before, the filesystems disk_d, disk_f, and src have been set o offset and length zero.

The /sdu/monitor/conf/std file contains information on how he UNIX disk is partitioned. Below is the new /sdu/monitor/conf/std:

```
cpu:     FB000000
vcmem:   F8000000
ram:     F9000000      200
ram:     FA000000      200
port:    FF000014      FFFFDFFC
disk:    0       3     6C7800          FFC00
disk:    0       3     1FCC00          7C7400
disk:    0       3     19A28000              0
disk:    0       3     0                     0
disk:    0       3     2710000         9C4000
disk:    0       3     0                     0
disk:    0       3     FA000           5C00
disk:    0       3     0                     0
logport:                       0
logchannel:      1
```

Other files changed to reflect the "std" changes are /etc/fstab, /etc/checklist, /etc/rc and /makedisk.

The file /etc/ttys has been changed to disable ttyb so that it can be used as a lineprinter, and to enable lines for supdup connectins and for streams connections via LISP. If the user has an MTI board, terminal lines tty0-tty15 are enabled and the Systech MTI board is absent. The rsd (raster scan display, or high resolution monitor) entry has been disabled, leaving control of the VCMEM board to LISP instead of UNIX.

An LMI version of the UNIX kernel "(/sdu/unix.new") is now on the root. It is based on the TI Distribution 3.1 unix, but can support Streams. "uboot", the program that boots UNIX, has also been changed. Chaos support has been added to the UNIX kernel. The MTI driver has been changed to support a different interrupt level (7 instead of 8); this change was needed to support the use of the MTI in Streams.

When UNIX boots multiuser, the "/etc/rc" program copies "/sdu/unix.new" to "/unix" so that it can be found by programs needing access to the system symbol table. "ps" is an example of one such program.

"config", a system configuration program, now takes care of board allocation. LMI UNIX uses the results of this program to know which memory boards to initialize, instead of using the memory board information in the "std" file.

The SDU arbitrates disk access, and a "sharing disk driver" allows LISP and UNIX to run concurrently. This driver runs instead of TI's driver. TI has two drivers, one for /sdu (the monitor) and one for /uroot (UNIX root). The /uroot driver takes up too much space in memory to allow the sharing driver to load. Therefore, in this release the /uroot driver is not loaded at power-up.

Files from /uroot cannot be accessed unless the uroot driver is loaded. This can be done by causing an error message (e.g., by typing "foo" at the SDU). After that , the uroot driver is loaded until the next time "init" is typed or the RESET button is pressed, and files in /uroot can be accessed. The machine cannot be superbooted, however, because there is not enough room for the sharing driver. "init" must be typed before the "superboot" command will work.

In the previous release, all LMI files were added to the root in "/lambda". Now, files are added to "/sdu/lambda", and these files are copied to "/lambda". The sdu filesystem is much smaller than the root, and so the minimum amount of software is added to /sdu. /sdu/lambda is a subset of /lambda, and /sdu is only used for files that must be accessed from the sdu (without the uroot driver, since there is not room for it to load).

To clear even as much space as there is in /sdu, all but the "tape" and "quart" diagnostics have been deleted from the directory /sdu/monitor/diag. These diags remain because it would be largely useless to run tape diagnostics from the tape. Hence, the tape (1/2" and 1/4") diags should be run from disk, while the rest of the diagnostics must be run from the new LMI diagnostic tape.

/lambda contains more files than /sdu/lambda. It includes files that are not needed at the SDU but that must be found by the old boot program (lboot271). In addition to the new software, /lambda contains the obsolete programs necessary to make the old boot software work. As long as the machine is configured with an appropriate Lambda processor (not the newest PC boards, and not the oldest wirewrap V2.n boards), if necessary the machine can be booted machine non-Streams, with old software. (Use memsetup, setspeed, and boot with "lboot271". This is not guaranteed, because of innumerable differences in various hardware and software versions.) This relic will be removed in a future

release, as Lambda board versions become standardized.

Two other files added to the root are "/etc/ddate" and "/version". The first file is the repository for dump dates, and is written to by the "dump" program. The other is a file meant to be the repository for any other software version information users wish to include. This file is not used by any program, but if kept up to date by installers and users it will aid us in tracking user software.

The tape device /dev/rmt0 was linked to /dev/rmt1 by typing

```
ln /dev/rmt0 /dev/rmt1
```

for compatiblity with previous software, and similarly the tape device /dev/rmt4 was linked to /dev/rmt by typing

```
ln /dev/rmt4 /dev/rmt
```

The following lines were added to the file /etc/ttys

```
07tty0
07tty1
07tty2
07tty3
07tty4
07tty5
07tty6
07tty8
07tty9
07tty10
07tty11
07tty12
07tty13
07tty14
07tty15
```

Nodes were made for the MTI devices by typing

```
cd /dev
/etc/mknod tty0 c 11 0
/etc/mknod tty1 c 11 1
/etc/mknod tty2 c 11 2
/etc/mknod tty3 c 11 3
/etc/mknod tty4 c 11 4
/etc/mknod tty5 c 11 5
/etc/mknod tty6 c 11 6
/etc/mknod tty7 c 11 7
/etc/mknod tty8 c 11 8
/etc/mknod tty9 c 11 9
/etc/mknod tty10 c 11 10
/etc/mknod tty11 c 11 11
/etc/mknod tty12 c 11 12
```

```
/etc/mknod tty13 c 11 13
/etc/mknod tty14 c 11 14
/etc/mknod tty15 c 11 15
```

The /version file was edited to say

```
6/24/84 wer
LMI Streams Release 1 Root:
qboot v10
config v5
uboot v4
unix 3.133
mti lines in /dev and /etc/ttys
```

UNIX usr Files

The usr file tape is based on the TI Distribution 3.1 tape, but contains new chaos and lineprinter files. The "lmtar: program is also on this tape, in the directory "/usr/bin".

Hardware

The "INT 0" jumper on the 3Com ethernet board is now important. With the new software, the 3Com boards will not work without this jumper.

rnet board is now important. With the new software, the 3Com boards will not work without this jumper.

Nu Machine Installation and User Manual

TI-2242824-0001

November, 1983

Distributed by LMI 6033 W. Century Blvd. Los Angeles CA 90045
USA

WARNING: "This equipment generates and uses radio frequency energy and if not installed and used properly, i.e., in strict accordance with the instructions manual, may cause harmful interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment.

Operation of this equipment in a residential area is likely to cause interference in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference."

This manual contains the procedures for customer installation of peripherals including the 1/2" Cipher Tape Drive, the 474 MB Fujitsu Disk Drive, the 84 MB Fujitsu Disk Drive, and the Raster Scan Display.

1.1 Installation Checklist

This checklist aids in properly installing the Nu Machine System. Follow all the applicable actions listed. Refer to the sections listed for detailed information concerning completion of checklist items.

____1. Unpack the system (see Section 2).

____2. Carefully roll the Rack Module or Office Module cabinet to its permanent location. To keep the cabinet from moving and to level it, rotate the leveler feet at the cabinet base until they are planted firmly on the floor and the cabinet wheels no longer contact the floor. Check with a leveler bubble that the cabinet is level.

WARNING: Do not move the cabinet or the leveler feet at all once the system is powered up. If the cabinet is to be relocated, shut down the system and lock the disk heads before moving.

____3. If Rack System, remove rack module door prior to installing disk drive or tape drive:

- o Cut tie wraps holding front panel interface cable against cabinet side.
- o Using a Phillips #2 screwdriver remove the printed circuit board mounted to the floor door.
- o Using needle nose pliers, pull off metal clothespins on door hinge pins.

o Using a back and forth motion, grasp the top of door hinge pins and "wiggle" up out of door hinges.

____4. If Rack System, install Eagle Disk Drive (see Section 3).

____5. If Rack System, install Cipher Tape Drive (see Section 4).

____6. If Office System, unlock heads on Micro Disk Drive (see Section 5).

____7. Install Raster Scan Display (see Section 6).

____8. Install, if any, RS232 terminals/devices to SDU ports (see Section 6).

____9. Check that all cables are securely connected.

____10. Check that all connections to the AC distribution box are secure.

____11. Select the desired Mode Selector Switch position (see Section 7).

____12. Power up the system (see Section 8).

____13. Run all board and controller diagnostics (See Section 9).

____14. If there are problems installing the hardware or successfully completing the diagnostics refer to the trouble-shooting flow (see Section 10). If further problems exist, contact:

Texas Instruments, Incorporated
Customer Technical Services
17881 Cartwright Road
Irvine, California 92714
(714) 660-8217

TI MSG = GNU

^UNIX is a trademark of Bell Telephone Laboratories, Incorporated

2 RECEIVING AND UNPACKING INSTRUCTIONS

Before unpacking your system, visually inspect the shipping containers for damage. Check the condition of the Tip and Tell on the large shipping container with the crate bottom. Note any damage problems on the delivery receipt and contact the carrier for instructions on filing a claim. Have the driver sign the delivery receipt. Retain all packing and delivery receipts.

2.1 Required Tooling

Sharp knife
Diagonal cutters for cutting steel bands
9/16" socket

2.2 Unpacking Procedure

1. Obtain the packing list. The packing list is coded by number to each box so that its contents can be identified. As you unpack the system components, inventory the items received against the packing list. Visually inspect each item for shipping damage. If there is any significant damage, stop unpacking the system and contact the shipping agent. After the shipping agent inspects the damage, contact Texas Instruments, Irvine, California.

2. Save all packing material until the shipment inventory is completed. Consider saving the shipping containers and foam packing materials for possible reshipment or relocation of the system.

3. Unpack the Nu Machine manuals, diagnostics tape and operating system tape (if ordered). These items are normally inside the shipping container which has the packing list attached to it.

4. Unpack the monitor, keyboard and their cables, if ordered. These items are normally packed together. Remove top foam inserts to access items.

5. Locate the Fujitsu Eagle disk drive, if ordered. Do not attempt to remove the disk drive from its shipping containers at this point. Check that the Fujitsu disk card extractor tool (to remove cards in the disk drive) is taped to the back inside cover of the Fujitsu CE manual.

   NOTE: The Fujitsu CE manual and extractor tool are provided but should not be used without first contacting Texas Instruments, Customer Technical Services, Irvine, California.

6. Locate the Cipher Tape Drive, if ordered. Do not attempt to remove the tape drive from its shipping containers at this point.

7. Unpack the rack module cabinet, if ordered:

   - With safety glasses on, cut the steel bands around the shipping container with the diagonal cutters. Remove the bands and band guards.
   - Remove tape from top of box and open the top flaps.
   - Lift out the top wood pallet from inside the box. Keep this pallet close by--it will be used for a ramp shortly.
   - Remove styrofoam corners packed around top of cabinet.
   - Lift cardboard box off of cabinet.
   - Remove tape from around styrofoam posts and cabinet.
   - Remove front and back styrofoam posts.

     NOTE: For repacking purposes, notice that the front posts have a thin and a thick side while the back posts have two thick sides.

   - Lift the protective plastic bag off of the cabinet.
   - Use the 9/16-inch socket to remove the nuts on the 2 by 2 board attached to the forward baseboard. Lift the 2 by 2 board/forward baseboard unit off the bolts. On these same bolts, hook on the top wood

pallet turned upside down (so 2 by 6 boards are up) to form ramp.

- With one person behind and one in front of the cabinet, gently push the cabinet down the ramp onto the floor.

8. Unpack the office module, if ordered:

NOTE: The office module is shipped with the 1/4-inch Tape Streamer and the Microdisk Drive installed.

- With safety glasses on, cut the steel bands around the shipping container with the diagonal cutters. Remove the bands and band guards.
- Remove the tape from top of box and open the top flaps.
- Lift out the top wood pallet from inside the box. Keep this pallet close by--it will be used for a ramp shortly.
- Remove styrofoam corners packed around top of office module.
- Lift cardboard box off of the office module
- Remove tape from around styrofoam posts and office module.
- Remove styrofoam posts.
- Lift the protective plastic bag off of the office module.
- Use the 9/16-inch socket to remove the nuts on the 2 by 2 board attached to the forward baseboard. Lift the 2 by 2 board/forward baseboard unit off of the bolts. On these same bolts, hook on the top wood pallet turned upside down (so 2 by 6 boards are up) to form ramp.
- With one person behind and one in front of the office module, gently push the office module down the ramp onto the floor.

3 EAGLE DISK DRIVE INSTALLATION

This section presents instructions for customer installation of a 474 MB Fujitsu Eagle disk drive into a Nu Machine rack module. If the Nu Machine Operating System was not purchased, the disk is formatted. If the Nu Machine Operating System was purchased, it is already configured for the system, installed on the disk and ready for immediate use.

3.1 Required Tooling

Phillips head screwdriver, 8" or longer.

3.2 Installation Instructions

1. In the disk drive compartment of the rack cabinet (see Figure 3-1), use a large #2 Phillips head screwdriver to remove the brackets (slide guards) holding the inner slides in place. Pull out slides on both sides until the latch mechanism activates.

   NOTE: Two, possibly three people are required to lift the disk drive disk drive.

2. Remove disk drive from shipping containers. Slide hands in grooves on sides of foam bottom to grasp under disk drive.

3. Mount disk drive onto slides and push toward the rear of the cabinet. Check the slide mechanism by pulling the disk drive in and out of cabinet.

   CAUTION: Do not extend the disk drive and tape drive units from the rack module cabinet simultaneously.

   WARNING: Do not plug AC cord into AC box until the following steps are performed.

4. Holding the front panel firmly, pull out disk drive until the top cover of drive clears the cabinet.

5. Using a Phillips head screwdriver, remove the two screws on both sides of disk drive cover. Lift the cover off the base of disk drive.

6. Unlock "HEAD" of disk drive (see Figure 3-2). From the rear of the disk drive, looking over the interface PCB, loosen screw A until it clears Hole A in the Lock Plate. With the screwdriver, rotate the Locking Lever clockwise so that Screw A is now aligned over Hole B. Gently tighten screw A into Hole B. Gently tighten screw A into Hole B. (Reverse this procedure to relock the disk head.)

   WARNING: Live data on the disk may be damaged if it is not treated gently once the disk head is unlocked.

7. Replace disk drive cover (reverse step 5).

8. Go to the back of disk drive and loosen the right Phillips head screw of the Flat Cable Strain Relief (see items 3 & 4 in Figure 3-3). Lift up the strain relief and install "Disk B" cable (see item 1 in Figure 3-3) onto the interface board located in back of the unit.

9. Install the "Disk A" cable onto the interface board located in the back of the unit (see item 2 in Figure 3-3).

10. Re-install the flat cable strain relief.

11. Plug the AC power cord into the unswitched outlet on the extreme right interior side of the AC distribution box.

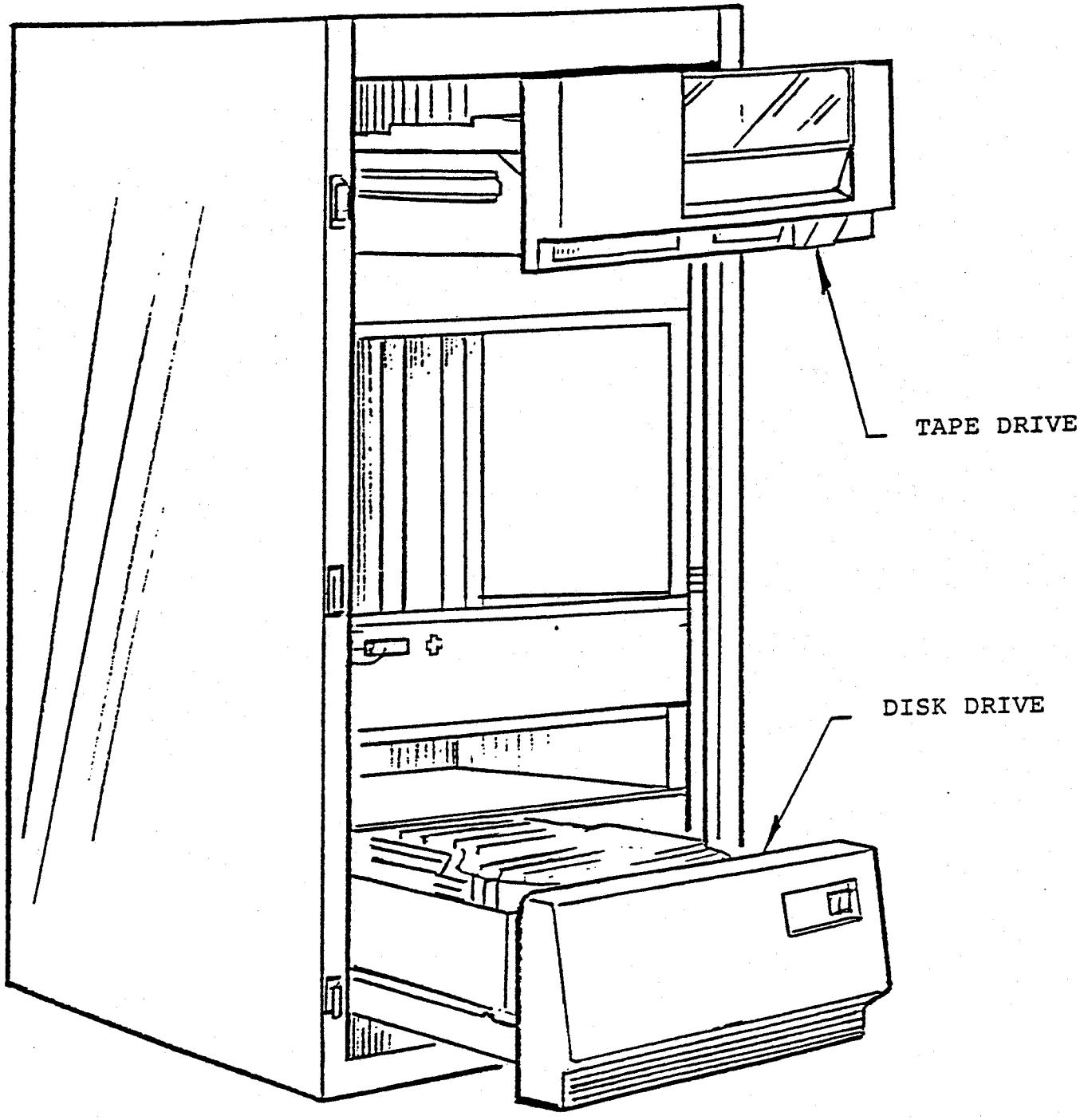


Figure 3-1. Tape Drive and Disk Drive Locations in Rack Module Cabinet.

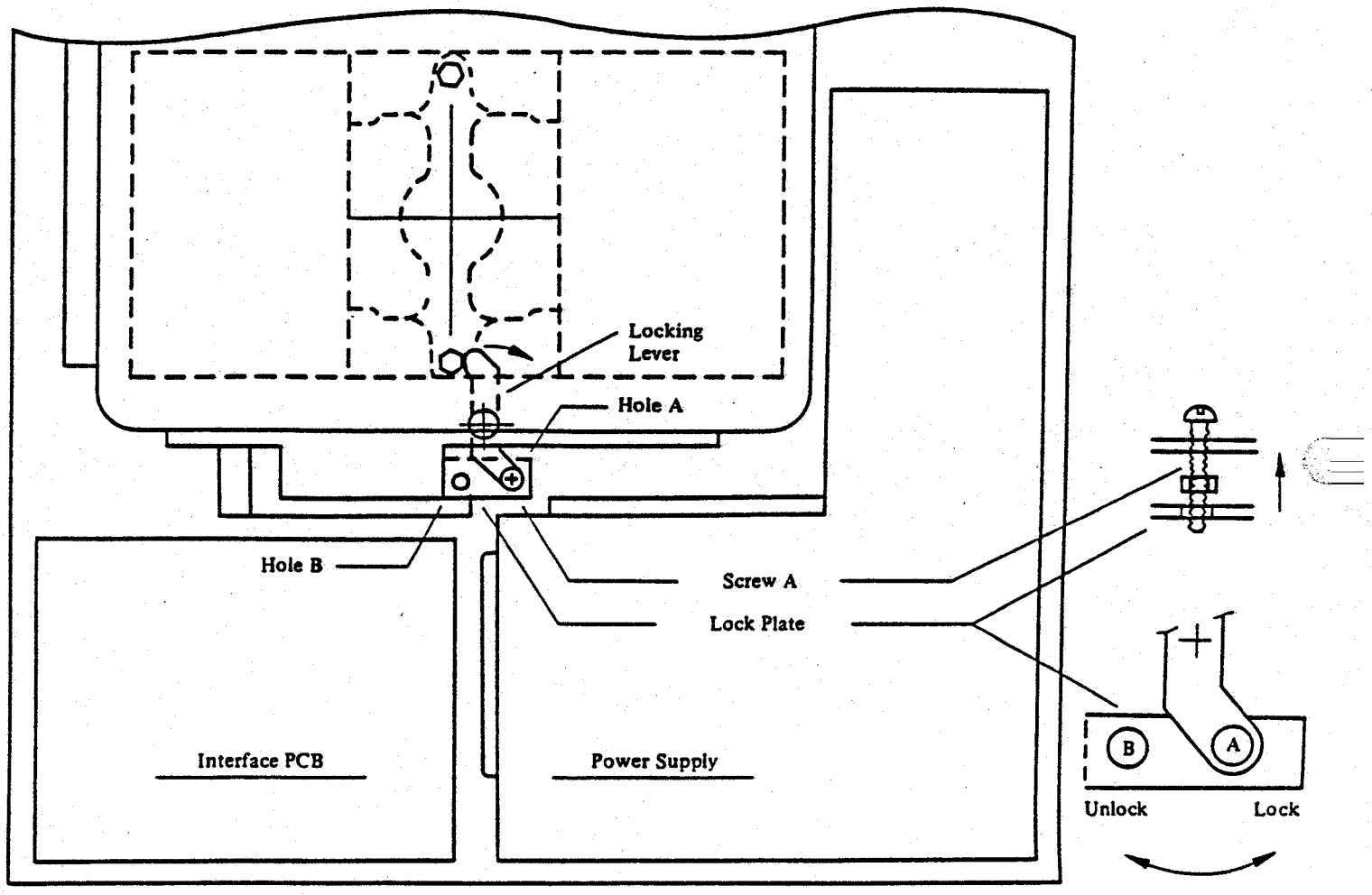


Figure 3-2. View Locking Down on Disk Drive from Rear for Unlocking 'Head'.

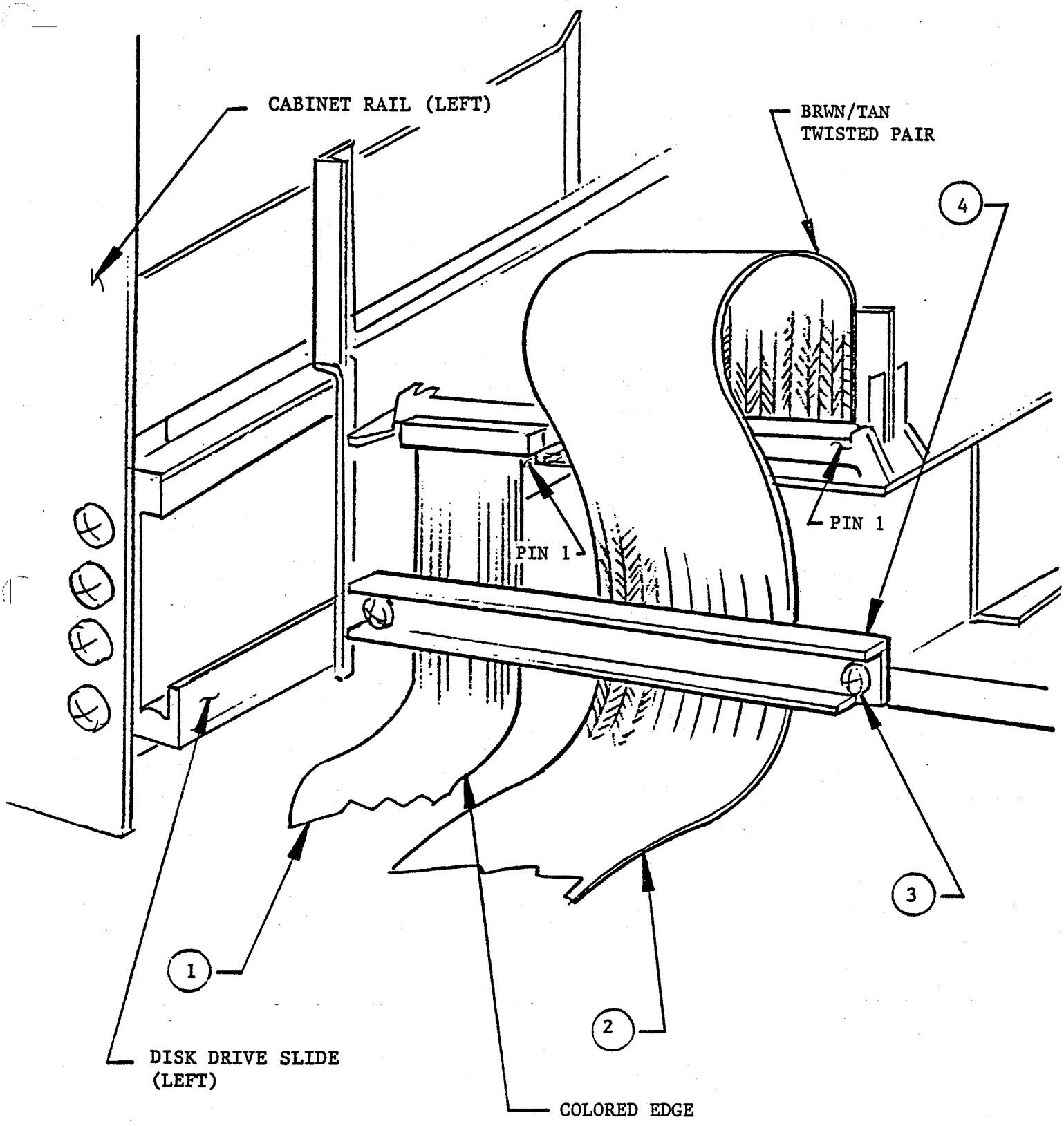


Figure 3-3. Installation of `Disk A' and `Disk B' Cables Onto Interface Board.

4 CIPHER TAPE DRIVE INSTALLATION

This section presents instructions for customer installation of 1/2" Cipher tape drive into a Nu Machine rack module.

4.1 Required Tooling

3/8" flat blade screwdriver

4.2 Installation Instructions

1. In the tape drive compartment of the rack cabinet (see Figure 3-1), use a large #2 Phillips head screwdriver to remove the brackets (side guards) holding the inner slides in place. Pull out slides on both sides until the latch mechanism activates.

2. Remove tape drive from shipping containers.

   NOTE: Two people are required to lift the tape drive.

3. Mount tape drive onto slides and push towards the rear of the cabinet. To slide the drive fully into the cabinet, simultaneously depress the slide locking buttons and push drive into cabinet. Check slide mechanism by pulling tape drive in and out of the cabinet. The tape drive should move freely.

   CAUTION: Do not extend the tape drive and disk drive from the rack module cabinet simultaneously.

   WARNING: Do not plug AC cord into AC box until the following steps are performed.

4. Holding the front panel firmly, pull out the tape drive until the cover of the tape drive clears the cabinet.

5. Pull up on both forward sides of the black cover and pull straight up. To hold cover up, swing white brace (mounted into left side of cover) down and fit it into the indentation in the top of the place casting.

6. Remove the Styrofoam material from the "Take Up Hub" (see item 1 in Figure 4-1).

7. Using a flat blade screwdriver, loosen the two spring-loaded screws at the side front top of the place casting (see item 2 in Figure 4-1). Lift the front panel by grasping the two lower corners. Lift the unit into its maximum upright position. The latch mechanism will automatically engage when the unit is lowered approximately one inch. Remove the foam packing material between the Printed Wire Board and the upper chassis (2 pieces: 1 block, 1 sheet).

8. Lift the unit slightly to unlock the latch mechanism and lower the unit until it seats on the bottom chassis. Using a flat blade screwdriver tighten the two spring-loaded screws (see item 2 in Figure 4-1). Raising the top cover slightly, push the white brace back and up into the cover and lower the cover into its closed position.

9. Slide the tape drive back into the cabinet.

10. Plug one end of the AC power cord into the tape drive back panel and the other end into the unswitched outlet on the extreme right interior side of the AC distribution box.

11. Holding the flat gray J1 cable (mounted in cabinet) with the red stripe to the right, connect the cable to the far left connection marked P1 on back of tape drive. Holding the flat gray J2 cable (mounted in cabinet) with the red stripe to the right, connect the cable to the P2 connection located between both P1 connections on back of tape drive.

NOTE: The red stripe of both cables represents Pin 1 of the connectors.

CAUTION: Do not slide the tape unit out with cables P1, P2 and AC cord attached.

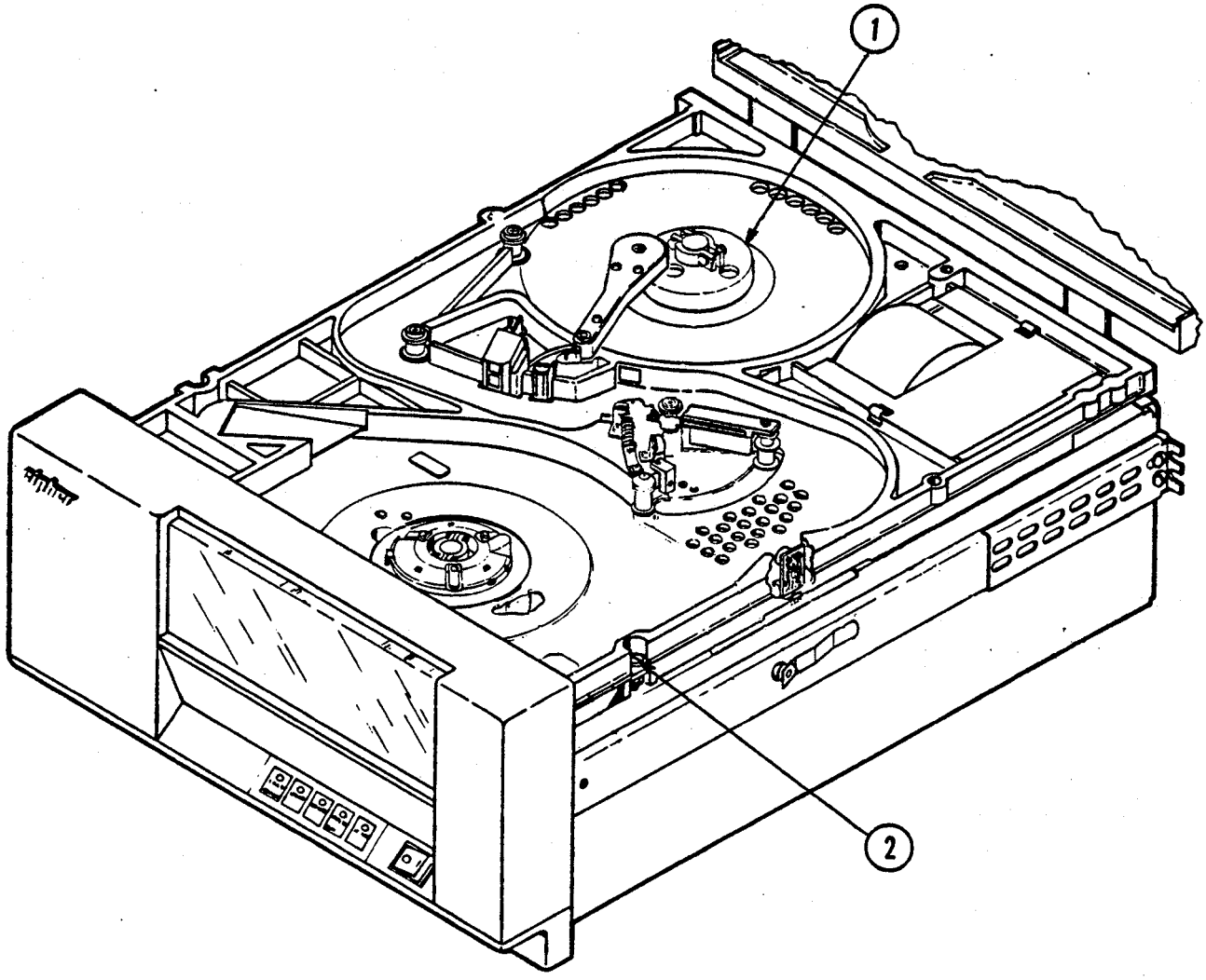


Figure 4-1. Cipher Tape Drive Unit.

5 MICRO DISK DRIVE PREPARATION

The Nu Machine Office System is shipped with the 84 MB Fujitsu Micro disk drive installed. The only procedure required is to unlock the disk heads.

5.1 Required Tooling

#2 Phillips head screwdriver.

5.2 Head Unlock Procedure

1. Remove upper rear panel of office system (8 screws).
2. Remove lower rear panel (4 screws).
3. Disconnect DC Power Disk Drive Cable, 2235194, from DC Peripheral Cable, 2235189. The cables are located at the rear of disk drive fan, pull connectors apart. Disconnect the fan AC connector immediately above the disk fan. (When reconnecting the fan AC connector, the flat keys on the plug go down.)
4. Unlock front door and swing open.
5. Remove (4) screws securing the disk drive to the peripheral chassis, in the lower right compartment.
6. Locate the flat cable clamp behind the card cage, mounted to the rear base of the cabinet. Insert screwdriver tip into indentation on clamp and pry up to free the disk cables.
7. Slide disk drive forward until the locking mechanism is visible on the lower right side of the disk drive (white "on"/"off" knob).

   NOTE: This operation requires two people, one to pull the drive out of the peripheral chassis and the other

to guide the ribbon cable properly from the back of the unit.

8. Use a Phillips head screwdriver to loosen the clamp holding the knob in place. Turn the knob to "off". Tighten the clamp with the screwdriver again. The heads of the disk drive are now unlocked.

9. Carefully guide the flat cables back into the original position as the disk is pushed into the peripheral chassis. This operation also requires two people.

10. Follow steps 1-5 in reverse to complete the procedure.

6 TERMINAL INSTALLATION

This section presents the instructions for customer installation of the Raster Scan Display and connection to the SDU RS232 serial ports.

6.1 Raster Scan Display

6.2 Required Tooling

1/8" flat blade screwdriver.

6.3 Installation Instructions

1. Attach the P1 connector (25 pin D-shell) of the Video Cable to the I/O plate of the back panel chassis with the single 25 pin female D-shell connector. See Figure 6-1.

2. Tighten screws on the P1 connector shell.

3. Attach the P2 connector (18 pin Berg) of the Video Cable to the 18 pin male Berg connector at the lower rear of the RSD. See Figure 6-1.

6.4 SDU RS232 Serial Ports

Two RS232 serial ports are provided through the System Diagnostic Unit located on the I/O plate of the back panel chassis with the dual 25 pin connectors. See Figure 5-2 for cabling and Figure 5-3 for the pin-outs.

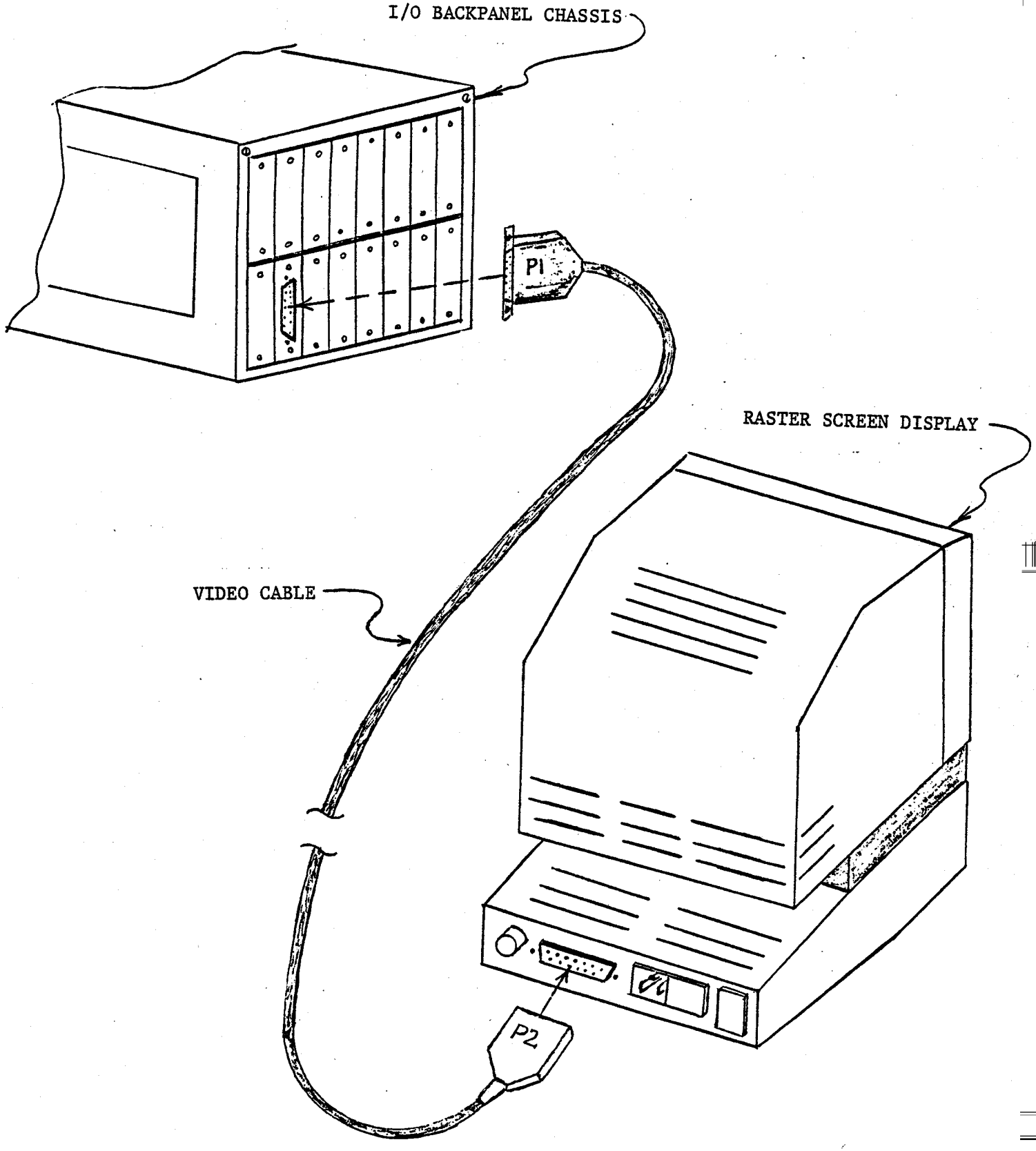


Figure 6-1. Video Cable Connections.

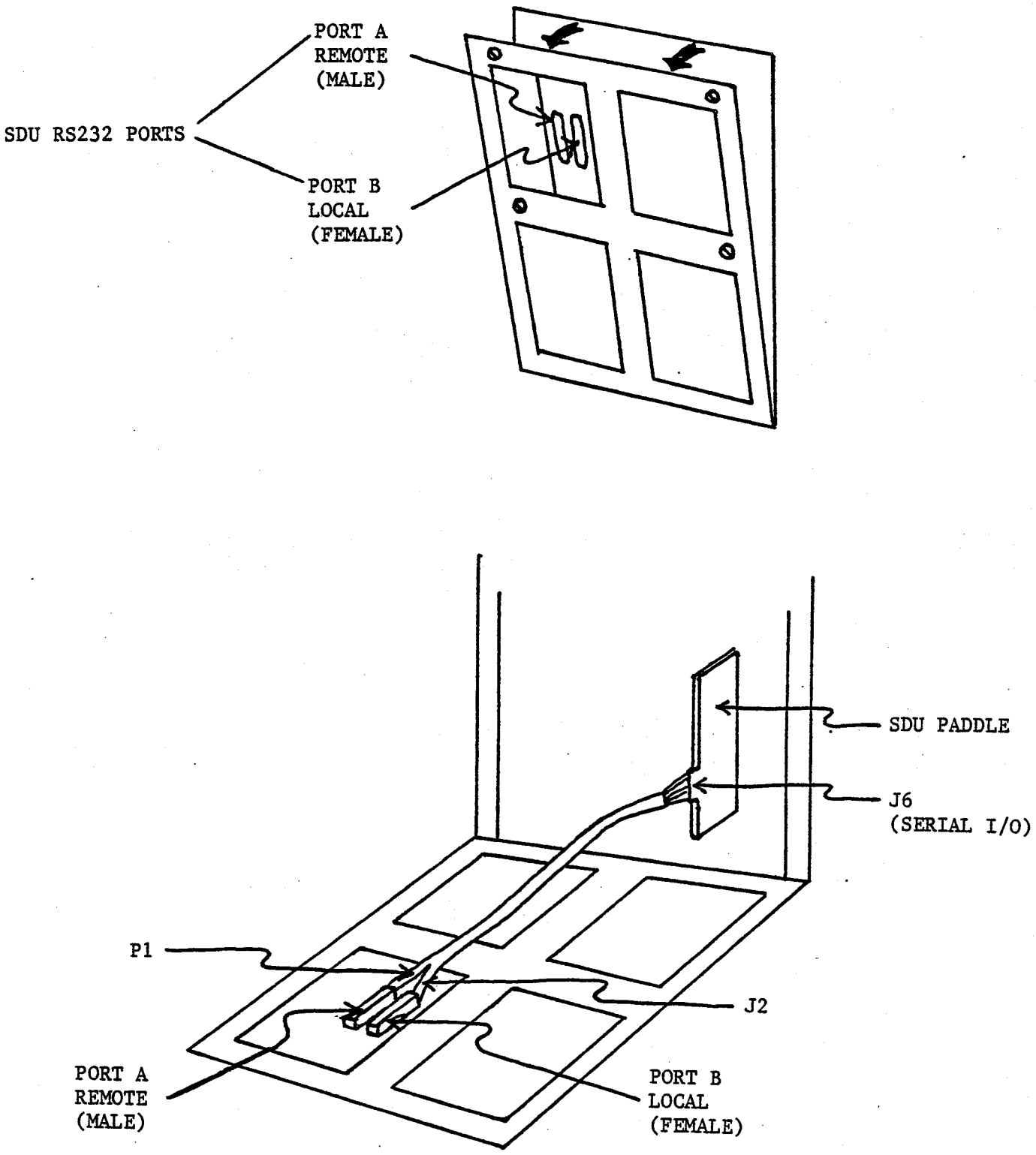


Figure 6-2. RS232 Cable Connections.

SYSTEM DIAGNOSTIC UNIT RS232 PORTS

PIN AND SIGNAL ASSIGNMENT TABLE

25-PIN, D-Shell Male Connector (Remote Channel)		25-PIN, D-Shell Female Connector (Local Channel)	
Pin No.	Signal Name	Pin No.	Signal Name
1	Chassis Gnd	1	Chassis Gnd
2	TXD	2	RXD
3	RXD	3	TXD
4	RTS	4	CTS
5	CTS	5	RTS
6	-	6	PULL-UP (+12V)
7	LOGIC GND	7	LOGIC GND
8	DSR	8	DTR
9	-	9	-
1Ø	-	1Ø	-
11	-	11	-
12	-	12	-
13	-	13	-
14	-	14	-
15	-	15	-
16	-	16	-
17	-	17	-
18	-	18	-
19	-	19	-
2Ø	DTR	2Ø	DSR
21	-	21	-
22	-	22	-
23	-	23	-
24	-	24	-
24	-	25	-

Figure 6-3. RS232 Pin-outs.

7 MODE SELECTOR SWITCH POSITIONS

When the power to the system is turned on or a reset is generated with the button on the back of the system, the SDU Monitor takes control of the system. The action taken by the SDU Monitor depends upon the position of the rotary switch located on the back of the system next to the reset button.

There are five switch positions. The following describes their use. Switch position Ø does not require SETUP, but the other switch positions do. If CMOS RAM is invalid (SETUP light remains on), then the action taken is that for switch position Ø, regardless of the actual actual setting.

Position Ø - The remote serial port (ttya) becomes the SDU Monitor at 3ØØ baud. This allows a modem to be used for remote setup or diagnostics.

Position 1 - The remote serial port becomes the SDU Monitor at the baud rate stored in CMOS RAM for this port. The default setup is 96ØØ baud. (See SETUP command in the SDU Monitor User's Manual) This is done by running the shell script in CMOS RAM corresponding to this switch position. The shell script for switch position 1 is:

```
ttyset ttya
```

Position 2 - Switch position 2 is only functional if the Nu Machine Operating System has been ordered with the disk drive. The shell script stored in CMOS RAM for this switch position is executed. These commands are:

```
reset
enable
uboot
```

The Operating System is booted automatically on power up, without operator intervention, using the remote serial port as the system console. If for some reason the uboot fails

then the remote port becomes the SDU Monitor at 9600 baud.

Position 3 - The shell script stored in CMOS RAM for this switch position is run. These commands are:

```
reset
enable
ttyset keytty
```

The RSD (Raster Scan Display) becomes the SDU system console. If for some reason the "ttyset keytty" fails then the remote port becomes the SDU Monitor at 9600 baud.

Position 4 - This works like switch position 1, except that the other port is used. The local serial port (ttyb) becomes the SDU Monitor at the baud rate stored in CMOS RAM for this port. The default setup is 9600 baud. This is done by running the shell script in CMOS RAM corresponding to this switch position. This shell script is:

```
ttyset ttyb
```

8 SYSTEM POWER UP

Once the system is fully installed the following steps should be followed:

1. Make sure that the disk heads are unlocked and all packing material has been removed. On Rack Systems, be sure all packing foam has been removed from the 1/2" tape drive.

2. If Rack System, on exterior side of AC distribution box, be sure that the chassis power switch (to the right) is off and the main circuit breaker (to the left) is down. On the tape drive front panel, be sure that the red toggle switch is in the "off" position. On the disk drive front panel, be sure that the furthest right white toggle switch is on "START". Now plug the 3ØA power cable into the wall outlet and twist the plug clockwise gently.

3. If Office System, on extreme right lower corner of rear panel be sure that main circuit breaker is down. Plug power cable into extreme lower right corner of rear panel and wall outlet.

4. Turn the main circuit breaker on (flip to up position).

5. If Rack System, on disk drive front panel, flip extreme right white toggle switch to "ON". On tape drive front panel, flip red toggle switch to "ON". On exterior side of AC distribution box, flip chassis power switch to "LOCAL".

6. All boards will come up with their LEDs on and the system CRT and RSD should be powered.

   When the system has been powered up or reset a >> prompt will appear on the SDU Monitor console device as determined by the Mode Selector Switch setting. This prompt indicates that the SDU Monitor is

operational.

7. Enter the command:

```
>>init
```

This command will reset and enable the Multibus, and the NuBus, and run SDU diagnostics.

8. Check the front panel lights. Both the ATTN and SETUP lights go on. After the SDU self-diagnostics pass the ATTN light goes off. The CMOS RAM on the SDU is then checked and if valid the SETUP light goes out. The RUN light indicates the SDU passed all diagnostics and is ready for use.

   If the SETUP light remains on and RUN light remains off, then the SDU has defaulted to Mode Switch position Ø. In this case only, connect a terminal to the SDU remote serial port (ttya), set it to 3ØØ baud and do the following:

   - If a Rack System, open the door on the tape drive front panel. Insert the 1/2" Diagnostics tape (clear side up), place on the hub and spin towards the right. Make sure tape turns smoothly. Push the LOAD REWIND button then the ON-LINE button on tape drive front panel. Setup the CMOS RAM via:

     ```
     >>/tar/setup clear
     >>/tar/setup eagle sp shell
     ```

     Push the square black button on back of rack cabinet (next to mode selector dial) to reset the system. If the SETUP light goes out and the RUN light goes on, then reset the mode selector dial as desired (see Section 7) being sure to connect the appropriate terminal.

   - If an Office System, insert the Diagnostics tape cartridge (clear side to the left). Set up the CMOS RAM via:

     ```
     >>/tarq/setup clear
     >>/tarq/setup micro sp shell
     ```

     Push the square black button (on extreme lower right panel next to mode selector dial) to reset the system. If the SETUP light goes out and the RUN light goes on, then reset the mode selector dial as desired (see Section 7) being sure to connect the

appropriate terminal.

9 DIAGNOSTICS

The following procedure is for running ram, vcmem and cpu diagnostics. The SDU Monitor User's Manual, Section 6 may be referred to for options used with the procedure below.

New Rack and Office Systems are shipped with the diagnostics installed on disk.

9.1 Procedure for Running Diagnostics from Disk

1. Display the configuration via:

```
>> cat /disk/monitor/conf/std
```

   The first column is the board, port or disk partition name. The next column gives the NuBus address. The second hexidecimal character in the address represents the slot number. For example:

```
cpu:    FB00000
vcmem:  F800000
ram:    F700000
ram:    F900000
```

   The cpu card is in slot 11. The vcmem card is in slot 8. There are 2 ram cards, one in slot 7 and one in slot 9.

2. Run diagnostics tests via:

```
>> cd /disk/monitor/diag
>> filename -[options] [n]
```

   where filename = ram, vcmem, or cpu;
   options = tS or other options listed in SDU Monitor User's Manual, Section 6; and
   n = card slot number.

For example:

```
>> cd /disk/monitor/diag
>> cpu -tS 11

      :
      : diagnostic messages

>> vcmem -tS 8

      :
      : diagnostic messages
```

Note: Disk drive and tape drive diagnostic tests are similar to the above, but should not be performed until the explanations given in the SDU Monitor User's Manual Section 6 are thoroughly understood.

9.2 Procedure for Running Diagnostics from Tape

If your disk is corrupted you may run diagnostics from tape.

- If Rack System, open the door on tape drive front panel. Insert tape (clear side up), place on hub and spin towards right. Make sure tape turns smoothly. Close door. Push the LOAD REWIND button then the ON-LINE button on tape drive front panel.

```
>> /tar/filename -[options] [n]
```

- If Office System, insert the tape cartridge (label side to the left).

```
>> /tarq/filename -[options] [n]
```

where filename = **ram, vcmem, or cpu;**
options = **tS** or other options listed in SDU Monitor User's Manual, Section 6; and
n = card slot number.

9.3 Installing Diagnostics on Rack System

The following procedure is for installation of diagnostics on Rack systems with Fujitsu Eagle disk drive.

New Rack and Office Systems are shipped with all software installed; no additional installation procedures are required.

This procedure should only be used when the disk has been totally corrupted. When the Nu Machine Operating System was supplied with the system, then use the procedures outlined in **Nu Machine Release and Update Information**, not the following.

1. Power up the system, select switch setting 3, and reset. The SDU Monitor should respond with a >> prompt on the **rsd**.

   NOTE: If the **SETUP** light remains on after the reset, then the SDU defaults to Mode Switch position Ø. In this case only, a terminal must be connected to the SDU remote serial port (ttya) and be set to **3ØØ** baud to do step 4.

   If an **rsd** is not available, a terminal connected to **ttya** can be used by selecting switch setting 1. However, when **SETUP** remains on after reset, the terminal should be set to 3ØØ baud. After setup is complete and the system is reset, the terminal should be set to 96ØØ baud.

2. Load the Nu Machine Diagnostics tape.

3. Setup the CMOS ram via:

   ```
   >> init
   >> /tar/setup clear
   >> /tar/setup eagle sp shell
   ```

4. Reset the system and check that the **SETUP** light goes out and the **RUN** light is on.

   Do not continue with the installation procedure if the **RUN** light is not on. The prompt should now appear on the rsd if switch setting 3 is used, or on ttya if switch setting 1 is used. The terminal on ttya must now be set to 96ØØ baud.

5. This step is only required if the disk is in an unknown state. If the disk was already formatted, then this step may be skipped. Reformat the disk via:

```
>> /tar/2181 -tvDF
```

6. Copy the SDU file system image from tape to disk via:

```
>> init
>> copy tape/3 $disk
```

The SDU Monitor should respond with "1047552 bytes copied".

9.4 Installation of Office System Diagnsotics

The following procedure is for installation of diagnostics on Office systems with Fujitsu Micro disk drive.

New Rack and Office Systems are shipped with all software installed; no additional installation procedures are required.

This procedure should only be used when the disk has been totally corrupted. If the Nu Machine Operating System was supplied with the system, then use the procedures outlined in **Nu Machine Release and Update Information**, not the following.

1. Power up the system, select switch setting 3, and reset. The SDU Monitor should respond with a >> prompt on the **rsd**.

   NOTE: If the **SETUP** light remains on after the reset, then the SDU defaults to Mode Switch position Ø. In this case only, a terminal must be connected to the SDU remote serial port (ttya) and be set to **3ØØ baud** to do step 4.

   If an **rsd** is not available, a terminal connected to **ttya** can be used by selecting switch setting 1. However, when **SETUP** remains on after reset, the terminal should be set to 3ØØ baud. After setup is complete and the system is reset, the terminal should be set to 96ØØ baud.

2. Load the Nu Machine Diagnostics tape.

3. Setup the CMOS ram via:

   ```
   >> init
   >> /tarq/setup clear
   >> /tarq/setup micro sp shell
   ```

4. Reset the system and check that the **SETUP** light goes out and the **RUN** light is on.

   Do not continue with the installation procedure if the **RUN** light is not on. The prompt should now appear on the rsd if switch setting 3 is used, or on ttya if switch setting 1 is used. The terminal on ttya must now be set to 96ØØ baud.

5. This step is only required if the disk is in an unknown state. If the disk was already formatted, then this step may be skipped. Reformat the disk via:

```
>> /tarq/2181 -tvDF
```

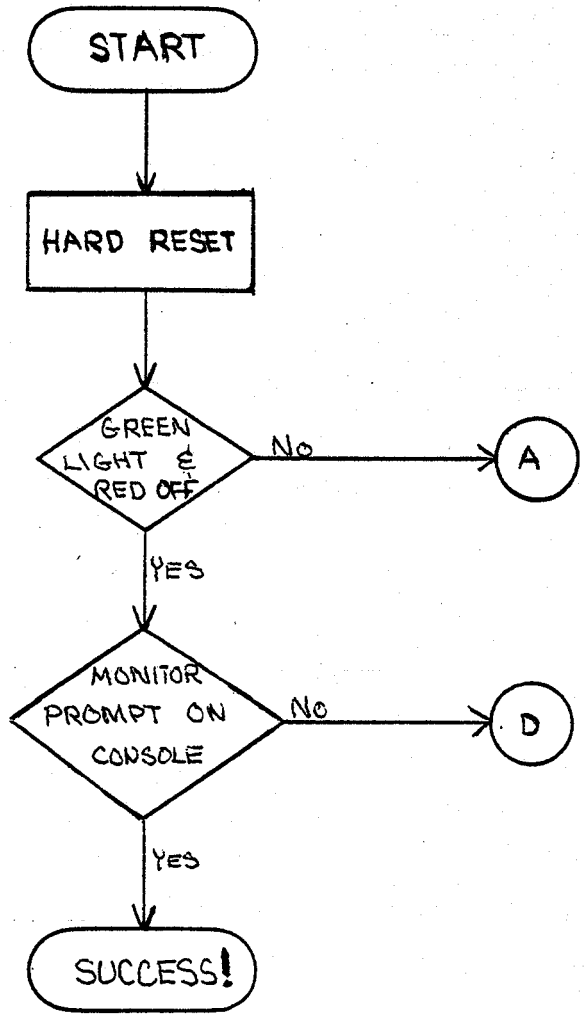
6. Copy the SDU file system image from tape to disk via:

```
>> init
>> copy quart/3 $disk
```

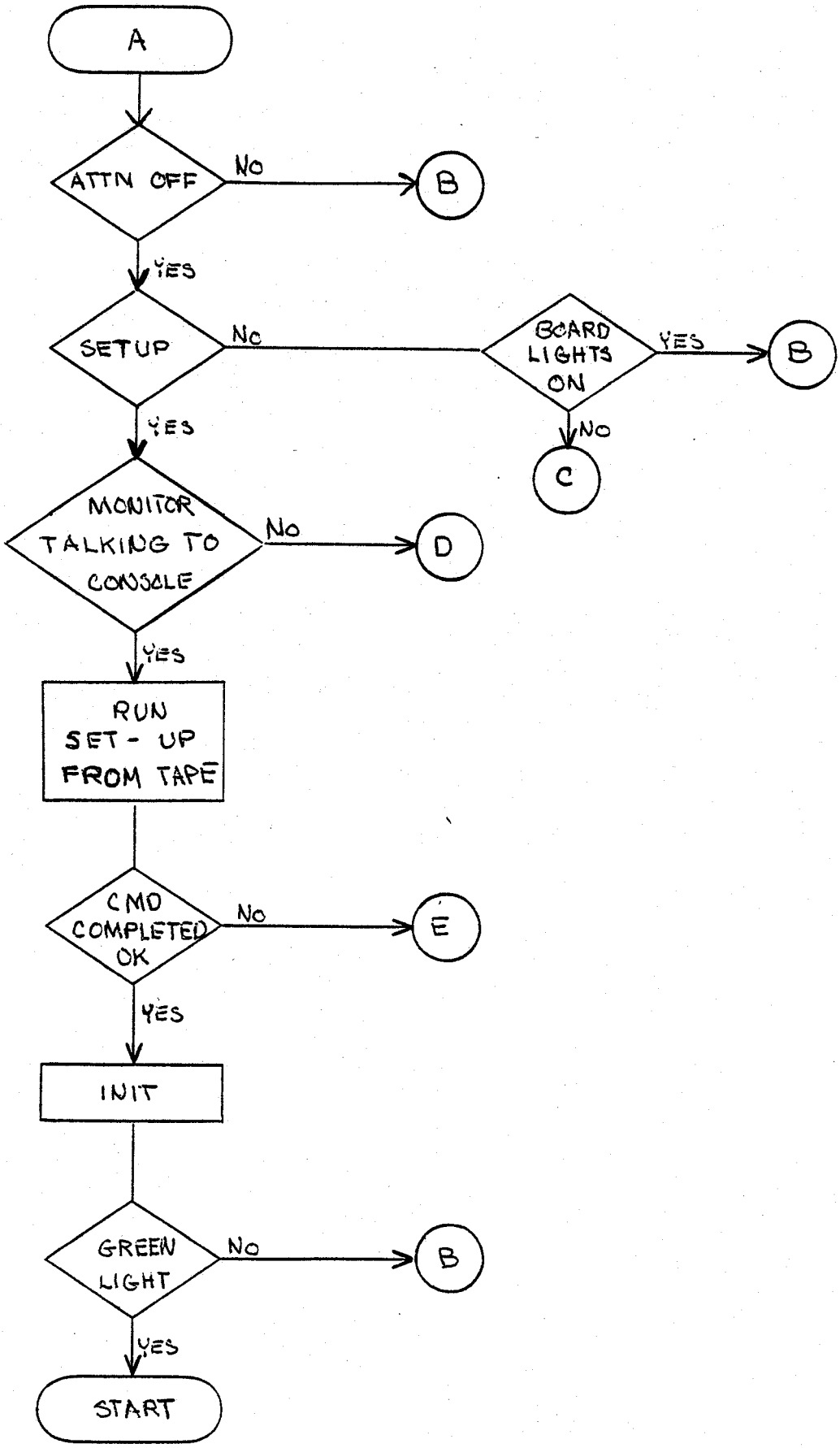
The SDU Monitor should respond with "1Ø47552 bytes copied".

10 TROUBLESHOOTING FLOW

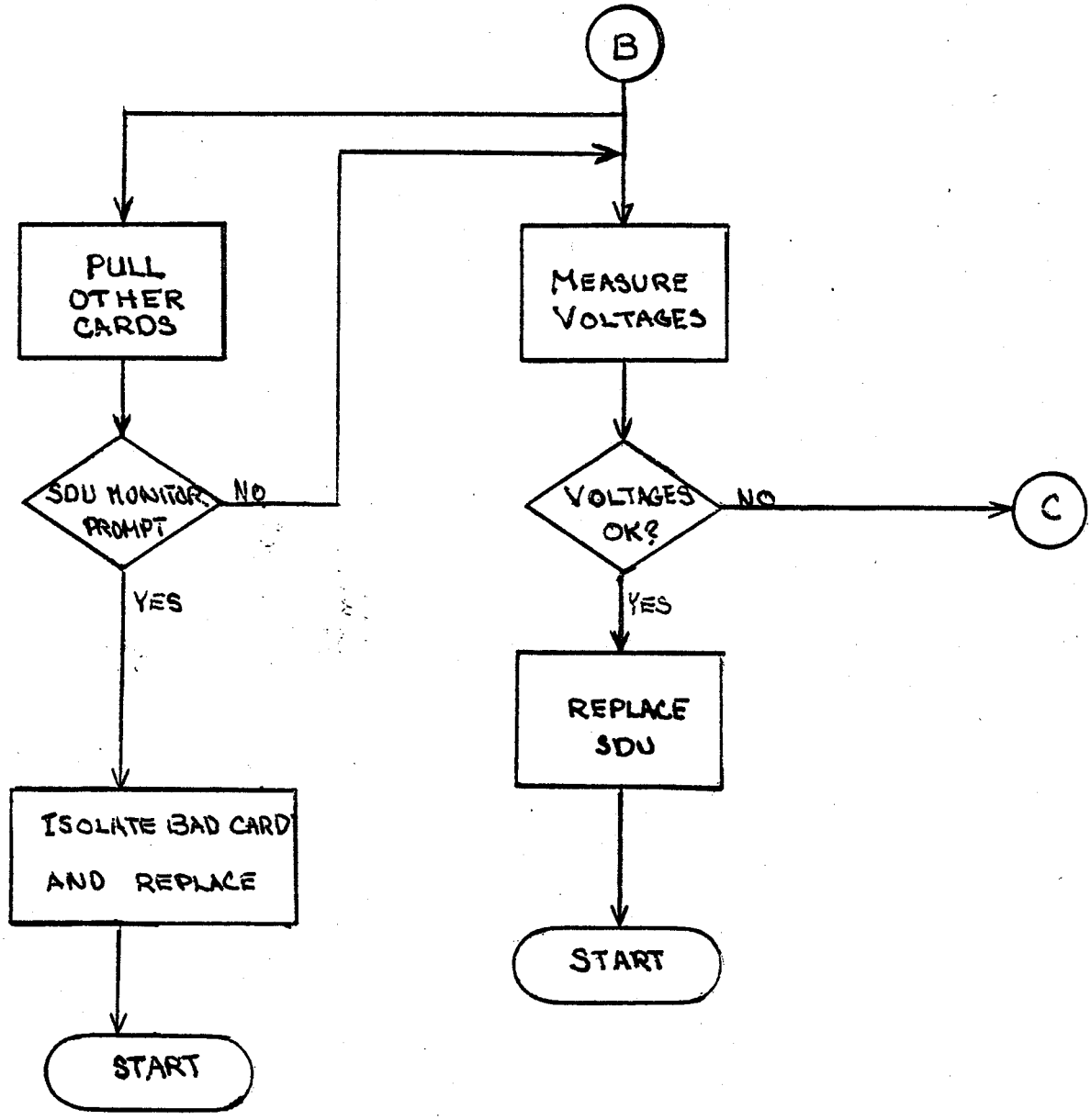
SYSTEM CHECKOUT



ALL LIGHTS OFF OR RED LIGHTS

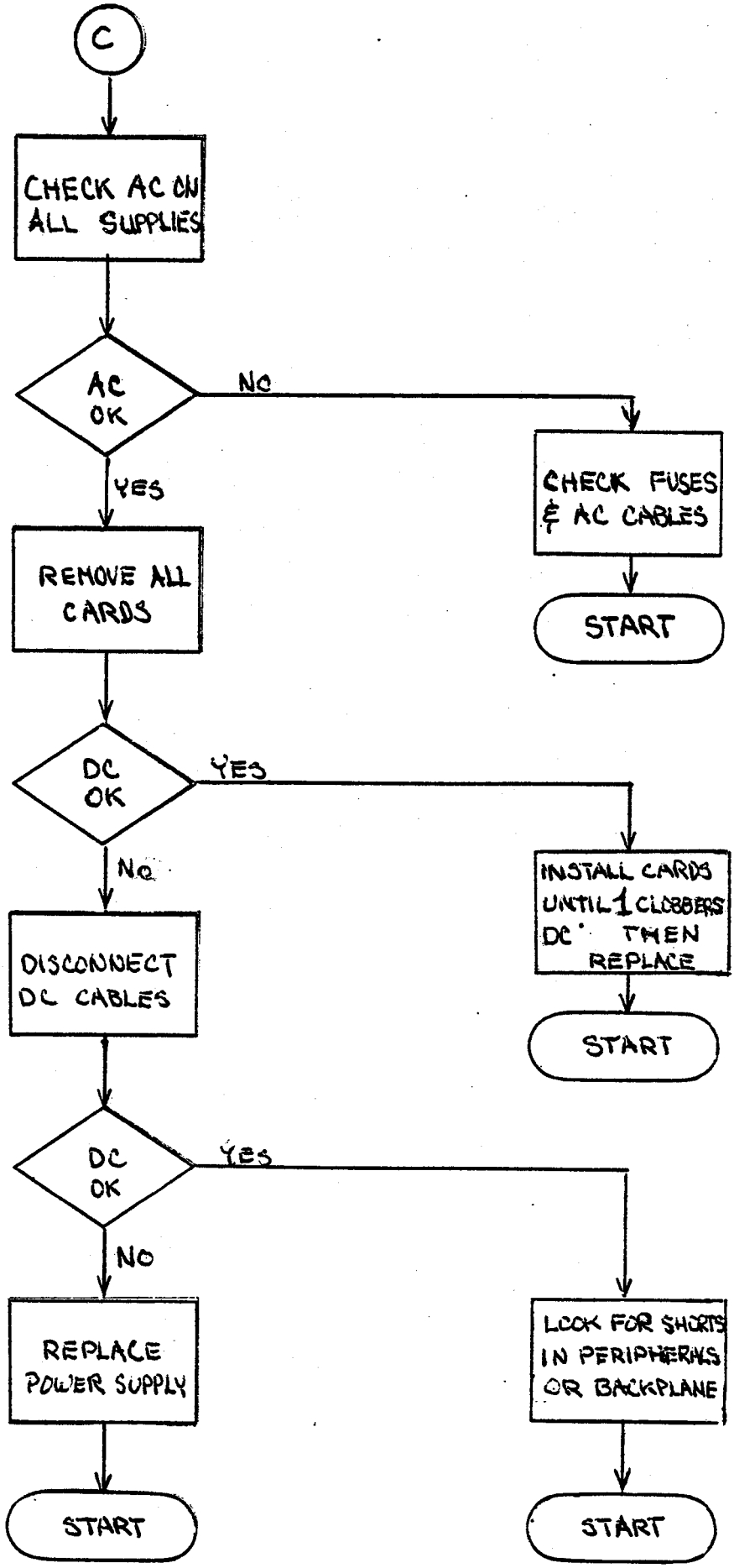


ATTN ON OR RANDOM SDU MISBEHAVIOR

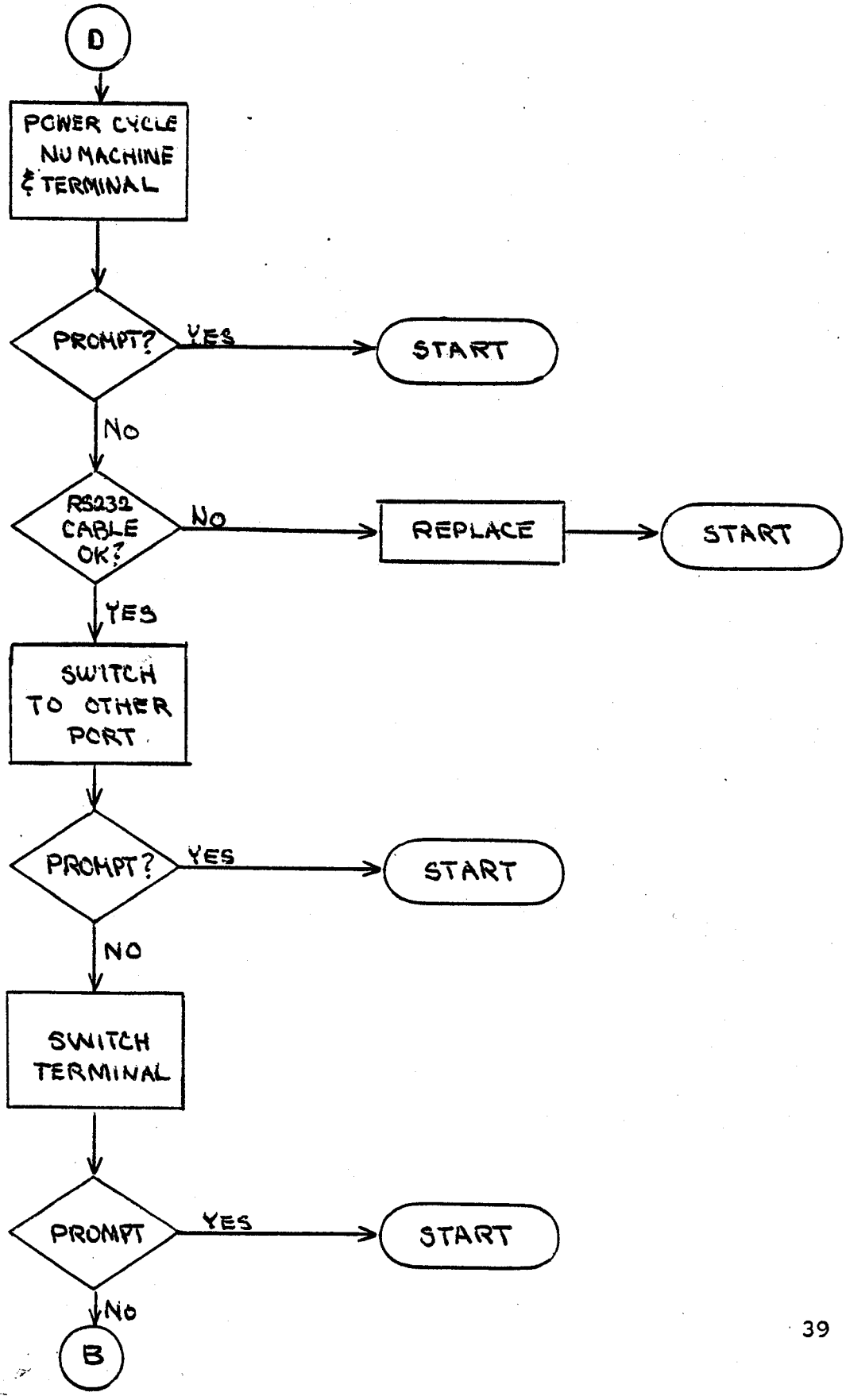


POSSIBLE SDU PROBLEMS

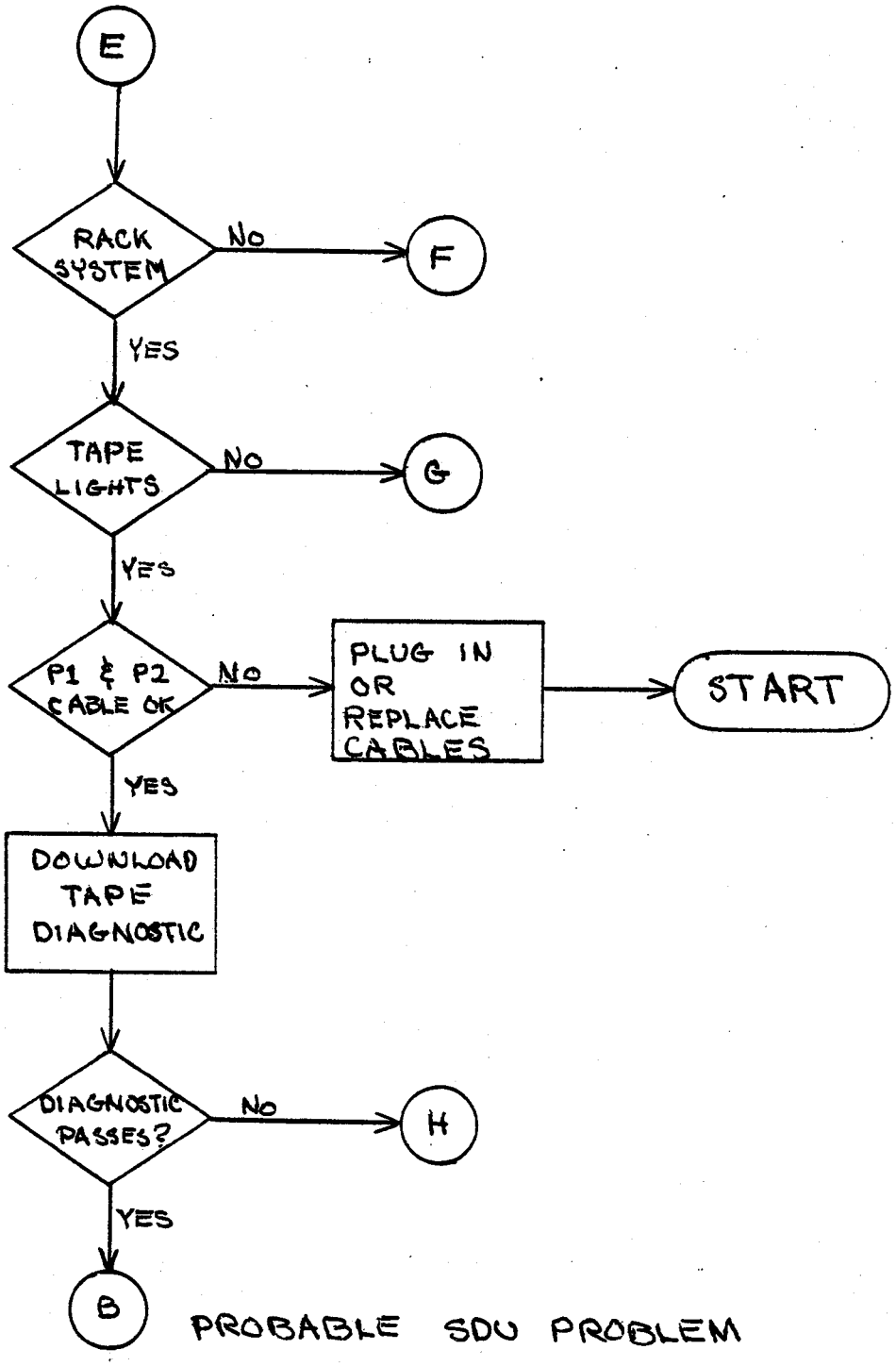
POWER SUPPLY PROBLEMS VOLTAGES DO NOT MEASURE OK



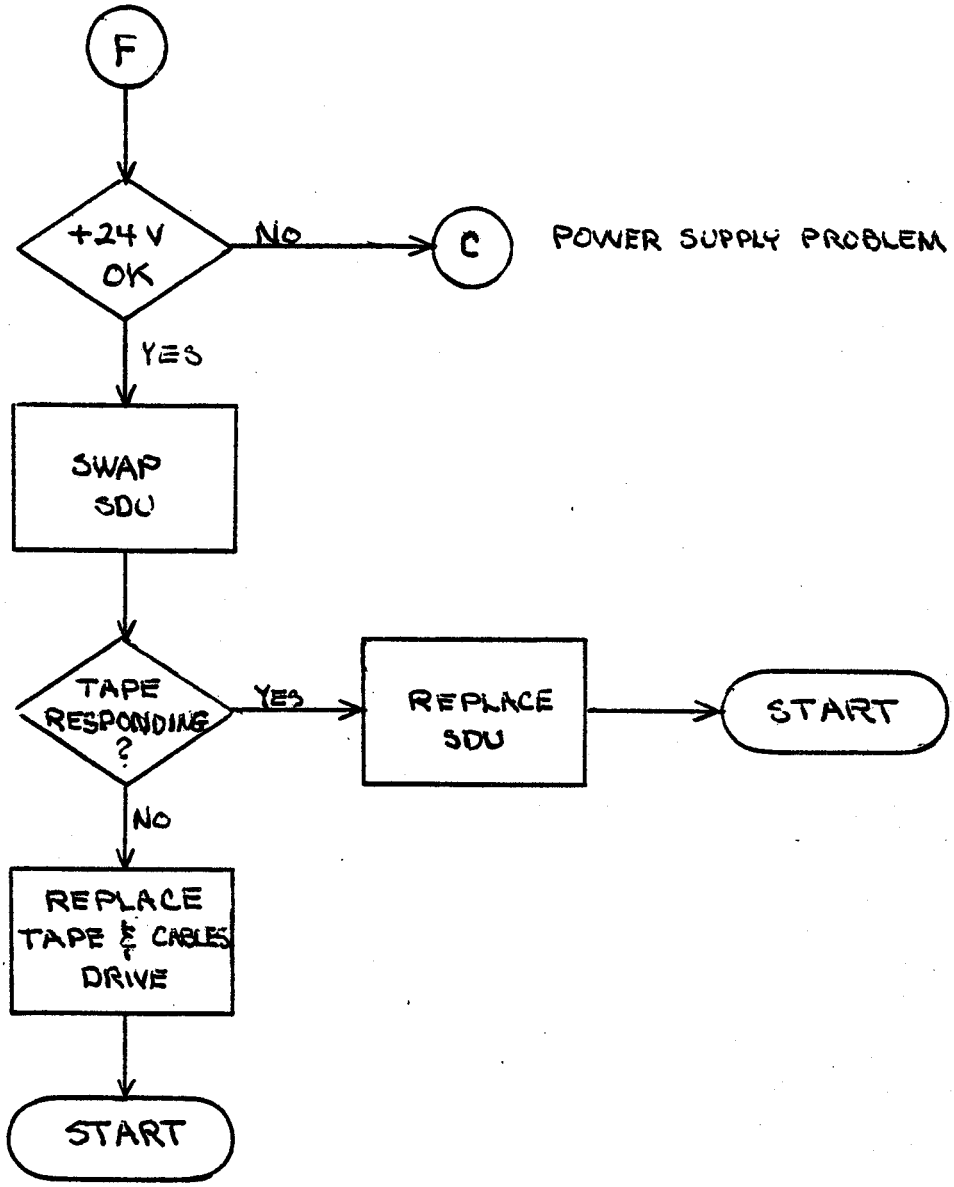
NO MONITOR PROMPT ATTN OFF



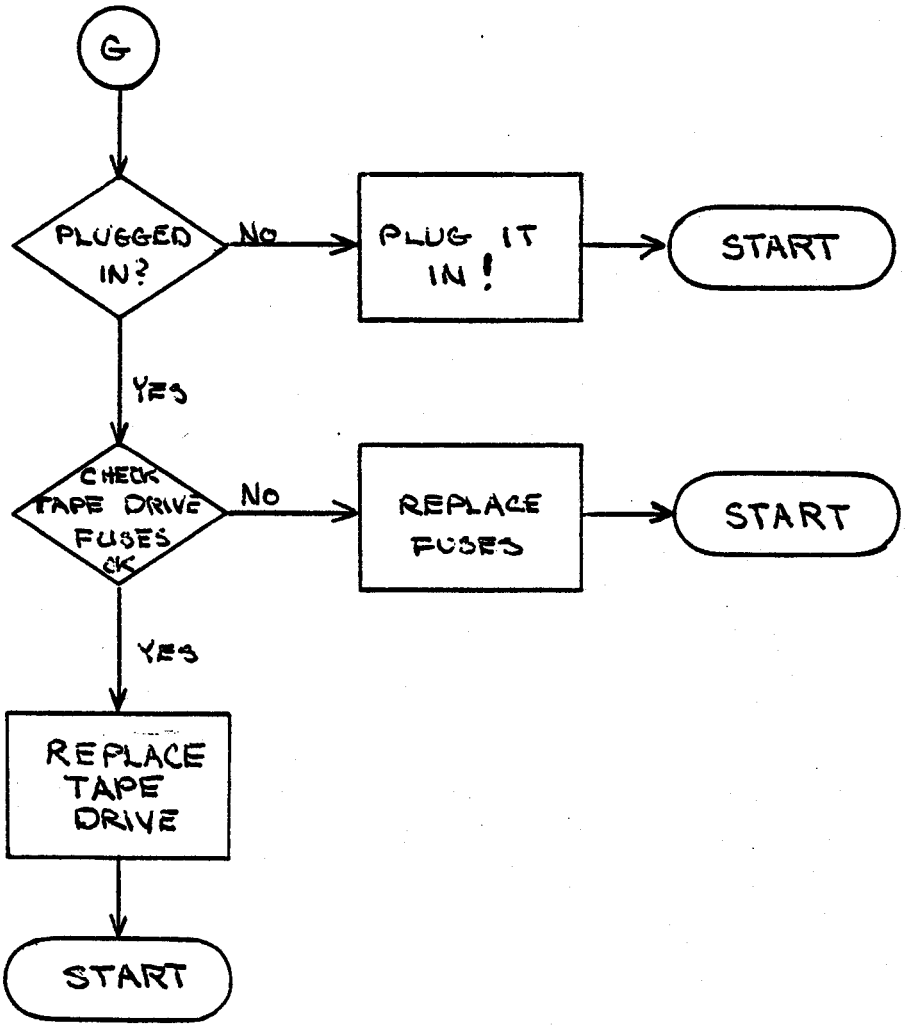
MONITOR SEEMS OK, CANNOT RUN SETUP FROM TAPE



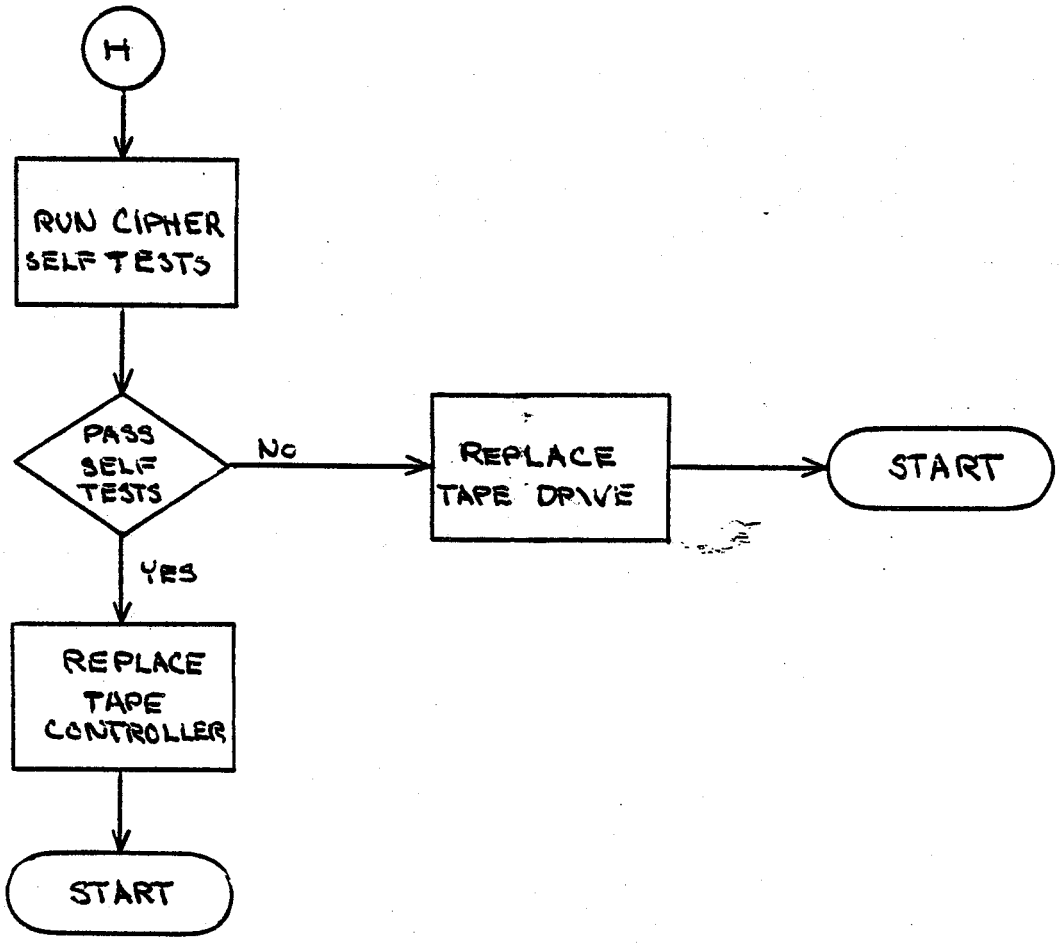
1/4 INCH TAPE NOT RESPONDING SDU MONITOR SEEMS OK



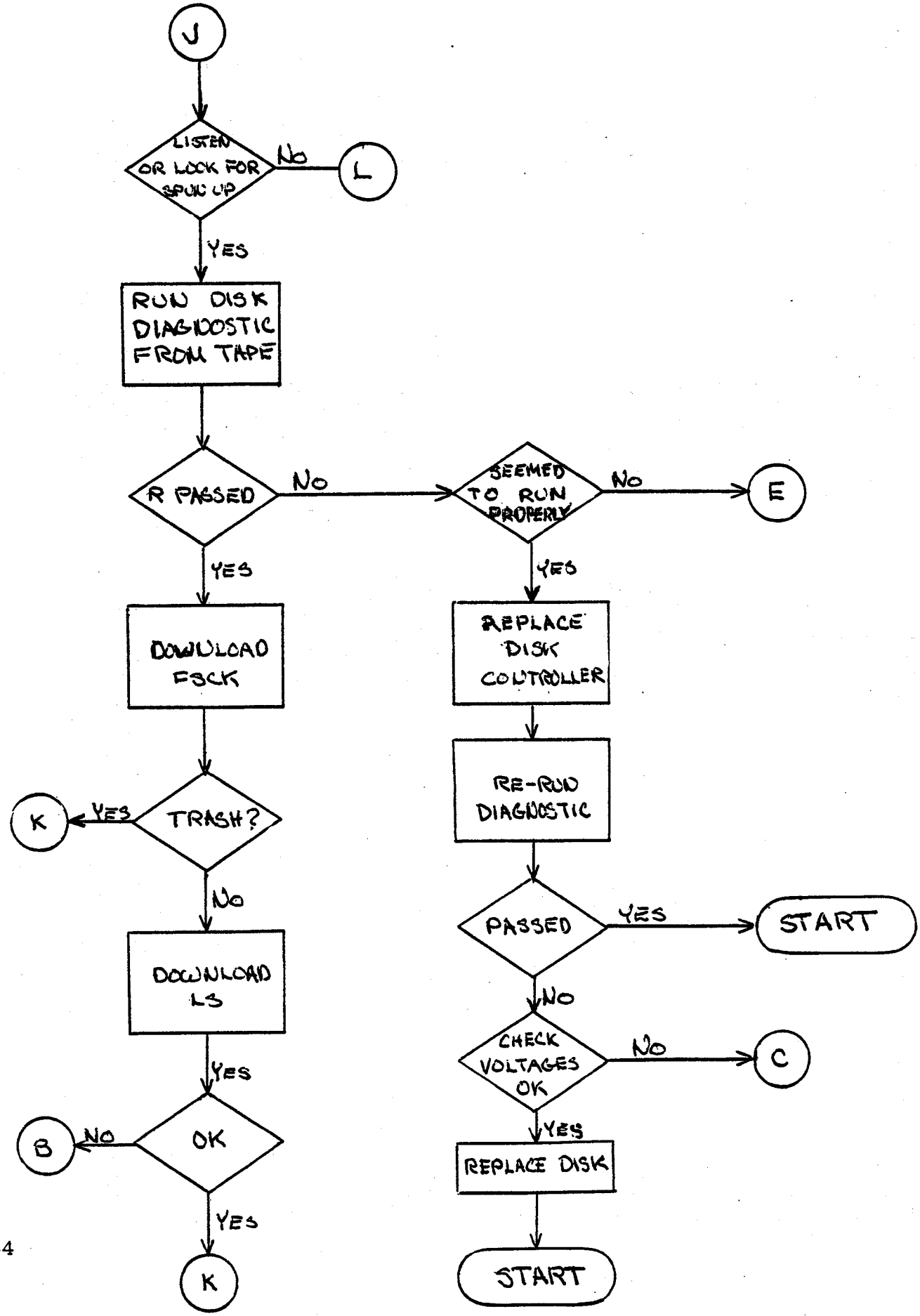
TAPE UNIT NOT RESPONDING



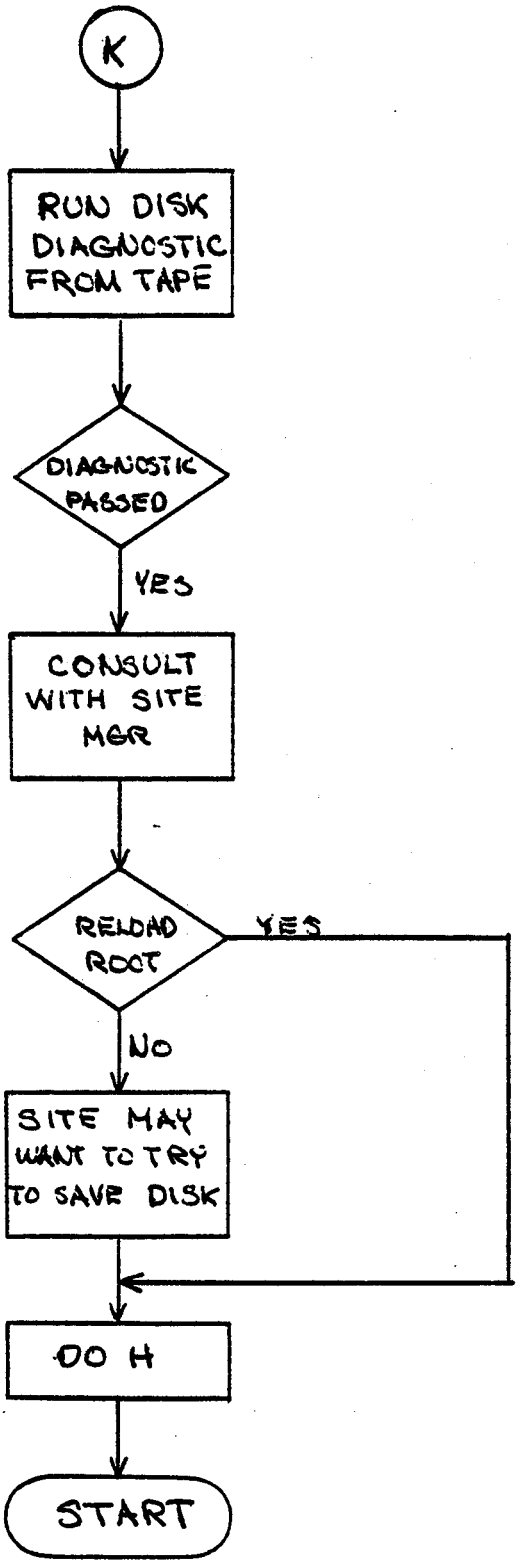
TAPE DRIVE/CONTROLLER PROBLEMS



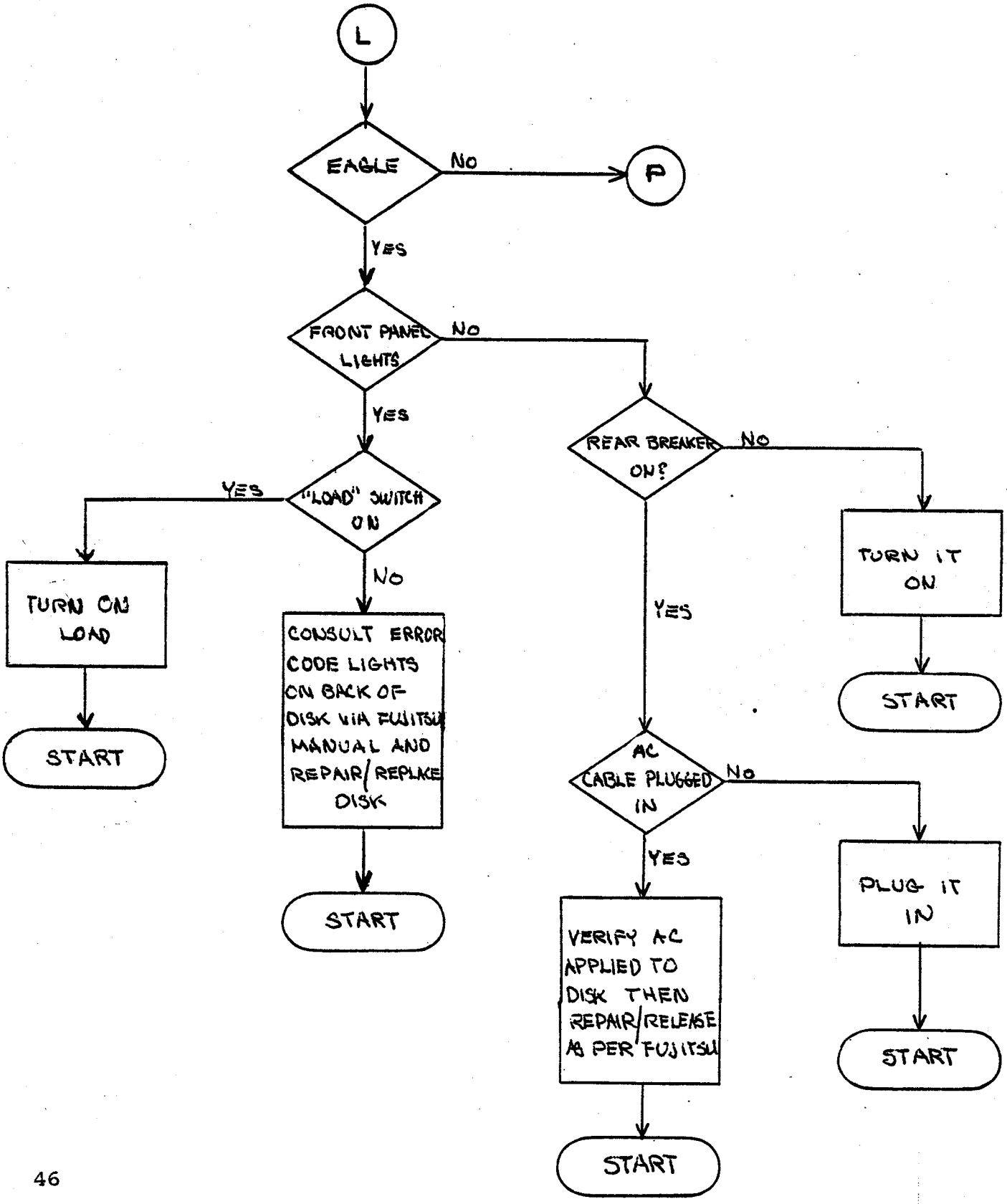
MONITOR LS FAILING



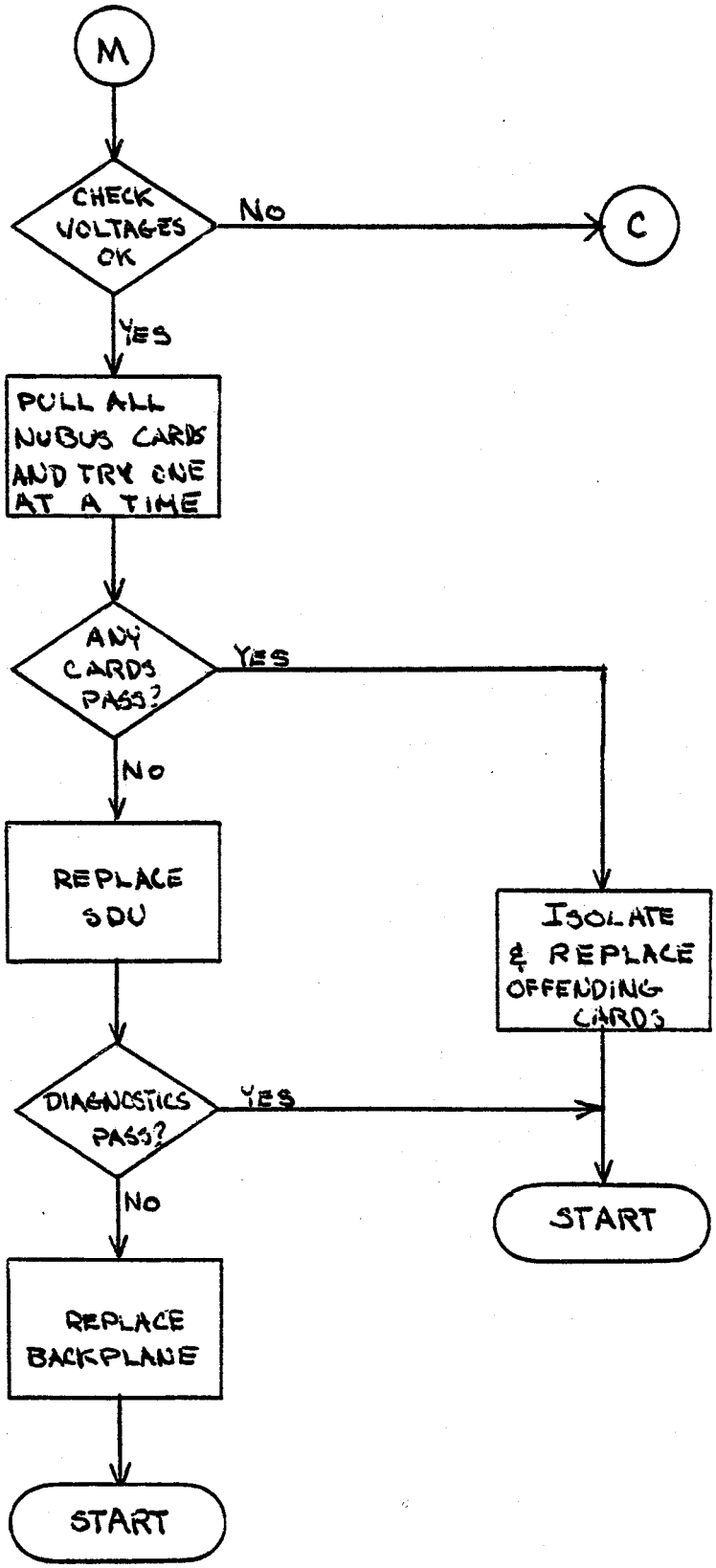
FSCK SAYS DISK SCRAMBLED OR CMOS MISSING



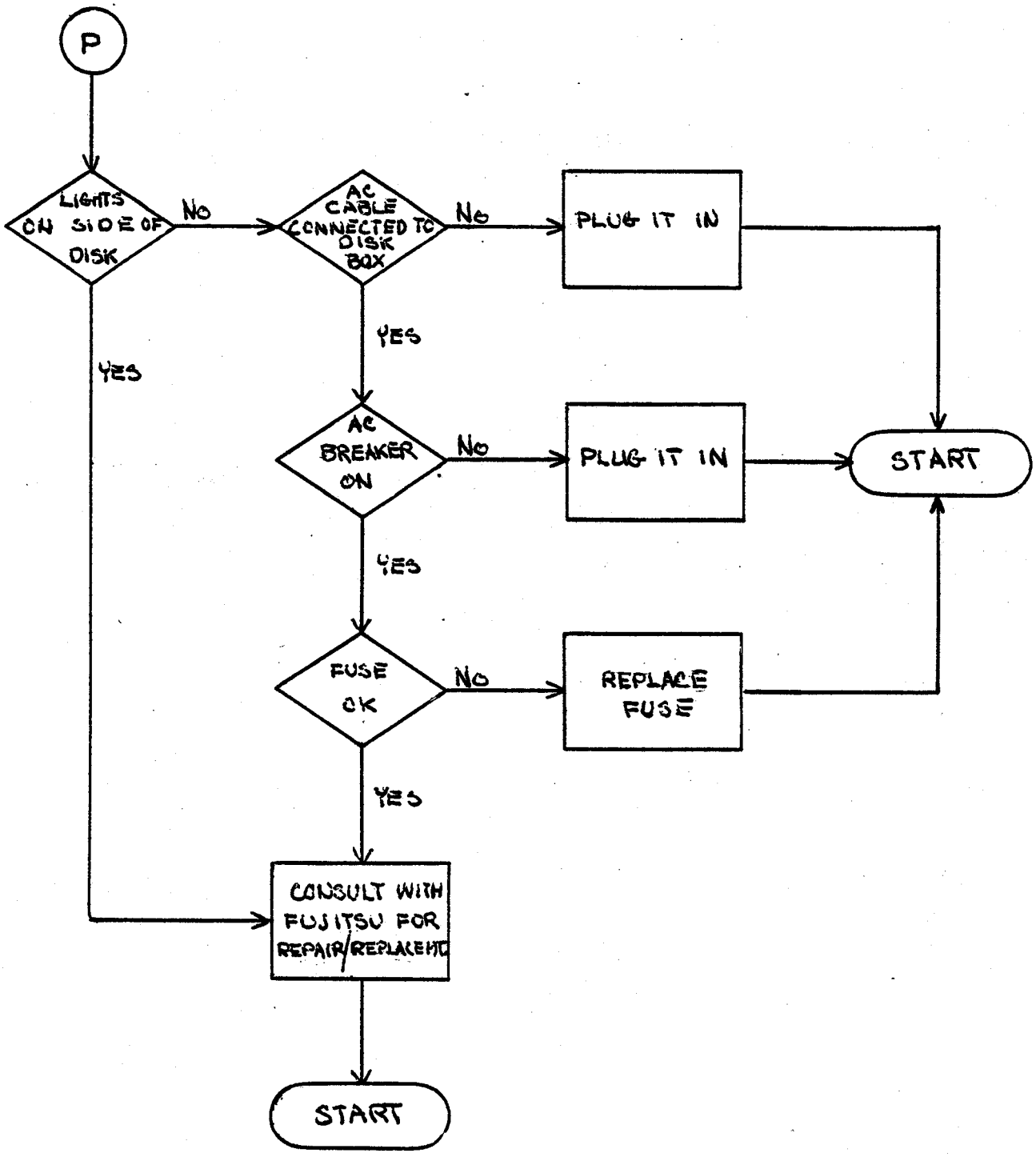
DISK DOES NOT SEEM TO BE SPINNING



ALL NUBUS BOARDS FAILING DIAGNOSTICS



MICRO DISK NOT WORKING



Introduction to the Lambda
A Guide for Programmers

for LMI Release 2.0 1370-0000

Published by LMI 1000 Massachusetts Avenue. Cambridge MA 02138 USA

Information in this document corresponds to Release 2.0.

This document was written by:

Meryl Cohen
Robert Ingria

Summary Table of Contents

Table of Contents

Preface

Purpose of This Manual

This manual is an introduction to the LMI Lambda programming system. It concentrates on software rather than hardware facilities, but hardware issues are mentioned when necessary. By the time you finish reading this Introduction and following the examples at the terminal, you should be able to use the Lambda as a serious programming tool, and to develop your ability as a LISP programmer.

LISP Machine Inc. welcomes all comments and suggestions for improvement; contact Dr. Sarah Smith, Documentation Manager, at (617) 876-6819.

Audience

The *Introduction to the Lambda* is written to meet the needs of most beginning users of this machine. No previous LISP Machine experience is assumed, but some familiarity with basic LISP concepts, syntax, and terminology is.

Other Information Resources

This manual is intended neither as a guide to LISP, nor as a comprehensive guide to the Lambda. For more information on both the hardware and software in the Lambda system consult the sources described below.

More Manuals

Appendix C contains a chapter by chapter guide to manuals or books that will give you more detailed information on the material covered in each chapter. With a few exceptions these materials are provided when you purchase the Lambda.

Telephone Support

LISP Machine Inc. provides telephone software and hardware support to its clients. (Support is free during the warranty period, and available subsequently with a service contract.) This service is available from 8:30 AM to 8:00 PM (Eastern Standard Time) Monday through Friday. To get help, from outside Massachusetts call 1-800-872-LISP; within Massachusetts call 1-800-325-6115. You can also dial in to our VAX computer and leave mail for customer service staff. For an account call George Colpitts of Customer Service.

Classes

LMI currently offers beginning and intermediate level ZetaLISP-Plus classes. Other classes will be added in the future. For information and reservations contact Pam Renowden at (213) 612-1116.

Documentation Conventions

There are many operations that can be performed at the Lambda console using the keyboard and mouse. In order to describe these operations unambiguously, the following notational conventions have been adopted in this document. (For definitions of the terms used, see Section 1.1.3, pages 11–12 below.)

Input Device Conventions

Modifier keys are represented as follows:

CTRL
GREEK
HYPER
META
SHIFT
SUPER
TOP

A character modified by a modifier key is connected to it by a dash:

CTRL-B
HYPER-B

If more than a single modifier key is pressed, each is connected to the next by a dash:

CTRL-META-B
HYPER-META-B

Function keys are indicated by a lozenge: ⟨ ⟩. Some common function keys are:

⟨ABORT⟩
⟨END⟩
⟨HELP⟩
⟨STATUS⟩
⟨SYSTEM⟩
⟨TERMINAL⟩

An argument to a function key is separated from it by a space:

⟨TERMINAL⟩ **F**
⟨SYSTEM⟩ **P**
⟨TERMINAL⟩ ⟨CALL⟩

Mouse buttons are indicated by the symbol: ⋂. This symbol by itself indicates clicking any mouse button. ⋂ with one of the letters **L**, **M**, or **R** inside indicates clicking the left, middle, or right mouse buttons, respectively. Any of these button indicators can appear with a **2** beneath it, indicating that that button should be clicked twice. For example:

⋂L indicates that the left mouse button should be clicked.
⋂R2 indicates that the right mouse button should be clicked twice.

A *mouse-sensitive item* appears in a box, such as **Split Screen**. To indicate that a particular mouse button should be clicked while the mouse is pointing at a mouse-sensitive item, a mouse button indicator appears before the mouse-sensitive item. For example:

R **Split Screen**
: Indicates that the right mouse button should be clicked while the mouse is pointing at **Split Screen**.

Choice boxes are indicated by the symbol ■. A choice box is preceded by its label. For example:

Do It ■ Indicates the choice box labelled **Do It**.

Function Conventions

Some of the general conventions for describing LISP forms are described here. For a full discussion, see the *LISP Machine Manual*.

A typical description of a LISP function looks like this:

function-name *arg1 arg2* &optional *arg3* (*arg4 arg2*) Function
: Adds together *arg1* and *arg2*, and then multiplies the result by *arg3*. If *arg3* is not provided, the multiplication isn't done. **function-name** then returns a list whose first element is this result and whose second element is *arg4*. Examples:

```
(function-name 3 4) evaluates to (7 4)
(function-name 1 2 2 'bar) evaluates to (6 bar)
```

Note the use of fonts and typefaces. The name of the function is in bold-face in the first line of the description and the arguments are in italics. Within the text, printed representations of LISP objects are in a different bold-face font, as in **(+ foo 56)**, and argument references are italicized, as in *arg1* and *arg2*. A different, fixed-width font, as in **function-name**, is used for LISP examples that are set off from the text, as well as for output printed on the Lambda screen. Another font convention is that filenames are in bold-face, all upper case, as in **LAM3:RJPI:LISPM.INIT**.

The descriptions of special forms and macros look like this:

do-three-times *form* Special form
: Evaluates *form* three times and returns the result of the third evaluation.

with-foo-bound-to-nil *forms* Macro
: Evaluates *forms* with the symbol **foo** bound to **nil**. It expands as follows:

```
(with-foo-bound-to-nil
     form1
     form2...) expands into:
(let ((foo nil))
     form1
     form2...)
```

Since special forms and macros are the mechanism by which the syntax of LISP is extended, their descriptions must describe both their syntax and their semantics; functions follow a simple

consistent set of rules, but each special form is idiosyncratic. The syntax is displayed on the first line of the description using the following conventions. Italicized words are names of parts of the form which are referred to in the descriptive text. They are not arguments, even though they resemble the italicized words in the first line of a function description. Ellipses ('...') indicate that the subform (italicized word or parenthesized list) that precedes them may be repeated any number of times (possibly no times at all). Thus the first line of the description of a special form is a "template" for what an instance of that special form would look like, with the surrounding parentheses removed.

The semantics of a special form includes not only what it "does for a living", but also which subforms are evaluated and what the returned value is. Usually this will be clarified with one or more examples.

Descriptions of variables ("globally special" variables) look like this:

typical-variable Variable

The variable **typical-variable** has a typical value....

LISP Culture

History of the LISP Machine

The LISP Machine was originally designed at the Artificial Intelligence Laboratory at MIT to provide an efficient, single-user programming environment for the development of large LISP programs. The need for such an environment arose from the problems of running multiple LISP programs on time-sharing systems. The earliest computers on which LISP was developed were expensive machines that were originally designed to be single-user systems. Since it was not cost-effective to devote such machines to only one user at a time, time-sharing systems were developed, which allowed several researchers to use a single machine at the same time. Project MAC, at MIT, developed such a time-sharing (or "time-stealing", as it was then called) system, called ITS, for PDP-6 and PDP-10 computers. In addition, a new dialect of LISP, MacLISP, was developed, based on LISP 1.5. As more and more researchers worked on LISP programs on individual ITS machines, system resources were strained and response-time was slowed considerably, hindering program development.

The earliest computers were much more expensive than the cost of programmer time, so it made sense to have as many users on a single system as possible. However, in later years the cost of computing time became relatively inexpensive compared to the cost of wasted programmer time, so that it made sense to have highly efficient single-user programming environments. The availability of integrated circuits and the relative cheapness of systems with massive computing power made it possible to develop a single-user system for LISP development.

The first LISP machine was called CONS and became operational in 1975. This was succeeded by the CADR, in 1978, and the Lambda, in 1983. The multiuser Lambda, in which multiple LISP processors share memory, and the UNIX interface became operational in 1984. All of these machines were designed for compatibility with the AI programmiing environment at MIT. The dialect of LISP that runs on the LISP Machine, ZetaLISP-Plus, is descended from MacLISP. Many of the features of the LISP Machine are based on analogous facilities in ITS: the who line, Peek, **MORE** processing, and the syntax of filenames are all developments of ITS. The ZMACS editor is an extended version of the EMACS editor, originally written in the TECO language on ITS. The multitasking environment of the LISP machine, where a user can move from a LISP Listener to an editor to any other program, also has its origins in ITS, which also supports multiple jobs.

The Lambda may be seen as a natural outgrowth of the programming environment that developed at MIT: extensions to LISP that have proved useful and aspects of the ITS operating system that have provided support for program development have been re-implemented in an expanded and more efficient manner, creating an optimum environment for LISP programming and development.

The LMI Lambda Philosophy

LMI has designed the Lambda to be the ideal programmer's workshop. All the best programming tools and facilities are immediately available for you to use. ZetaLISP-Plus is an extremely powerful and natural programming language; it is easy to design and to code programs the way that the problem appears to you. Flavors give you the power to use Smalltalk-like, object oriented programming techniques within the Lambda environment. ZetaLISP-Plus may be used as either

an interpreted or a compiled language. This gives you the running speed of compiled code and the ease of development of an interpreted language.

While LMI believes that ZetaLISP-Plus offers an ideal programming environment, both as a language and an operating system, we also offer other languages (Prolog, Fortran, and C; Pascal will be availble in late 1985), and the UNIX operating system. You can use these separately or in conjunction with LISP.

1. The Lambda Environment

1.1 The Hardware

The hardware in your LMI Lambda can be thought of as consisting of two groups: the hardware that runs the computer, and the hardware that you use to talk to the machine. This first group includes:

- one cabinet containing:
  - a disk drive
  - a tape drive
  - Lambda boards, memory, and an optional UNIX processor.
- one terminal used for booting.

Depending upon how your site is arranged, you may never (or rarely) see this equipment. It can be physically distant from the second group of hardware—the user console. This consists of:

- a high resolution black and white monitor
- a medium resolution color monitor (optional)
- a high resolution color system (optional)
- the AI keyboard
- a mouse and optical pad

1.1.1 The High Resolution Monitor

The high resolution monitor included with your Lambda provides an easy-to-read display for both graphics and text applications. When text reaches the bottom of the screen it wraps around again to the top; text does not scroll. You can adjust the viewing angle by gently pushing the monitor housing.

Normally, the monitor displays black output on a white background; this is called black-on-white mode. If you prefer white output on a black background (white-on-black mode), you can switch from one to the other by typing ⟨TERMINAL⟩ C.

Information Lines

The display on the high resolution monitor looks like that in Figure 1. At the bottom of the screen are two lines that always provide you with information about the status of the system. The reverse video upper line is the mouse documentation line. This tells you what the mouse buttons do in the current environment. If you are looking at a pop-up menu, this line tells you the effect of the currently "boxed" choice. (For a more complete discussion, see Chapter 6, "The Window System".)

The line below the mouse documentation line is called the *who line*. This contains several kinds of important information. The leftmost entry is the current date and time. Next in line is the username of the person currently logged in. Following this is the name of the current package (for details, see Section 4.3, beginning on page 28). Next is usually a description of the state of

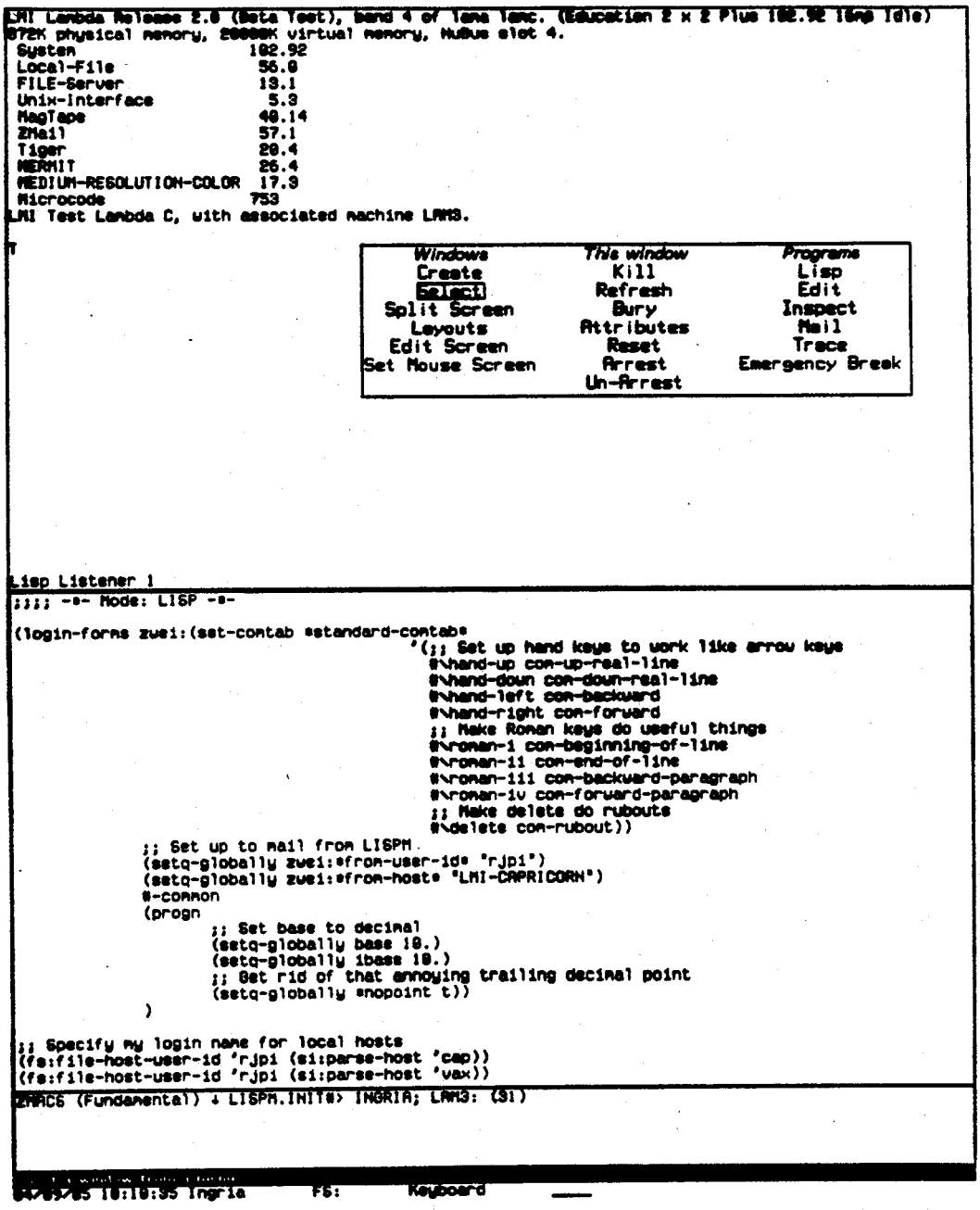


Figure 1. Lambda high resolution screen display

the current process (sometimes called the ***run state***). The area on the far right displays different information depending on the circumstances. Usually the area is empty. However, when you are reading a file from disk or writing to disk, it will display the filename and the progress of the operation. The filename will also be preceded by a ← if you are reading it in or a → if you are writing it out. When the mouse hasn't been clicked or a key pressed for at least 5 minutes, it displays the message **Console idle for *n* minutes.**

Below the who line to the middle right are three small lines called ***run bars***; they can give you a sense of what the Lambda is doing. You will see the leftmost one when garbage collection occurs, the middle one during disk accesses, and the rightmost bar while any program is being run. You should see the middle bar when reading and writing files; if you see it too frequently at other times there may be a problem with your machine. You should see the rightmost one quite often. One way to tell if your machine has crashed is to move the mouse rapidly and look for the run bar to appear. If you don't see anything, your machine is probably down. You can confirm this by seeing if the clock in the lefthand corner has stopped.

1.1.2 The Color Options

Two color options are available: a high resolution system and a medium resolution monitor. The color monitor can be used in either a color or a black-and-white mode. It is designed so that you can distinguish different layers of an image. This makes it useful for both CAD/CAM and AI vision-oriented work. The medium resolution monitor also supports a "frame-grabber" option. (For complete documentation of the medium resolution monitor see *LMI Lambda Medium Resolution Color Board and Monitor Manual.*)

1.1.3 The AI Keyboard

The AI keyboard that comes with your Lambda has many more keys than a typical typewriter or terminal keyboard; it can be quite imposing when you first encounter it. There are three types of keys on the keyboard: character keys, modifier keys, and function keys.

Character Keys

These are the small grey keys in the middle of the keyboard (white key area of Figure 2). They include standard typewriter keys as well as other keys that produce mathematical symbols, or have special uses in certain programs.

Modifier Keys

The modifier keys are the blue keys near the bottom on the right and left with black lettering (dark grey key area of Figure 2). Almost all of these work like shift keys: you press one of these keys together with a character key to modify the behavior of the character key. For instance, the **TOP** modifier key and a letter key will produce the symbol printed above the letter on the character key.

The two exceptions to this are the small **CAPS LOCK** and **ALT LOCK** keys on the left and right respectively. **CAPS LOCK** acts like the Shift Lock on a typewriter, except that it shifts only the letter keys. For instance, if you activate the **CAPS LOCK** and then press the "1" key you will get a "1" (one), not an exclamation point. **ALT LOCK** currently has no effect.

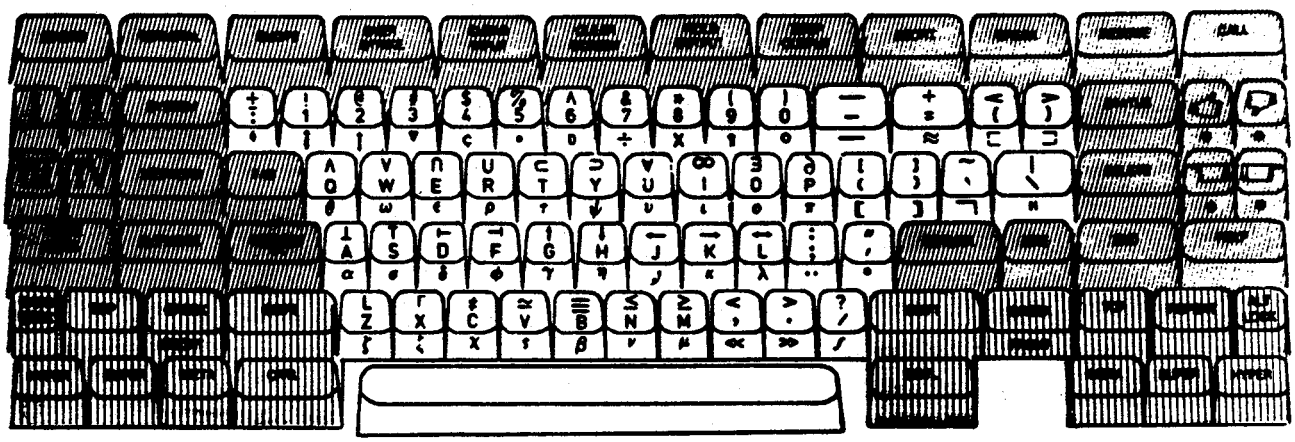


Figure 2. The AI Keyboard

SHIFT is a synonym for **HYPER**. Any command that requires you to press the **HYPER** key will work if you press **SHIFT** instead. The opposite is not true, however. A command that requires you to press **SHIFT** will not necessarily work if you press **HYPER** instead.

Function Keys

The function keys are the large blue keys with white lettering (the light grey key area of Figure 2). Some, like ⟨TERMINAL⟩ and ⟨SYSTEM⟩, are used throughout the Lambda environment; others are used in specific programs, and still others are currently not assigned to any function. These keys will be discussed more fully in the context of individual programs. (Appendix B contains a reference guide to the function key commands.)

Some function keys take arguments. To type such a sequence, press the function key and then the argument. For example, to type ⟨TERMINAL⟩ **S**, first press ⟨TERMINAL⟩ and then **S**.

Use the ⟨RUB OUT⟩ key to delete single characters to the left. The ⟨DELETE⟩ key is currently unassigned.

1.1.4 Mouse and Optical Pad

The *mouse*, like the keyboard, is an input device, a way for you to speak to the computer. LMI uses an optical, rather than a mechanical, mouse because they are much more reliable (the wheels don't fall off... there aren't any). The mouse's companion is the optical pad, a rectangular silvery mat with blue and grey grid lines. The little eyes on the bottom of the mouse use those lines to figure out where it is. You should orient the mouse pad so that the blue lines point up and down.

The mouse can transmit information to the computer in two ways: through the three buttons

on its back, and by changes of position on the optical pad. Information from the buttons usually calls up or chooses items from the pop-up mouse menus. The mouse documentation line usually tells which buttons are active and what they will do. Position information is indicated on the screen by a small arrow called the mouse cursor. Programs such as ZMACS and the **Edit Screen** option on the System Menu use this to allow you to move text and windows around the screen. (For more information on the Mouse and pop-up menus see Chapter 6.)

1.2 Software Features

The Lambda system includes software tools that together create a sophisticated programming environment. These include:

- ZetaLISP-Plus, a powerful and flexible dialect of LISP that includes Flavors, and Common LISP compatability.
- ObjectLISP, an alternate object-oriented programming tool. (optional)
- ZMACS, a text editor tailored to take full advantage of the Lambda environment.
- The Window System, a user-friendly, yet unobtrusive user interface.
- The Error Handler and the Inspector, two high-function debugging aids.
- UNIX, a popular operating system used primarily in academia and software development. (optional)
- Other Languages: (optional)
  - Prolog
  - Fortran
  - C

The greatest strength of the LMI Lambda lies in its modeless operating environment. All the software is instantly accessible; you can never get "stuck" someplace that you don't want to be.

As you use your Lambda you will develop a work pattern that best suits your own preferences and programming needs. Some people prefer to do most of their development, in the LISP Listener and transfer code to an editor buffer once it works; others prefer to develop code in ZMACS and go to a LISP Listener periodically to test it out. The Lambda programmming environment supports both of these styles equally well.

In addition, you may find that, even if you have developed a style of LISP programming on another machine, you may want to alter it to take advantage of features of the Lambda environment. For instance, it is much easier to compile ZetaLISP code than it is to compile code in many other languages, or dialects of LISP: compiled code is immediately runnable and does not require a linking step. This means that many people prefer to compile their ZetaLISP functions, rather than run them interpreted and thereby take advantage of the error checking done by the compiler. Also, compiled and interpreted code run together seamlessly, so that you don't have to commit yourself totally to one or the other.

Another technique that can be combined with the above options is that of splitting the screen and using an editor in one window, and a LISP Listener in another. If you are doing graphics applications you may want to vary this and send your trace and error messages to a different window than the Listener that you use for display. This way, you can debug your program and see its output simultaneously.

You will discover the arrangements that work most effectively for you; the most important thing to remember is that the Lambda provides the tools for customizing your programming environment, but you must take advantage of these facilities and use them.

2. Starting Up

2.1 Booting the Lambda

To "boot" a computer means to clear out memory and restart the machine with the default environment. If you are from a mainframe environment, you have probably never done this unless you are a computer operator or a systems programmer. If you come from the microcomputer world, you do this every time you start the machine. The Lambda fits in between these two groups, so, depending upon your site, you may have to boot every time you use the machine or only very rarely. The information below explains the standard procedure for booting your Lambda.

There are two main reasons to boot the Lambda: either your machine has crashed, or you want a clean LISP environment to work with. The method explained below that allows you to boot from the user console may not work if your system has had a serious crash. If so, you will need to follow the complete booting instructions for your Lambda. For detailed instructions see the *Release Notes* included with your most recent software release.

2.1.1 Cold Booting

To cold boot press all five of the following keys at once.

CTRL-META-CTRL-META-RUB OUT

Note that you need to press *both* **CTRL** and **META** keys.

After you boot, you will notice that there are active run bars in the upper right quarter of the screen; they will move down to their normal position during the booting process. Do not use the keyboard or the mouse until the run bars have returned to their usual location in the who line and the message **Cold-booted** appears.

You can also cold boot from a LISP Listener by evaluating the form:

```
(disk-restore)
```

This is a clean interface for booting if you simply want to get a fresh Lambda environment. You will be queried with a question like the following:

```
Do you really want to reload LODn? (Yes or No)
```

Type **Yes** if you really want to reboot and **No** if you do not.

2.1.2 Warm Booting

A warm boot resets the software, but does not clear memory. The only time to use this is when the machine crashes, or is behaving strangely and you can't save your files. Once you warm boot, save your files immediately and then cold boot. Warm booting does not leave the Lambda in a very stable condition. For example, unwind-protection, which provides a clean exit to functions, even if they are halted abnormally, is not guaranteed to work if you warm boot.

To warm boot press the following keys all together.

CTRL-META-CTRL-META-⟨RETURN⟩

2.2 How and When to Log In

Under many circumstances, especially while you are learning to use the Lambda system, you will not need to log in. Since each Lambda processor is a single-user, low-security system, anyone can walk up to a machine and just begin using it. You do not need to have an account set up for you. You cannot carry out file operations unless somebody (anybody) is logged in, but if you encounter such a situation the Lambda will prompt you to log in and then carry out your request.

The main reason to log in is to set any options that you have placed into your LISPM.INIT file. This file can help you set up the Lambda environment to your satisfaction. A sample file with some useful options is in Appendix A.

To log in, you should get to a LISP Listener (use ⟨SYSTEM⟩ L if necessary), then type:

(login *username***)**

username can be any name that makes you happy. *username* can be either a quoted symbol or a string. It is usually preferable that it be a string. If you don't have a LISPM.INIT file, the system will look for one and get very unhappy if it can't find one; to prevent this, log in like this:

(login *username* **t)**

To logout type:

(logout)

When a new person logs in, any previous user automatically gets logged out by the Lambda.

3. Directories, Filenames, and Pathnames

3.1 Directory Creation

You will probably want to have your own directory to store files in. If you want the Lambda to be able to locate your LISPM.INIT file, the name of your directory and your username should be identical.

Again, it is not strictly necessary to have your own directory; you can store files in any directory you want, since this is an open system, but it does help to organize matters. To create a directory, type:

(fs:create-directory *dirname*)

(*dirname* should be a string that ends with a semicolon, for example **"jonah;"**.) **Note** that if you don't specify a machine, the directory will be created by default on the associated machine.

You can also create a directory from ZMACS, with the command **META-X Create Directory**.

You can create a subdirectory by typing:

(fs:create-directory *dirname.subdir*)

You can delete a directory you no longer want with the **D** command in Dired. See page 21 for more details.

3.2 Filename Syntax

Filenames on the Lambda file system have the following form:

host:*directory*;*name.type*#*version*

host is the physical machine where the file is stored. ***directory*** is the directory on the host where the file is located. Subdirectories are allowed and are indicated by the use of a period (".") *name* is the first part of the filename proper of the file; this will usually be some descriptive name. *type* indicates the type of file this is, such as **LISP** for a LISP file or **TEX** for a TEX file. ***#version*** indicates the version number of the file.

The syntax just described is that of the internal file system of LISP Machines. There are similar syntaxes for other file systems. If you have a Lambda/Plus, you will need to know about the syntax of UNIX filenames. UNIX filenames do not support types or version numbers. In addition, *directory* will consist of a series of names separated by slashes ("/"). Directories in UNIX are terminated with "/" rather than with ";". For further discussion of UNIX directory structure, see Section 8.2 on page 81.

3.2.1 Dired

The Dired facility (for *DIR*ectory *ED*it) is used to examine and modify a directory. Dired can be used to examine a directory on either the local machine where you are working or on a remote host accessed over a network. Dired can be invoked in two ways:

- From the LISP Listener, by the function:

dired &optional *pathname* Function

pathname should be a string. If *pathname* is omitted, the host and directory of the last file you loaded, or attempted to load, will be accessed.

- From ZMACS by the extended command **META-X Dired**. ZMACS will then prompt for a directory name and will indicate a default choice. Once you have specified a directory name, it should be terminated with (RETURN). (If you wish to examine the default directory that ZMACS presents, simply press (RETURN).)

For directories on a LISP Machine file system, *pathname* will usually consist of the name of the host, delimited by a ":" (colon), followed by the name of the directory, delimited by a ";" (semi-colon). For example, if you wanted to edit the directory **INGRIA** on a Lambda named **LAM3**, you would type the following:

```
(dired "LAM3:INGRIA;")
```

For directories on a UNIX file system, *pathname* will consist of the name of the host, delimited by a ":" (colon), followed by the name of the directory, delimited by a "/" (slash). Since / is a special character to ZetaLISP, you will need to "quote" it, with another /, if you invoke Dired via the **dired** function. For example, if you wanted to edit the directory **/lmi/mrc/** on a UNIX system named **CAP**, you would type the following:

```
(dired "CAP://lmi//mrc//")
```

"/"s in UNIX directory names do not need to be "quoted" if you invoke Dired from ZMACS.

Wildcards are also permitted in filenames. Wildcards substitute for portions of a filename. The "*" wildcard may be used in both Lambda and UNIX filenames. "*" substitutes for any number of characters. For example,

```
(dired "LAM3:INGRIA;L*")
```

will display all the files in the directory called **INGRIA** on the Lambda called **LAM3** beginning with the letter "L", while

```
(dired "CAP://lmi//mrc//l*")
```

will do the same for the directory called **/lmi/mrc/** on the UNIX system called **CAP**.

The Lambda file system also accepts the wildcards "<" and ">" in place of *version*. "<" means to look for the lowest version of the file; ">" means to look for the highest. For example,

```
(dired "LAM3:INGRIA;*.*#>")
```

will display the most recent version of each file in this directory.

The UNIX file system also accepts the wildcard "?". "?" substitutes for exactly one character. For example,

```
(dired "CAP.//lmi//mrc//?at.l")
```

will display all the files in the directory **/lmi/mrc/** that consist of a single character followed by **at.l**. See page 83 for further discussion of UNIX wildcards.

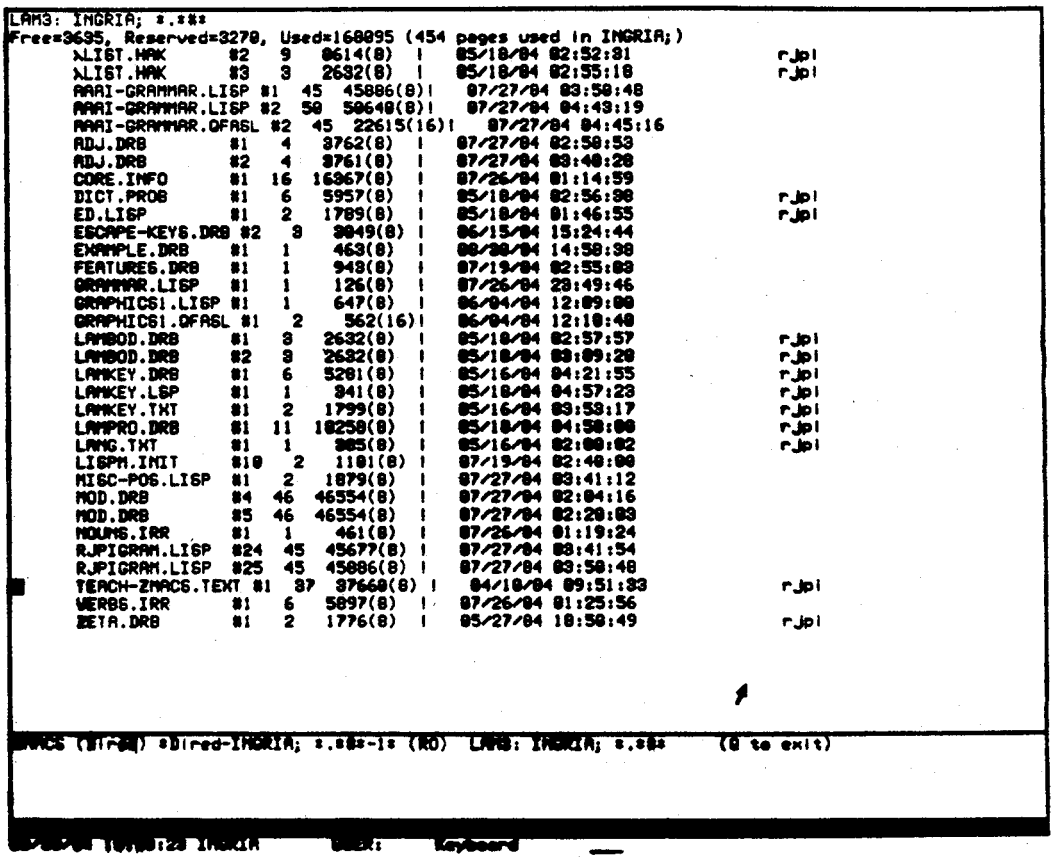


Figure 3. Dired display

Whether you invoke Dired from the LISP Listener or from ZMACS, a special **Dired** buffer, like that shown in Figure 3, will be created in ZMACS. The top line of this buffer will contain the name of the host and the directory as well as any other information that the host file system normally presents. Below this will be an alphabetically ordered list of all the files in that directory. In addition, each file will be followed by the following information:

- The version number of the file (on file systems that support version numbers).
- The size of the file in blocks, pages, or some other (file-system-dependent) measure.
- The length of the file in bytes followed by the size of each byte (in bits), enclosed in parentheses.
- Indicators of various flags that have been set in the file specification:

  ! If the file has not been backed up on tape.

  $ If the file has been marked as protected from being "reaped" or "migrated", i.e. moved off disk to tape if it has not been accessed recently.

@ If the file is protected from deletion.

- The creation date and creation time of the file.
- The date the file was last read, enclosed in parentheses. (This is file system dependent.)
- The username of the creator of the file.

Dired allows you to perform various operations on the files in the specified directory. A full list of them will be displayed if you type ⟨HELP⟩ **M**. Dired commands are single characters and affect the file on the line where the cursor is currently positioned. To move the cursor forward one line in a Dired buffer, type ⟨SPACE⟩ or **CTRL-N**; to move backward, type **CTRL-P**. Some of the more useful commands are:

D
K
CTRL-D
CTRL-K Mark the file for deletion. Actual deletion takes place when you exit Dired. After you have marked the file for deletion, a **D** will appear in the left margin on the same line as the file.

U Undeletes the file.

P Prints the file on the default local hardcopy device.

C Copies the file. ZMACS will prompt you to enter a filename, which should be terminated with ⟨RETURN⟩.

R Renames the file. ZMACS will prompt you to enter a filename, which should be terminated with ⟨RETURN⟩.

E Edits the file. ZMACS will create a new buffer and read the file into it.

V Views the contents of the file. This is useful is you want to see what is in a file without changing the contents. If the contents of the file are too big to fit on a single screen, display will stop at the end of each screenful, and **--More--** will be printed at the bottom of the screen. Typing ⟨SPACE⟩ will cause display to advance; typing ⟨RUB OUT⟩ will cause display to stop and will return you to the Dired display. **--Bottom--** will be displayed when the end of the file is reached. Type ⟨RUB OUT⟩ to return to the main Dired display.

You can exit from Dired by typing **Q**. If you have marked any files for deletion, ZMACS will list them in a typeout window and present you with the options: **Delete? (Q, E, Y, or N) Y** deletes the files but does not expunge them; **E** deletes and expunges the files. (Some file systems, such as that on Tops-10 and Tops-20 systems, as well as that on the LISP Machine itself, support two stages of deletion: deleted files can be recovered easily and are not physically removed from the disk, while expunged files are physically removed and cannot easily be restored.) **N** returns you to Dired. **Q** aborts from Dired. Similarly, if you have marked any files for printing, you will be presented with a list of them and will be queried: **Print? (Q, E, Y, or N)**. These options have the same meanings as they do in case of deleting files.

3.2.1.1 Editing File Systems

It is possible to edit the entire Lambda file system by using the ~ wildcard. For example,

```
(dired "LAM3:~;")
```

will display all the directories on a Lambda named LAM3.

You will be placed into a Dired buffer listing all the directories on the Lambda. You can operate on directories just as you can on files, by positioning the cursor on the same line as the directory. The relevant commands for a directory are:

S Inserts the files of the directory into the Dired display. To remove the files from the display, type **S** once more. The question:

Remove subfiles of *directory*? (Y or N)

will appear in the Echo Area at the bottom of the ZMACS window. Do not be frightened by this question! The files will be removed only from the display, not from the physical file system. Type **Y** if you wish to remove the files from the display, **N** if you do not.

D
K
CTRL-D
CTRL-K Mark the directory for deletion. You should delete all the files in a directory before you mark the directory itself for deletion.

You can also edit the file system on a UNIX system. UNIX directories are arranged in a tree, with the root "/", as discussed below on page 81. For example,

```
(dired "CAP://")
```

will display all the directories on a UNIX machine called CAP.

3.3 Pathnames

In the previous section, we have discussed the representation of filenames in two particular filesystems: those of the LISP Machine and UNIX. In addition to such literal *namestrings*, ZetaLISP allows you to represent filenames as *pathnames*, which are a file-system-independent representation. Pathnames are particularly useful if you are using your Lambda on a network and will be accessing files on different machines with different file systems. You may never need to express a filename as anything but a literal namestring, but you should know that the pathname representation is available, in case you should ever need it. Pathnames are discussed in Chapter 24 of the *LISP Machine Manual*.

4. Using the LISP Listener

After you have booted your Lambda, the first environment the computer will present you with is a LISP Listener. You should see the words

Lisp Listener 1

in the bottom left corner of your console screen. (If you are not using a freshly booted Lambda, then your screen may not look like this. Get yourself a fresh LISP Listener by typing SYSTEM CTRL-L. Your LISP Listener will have a number higher than one, but you will be able to follow the material in this chapter equally well.)

The LISP Listener is the Lambda sub-system that probably most closely resembles other LISP systems you may have used. You can type LISP expressions to it and it will evaluate any expression as soon as it is completed. On the most basic level the LISP Listener consists of a read-eval-print loop. You can use it in this simple way or you can take advantage of the many additional features that ZetaLISP-Plus offers. Some of these additional features are presented and explained in this chapter. You can simply read the material if you want, but most people learn best when they practice, so you will probably get more out of the following material if you sit at the Lambda console and try out the material as it is presented. (Be adventurous! Don't be afraid to try things out for yourself to answer questions that this manual does not cover.)

The cursor in the LISP Listener is a blinking box. The LISP Listener does not have a prompt that indicates when it is ready for new input, but you can still tell easily when it is ready; the run state displayed in the status line will be **Keyboard**. If you type before the Listener is ready for you, the Lambda saves your input and will deal with it when it can.

The default base for numbers on the Lambda is decimal. If you want to change the default radix, the following variables control the base in which numbers are read and printed.

print-base Variable
base Variable

Specifies the radix for printing integers and rational numbers. Valid bases are 2 through 36, inclusive. ***print-base*** is the Common LISP name for this variable.

read-base Variable
ibase Variable

Specifies the radix for reading integers and rational numbers. Valid bases are 2 through 36, inclusive. ***read-base*** is the Common LISP name for this variable.

4.1 Editing in the LISP Listener

Often, you realize in the middle of typing an expression that you want to change something. The ZetaLISP-Plus environment provides many editing features. You can use many of the function keys from the Listener.

CLEAR INPUT
Deletes any expression that you have not yet completed.

RUB OUT Deletes an expression one character at a time.

CLEAR SCREEN
Cleans off the currently selected window but does not affect the LISP environment.

4.1.1 ZMACS Commands

A few ZMACS editing commands are available from the Listener. Use these to edit a LISP expression that you have typed into the LISP Listener. Some of the most useful are:

CTRL-F Moves the cursor forward one character.

CTRL-B Moves the cursor backward one character.

CTRL-D Deletes the character under the cursor.

CTRL-P Moves the cursor up one line.

CTRL-N Moves the cursor down one line.

CTRL-K Deletes the contents of the line after the cursor; if you press this twice, the line will also disappear.

CTRL-META-F
Moves the cursor forward one set of matching parentheses.

CTRL-META-B
Moves the cursor backward one set of matching parentheses.

CTRL-META-D
Deletes the material inside the parentheses.

CTRL-O Opens a new line at the cursor.

⟨STATUS⟩ Prints the first 20 elements of the input history.

META-⟨STATUS⟩
Prints the rest of the input history.

CTRL-C
CTRL-META-Y
Yank a copy of the top item from the input history to the Listener.

META-C Cycles through the input history yanking copies of all items in order.

CTRL-⟨STATUS⟩
Prints the first 20 elements of the kill history.

CTRL-META-⟨STATUS⟩
Prints the rest of the kill history.

CTRL-Y Yanks a copy of the top item from the kill history to the Listener.

META-Y Cycles through the kill history yanking copies of all items in order.

To find out more about editing commands available in the Listener, type **CTRL-⟨HELP⟩**. (For a more complete explanation of ZMACS commands and syntax, see Chapter 5 or the *ZMACS Introductory Manual*.) The *ZetaLISP-Plus Commands* provides a list of the most frequently used ZMACS commands.

4.1.2 Editing Previous Expressions: The input history

Each LISP Listener maintains a list of all the commands that have been typed to it, called the *input history*. Pressing the ⟨STATUS⟩ key will display a list of the last twenty expressions that you have typed to the currently selected LISP Listener. If there are more than twenty, **n more elements in the history**. will be displayed at the end of the list. Typing **CTRL-⟨STATUS⟩** will

display the rest of the input history. Figure 4 shows an example of the display that ⟨STATUS⟩ and META-⟨STATUS⟩ produce.

CTRL-C Reprints the last expression you typed so that you can modify it. This can save you a great deal of repetitive typing when rectifying trivial errors. If you only modify the contents of the LISP expression, without changing the degree of nesting of parentheses, you will need to signal the LISP Listener that you are done. You can do this by typing ⟨END⟩ or by moving the cursor to the end of the form and typing ⟨RETURN⟩. For example, if you accidentally typed:

```
(conss 1 2)
```

the LISP Listener would recognize that there is no **conss** function and offer to place you in the debugger. (See Chapter 7 for more discussion.) If you decline, you can type CTRL-C to retrieve this expression, edit it and then type ⟨END⟩ to cause the Listener to evaluate it.

If you embed the retrieved expression in a new LISP form, the Listener will evaluate it when the expression is syntactically complete, as if it were a newly entered form. For example, to continue our example, if you typed CTRL-C after editting the previous expression, (cons 1 2) would be retrieved. If you then embedded it in a larger expression, such as:

```
(cons (cons 1 2) (cons 3 4))
```

the Listener would evaluate the form when the last) was typed, as if the expression had been entered from scratch.

META-C (Must be used immediately after CTRL-C.) Replaces the previous expression with progressively older expressions each time you use it.

4.1.3 Editing Previous Expressions: The kill history

Commands that delete more than a single character, such as CTRL-K, place the deleted material on a special list called the *kill history*. The kill history is shared among all LISP Listeners and all programs built on ZMACS, such as ZMail and Converse. This makes it useful for transferring expressions from one LISP Listener to another or between a LISP Listener and ZMACS. When you are in a LISP Listener, typing META-⟨STATUS⟩ will display a list of the last twenty expressions on the kill history. If there are more than twenty, ***n* more elements in the history**. will be displayed at the end of the list. Typing CTRL-META-⟨STATUS⟩ will display the rest of the kill history.

CTRL-Y Reprints the most recent expression on the kill history.

META-Y (Must be used immediately after CTRL-Y.) Replaces the most recently yanked expression with progressively older expressions each time.

For more information on the kill history see Section 5.6.

4.1.4 Editing Previous Expressions: Expression Variables

The following are several variables that read-eval-print keeps track of that you can use as shortcuts when typing LISP forms. All of these are active while a form is actually being evaluated by the read-eval-print loop.

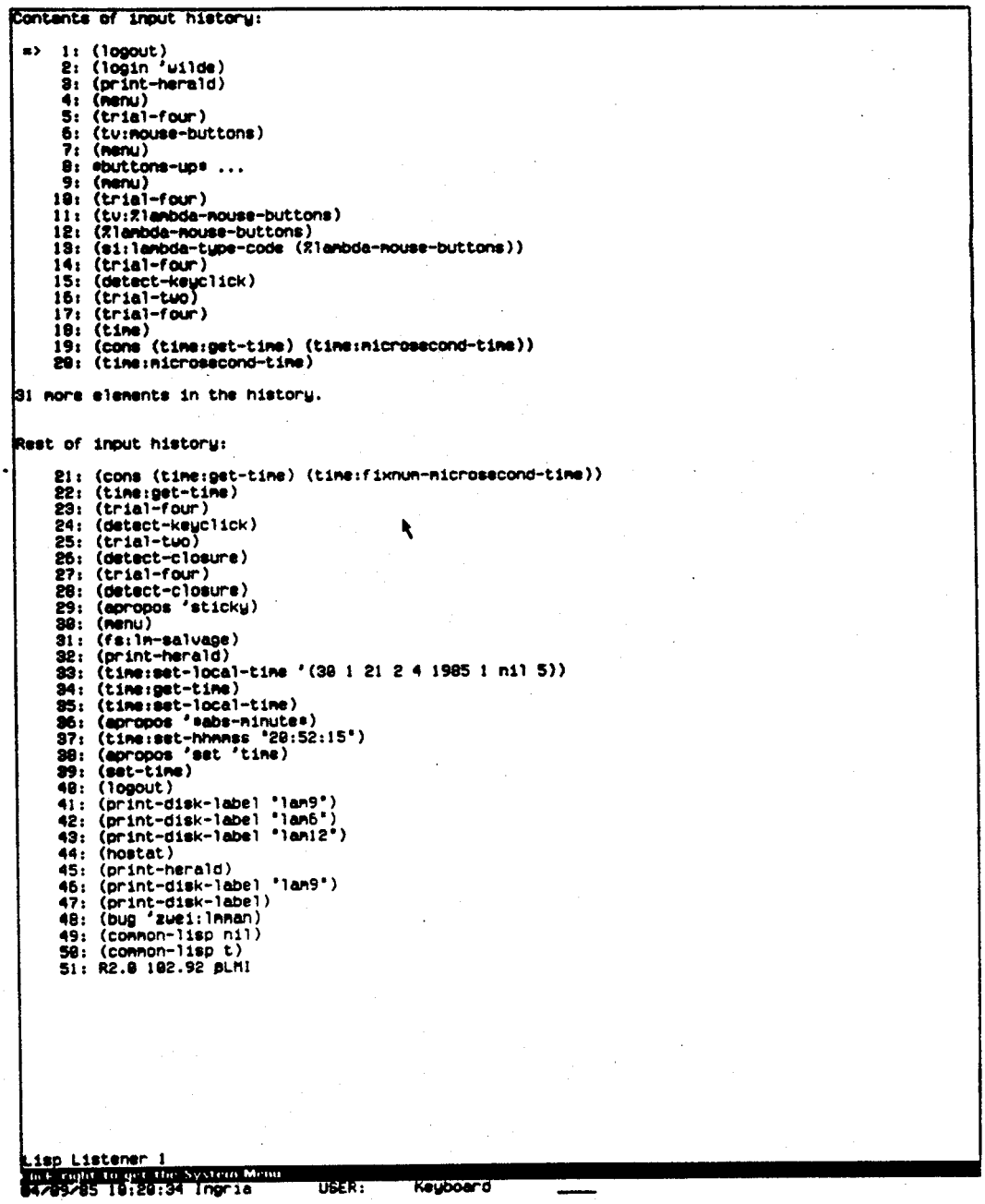


Figure 4. (STATUS), followed by META-(STATUS)

+ Is bound to the previous form read by the loop.

  ++ Holds the previous value of +.

  +++ Holds the previous value of ++.

* Is bound to the result printed the last time through the loop. (If several values were printed, * is bound to the first value.)

  ** Holds the previous value of *.

  *** Holds the previous value of **.

/ Is bound to a *list* of the results printed the last time through. (Since / is a special character to ZetaLISP, you will actually need to type //.)

4.1.5 Printing the Screen

To get a printout of the contents of the screen type (TERMINAL) Q. If you want to print only the currently selected window use (TERMINAL) 1Q. See the *Printer Software Guide* for further discussion of printing screens and windows or saving them to tape or disk.

4.2 Useful LISP Forms

This *Introduction* is not intended to teach LISP programming; however, here are just a few forms that should make programming immediately easier to accomplish. All of the forms in this section are discussed in greater detail in the *LISP Machine Manual*.

print-herald Function

Prints information about the software available on your Lambda and the name of the associated file server. The system calls this as part of the cold-booting procedure, so this is what you see just after booting a machine.

load *pathname* Function

Loads the file specified by ***pathname***, which should be a string. This works properly with both source and compiled (.QFASL) files. If no file type is specified, and a .QFASL version of the file exists, the compiled version will be loaded.

compile *function* Function

Compiles ***function***.

describe *symbol* Function

Tries to give you all the interesting information about the ***symbol***. It does not give array contents. **describe** returns its argument so that you can do something else with it.

who-calls *symbol* Function

symbol must be a symbol or a list. This function tries to find all of the functions in the LISP world that call ***symbol*** as a function, or use ***symbol*** as a variable or constant.

The symbol **:unbound-function** means something special to **who-calls**; if you use this as *symbol*, then **(who-calls :unbound-function)** searches the compiled code for undefined variables or functions. This is useful for finding errors such as misspelled or accidentally undefined functions.

grindef *function...* Macro

Pretty-prints (displays a nicely formatted version of) the definition of ***function***. This always works for interpreted code; it will work for compiled code, but only if the interpreted version was in force at some point. ***function*** is not evaluated, so it does not need to be quoted. **grindef** can be given more than one function name. It can also be given no function name; in this case, **grindef** prints out the definition of the function it was last called on.

4.3 Packages

Every symbol in the LISP environment is (and needs to be) unique. If you redefine a symbol that has already been defined in the environment, the previous definition (or value) gets smashed and yours takes over. This could be very embarrassing if, for example, you redefined something like **setq**. To avoid the problem of different people wanting to use the same symbol names, ZetaLISP-Plus has a feature called the ***package system***. This allows the existence of more than one name space (this is what controls the mapping between printed representations and LISP objects). You can then differentiate between different symbols with the same name by including the package (or name space) prefix. For instance:

pie:blueberry

Is in package **pie**, whereas

jam:blueberry

Is in package **jam**.

At any given time, only one package is the current package. This is the name space where symbols will be looked up. The variable ***package*** is bound to the current package. The name of the current package is normally displayed in the who line. You don't need to use the package prefix for anything that is in the current package. Note that a colon (:) signals a package name to the LISP interpreter and separates it from the name of the symbol.

A freshly booted Lambda always starts out with the **user** package as the current package. As a new user, you will probably need to use only the **user** package. You can also change the current package in several different ways. You can simply use **setq** to change the value of ***package***. ZetaLISP-Plus also provides a convenient interface to set the current package.

pkg-goto *package &*optional *globally* Function

Makes *package* the current package and makes it the value of ***package***. If *globally* is non-**nil**, makes ***package*** the global value of ***package***. Normally, when you change the value of ***package***, either with **setq** or **pkg-goto**, this sets the current package only in the process in which you have changed the value. This does not change the "default" current package for other processes. If you specify that this change should be done globally, ***package*** will be the default package for all processes.

To find out what packages your Lambda knows about, you can check the value of the variable:

all-packages Variable

Is bound to a list of all packages.

Normally, you will need to type only one colon (:) if you use a symbol in one of the Lambda system packages. However, Common LISP package syntax allows packages to distinguish between symbols that are declared external, and which can be referenced from another package with only

one colon, and those that are internal, which should be referenced with two colons. ZetaLISP-Plus does not make this distinction and allows you to refer to symbols with only one colon at all times. However, such symbols will be printed with two colons, so that data produced with a Lambda will be compatible with other Common LISP implementations.

For example, if you were using a package called **graphics**, that contained the two functions **draw-curve** (declared external) and **draw-spline** (internal), you could invoke these functions from another package as:

```
graphics:draw-curve
```

and

```
graphics:draw-spline
```

However, their names would print out as:

```
graphics:draw-curve
```

and

```
graphics::draw-spline
```

You can find full documentation of the Lambda package system in Chapter 27 of the *LISP Machine Manual*.

4.4 Readtables

Common LISP is supported on the Lambda. Most of the functions that have the same names in Common LISP and ZetaLISP behave identically in both dialects. However, some functions, such as **member**, that have been around for a long while and are used in many programs have different meanings in Common LISP from their traditional usage in ZetaLISP. To accommodate such differences, Common LISP functions that are incompatible with their ZetaLISP counterparts are in the cli: package (for *C*ommon *LISP* *I*ncompatible). These functions can be accessed from any program by including the package prefix.

The Lambda also provides a more global mechanism of selecting Common LISP syntax. The *readtable* controls whether symbols are intepreted as in Common LISP or traditional ZetaLISP. In addition to differences in the meanings of certain functions, Common LISP makes \ a special character, while ZetaLISP uses /. The Lambda provides two readtables: the traditional ZetaLISP readtable (this is the default) and a Common LISP readtable. You can switch from one to the other interactively. (LISP Listeners check to see which readtable is in force before evaluating the current expression.) The current readtable is available as the value of the variable:

readtable Variable
readtable Variable

Specifies the current readtable. ***readtable*** is the Common LISP name for this variable.

You can set this variable yourself. However, a safer interface to changing the current readtable is provided by the function:

common-lisp *flag* &optional *globally-p* Function

Specifies the current readtable, based on *flag*:

t Selects Common LISP syntax and function definitions.

nil Selects traditional ZetaLISP syntax and function definitions.

If *globally-p* is non-**nil**, Common LISP syntax and function defintions are selected globally, not just in the current process.

When the Lambda boots up; a new LISP Listener is created; or the readtable is changed; one of the following messages will appear:

; Reading in base *n* in package *pkg* with standard Zetalisp readtable.

if the traditional readtable is in force, or

; Reading in base *n* in package *pkg* with standard Common-LISP readtable.

if the Common LISP readtable is in force.

4.5 Getting a Record of Your Lambda Session

Sometimes it is useful to make a record of your programming session with the Lambda. In ZetaLISP-Plus you can create a "dribble" file that will record all of what you typed and LISP returned. The functions that you need to know about are:

dribble &optional *filename* Function

Opens *filename* as a "dribble" or "wallpaper" file; this records all terminal interaction. Calling **dribble** with no argument terminates dribbling and closes *filename*.

dribble-all &optional *filename* Function

Same as **dribble**, except this also records break loops and debugger sessions, etc.

Note: Information produced by facilities that use temporary or typeout windows for display will *not* appear in your dribble file. (See Section 6.3 for a discussion of these types of windows.)

For more detailed information on these functions see the *LISP Machine Manual*.

4.6 Function Information

Sometimes you want to use a LISP function and don't remember exactly what arguments it takes, or quite what it does. Two LISP Listener commands can help with this. **CTRL-SHIFT-A** prints the function's arguments and **CTRL-SHIFT-D** prints the function's documentation string. To use either, first type a "(" (left parenthesis) to open the expression, and then the function you want information on; follow this with either **CTRL-SHIFT-A** or **CTRL-SHIFT-D**.

CTRL-SHIFT-V is a similar command for variables. Typing **CTRL-SHIFT-V** while the cursor is positioned over the name of a variable will print its documentation string. There are also LISP functions that correspond to these commands:

arglist *function* &optional *real-flag* Function

Returns the argument list of *function*. **arglist** is more versatile than **CTRL-SHIFT-A**, since it can take either a function or a function spec. (See the *LISP Machine Manual*, Section 11.3, pages 223ff. for a discussion of function specs.) The use of *real-flag* is described on page 242 of the *LISP Machine Manual*.

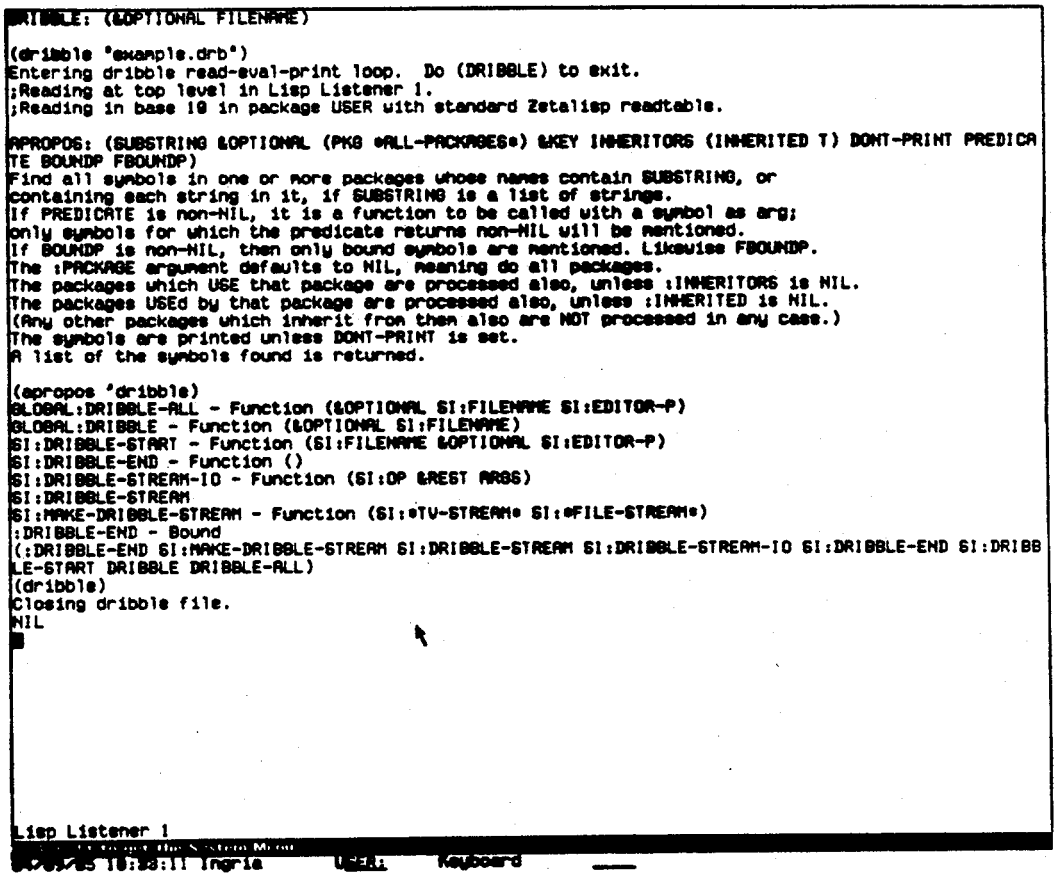


Figure 5. Example of **dribble** function, showing use of information commands

documentation *name doc-type* Function

Returns the documentation string of ***name*** in its use as ***doc-type***. **documentation** is more versatile than **CTRL-SHIFT-D**, since it can return the documentation string of more than functions. Some of the possible values of ***doc-type*** include: **function**, **variable**, **flavor**, **structure**. See the *LISP Machine Manual* for a full discussion of the possibilities.

apropos *substring* &optional *package* &key (*inheritors* **t**) *inherited dont-print* *predicate boundp fboundp* Function

Finds all symbols that contain ***substring*** as a substring. This can be useful if you don't remember the exact name of a LISP form that you wish to use. You can restrict the places that **apropos** searches by using the optional and keyword arguments.

package Is the package to search. If this is not specified, all packages are searched.

:inheritors *t-or-nil*

Specifies that any packages that use ***package*** should also be searched. The

default value is **t**. For example, if you have defined packages called **mypackage** and **myotherpackage**, and **myotherpackage** uses **mypackage**:

```
(apropos "modify" 'mypackage)
```

will search **myotherpackage** as well as **mypackage**.

:inherited *t-or-nil*
Specifies that any package that ***package*** uses should also be searched.

:dont-print *t-or-nil*
apropos normally prints out a list of all the symbols found, along with other useful information, such as the argument list of a function, and returns a list of all the symbols found. If **:dont-print** is non-**nil**, only the list is returned.

:boundp *t-or-nil*
If non-**nil**, only symbols with values will be searched for. For example:

```
(apropos "mouse" 'tv :boundp t)
```

will find only bound variables whose print names contain the substring **mouse**.

:fboundp *t-or-nil*
If non-**nil**, only symbols with function definitions will be searched for. For example:

```
(apropos "mouse" 'tv :fboundp t)
```

will find only functions whose print names contain the substring **mouse**.

:predicate *predicate*
Specifies that only symbols that satisfy ***predicate*** should be searched for. For example:

```
(apropos "mouse" 'tv :predicate #'(lambda (x) (get x 'si:flavor)))
```

will find only flavors whose print names contain the substring **mouse**. (Flavors have the property si:**flavor**.)

Note that you *must* include the optional ***package*** argument to use a keyword argument. ***all-packages*** can be used as ***package***, if you wish to search all packages.

5. Using ZMACS—The Editor

ZMACS is a full-screen editor, based on the EMACS editor developed at the MIT Artificial Intelligence Laboratory. ZMACS is written in ZetaLISP and can be used to edit both text and program source files. It has features designed to aid in writing and debugging LISP code. This chapter provides a brief overview of ZMACS. For a more detailed introduction, see the *ZMACS Introductory Manual*; for full documentation of ZMACS, see the *ZMACS Reference Manual*.

5.1 Additional Information: Teach-ZMACS

In addition to the introductory and reference manuals, which provide written documentation of ZMACS, there is also an online tutorial that introduces ZMACS in an interactive, "hands on" manner. This tutorial can be accessed in two ways:

- In a LISP Listener, evaluate the form **(teach-zmacs)**.
- In ZMACS, execute the extended command **META-X Teach ZMACS**. That is, simultaneously hold down the **META** and **X** keys and then type **Teach ZMACS**.

In order to use teach-ZMACS you need to have a directory; if you have not yet created one, see the information in Section 3.2.1.

5.2 ZMACS Background Concepts

5.2.1 Accessing ZMACS

You can invoke the ZMACS editor in three different ways:

- By typing (SYSTEM) **E**.
- By evaluating the form **(ed)** in a LISP Listener.
- By clicking (R) to cause the system menu to appear and then clicking (L) Select and selecting Editor.

A ZMACS window will then be selected (or created and then selected, if one does not already exist) and you will be in a ZMACS buffer. This will look similar to the buffer displayed in Figure 6; however, if this is your first time during a session that you have entered the editor, the text area of the buffer will be empty.

The largest part of the buffer is devoted to the editing area proper. When you read in the contents of a file, they will be displayed in this area. If you create a new file, using an empty buffer, the text that you enter will appear in this area.

Below this, and separated by a line that runs the full width of the window, is an area consisting of the *Mode Line* and *Echo Area*. The Mode Line displays useful information about the selected buffer. This information includes:

- The mode that ZMACS is in. The default mode for a new buffer created in ZMACS, if no mode is specified, is LISP Mode. See the next section for a discussion of ZMACS modes.

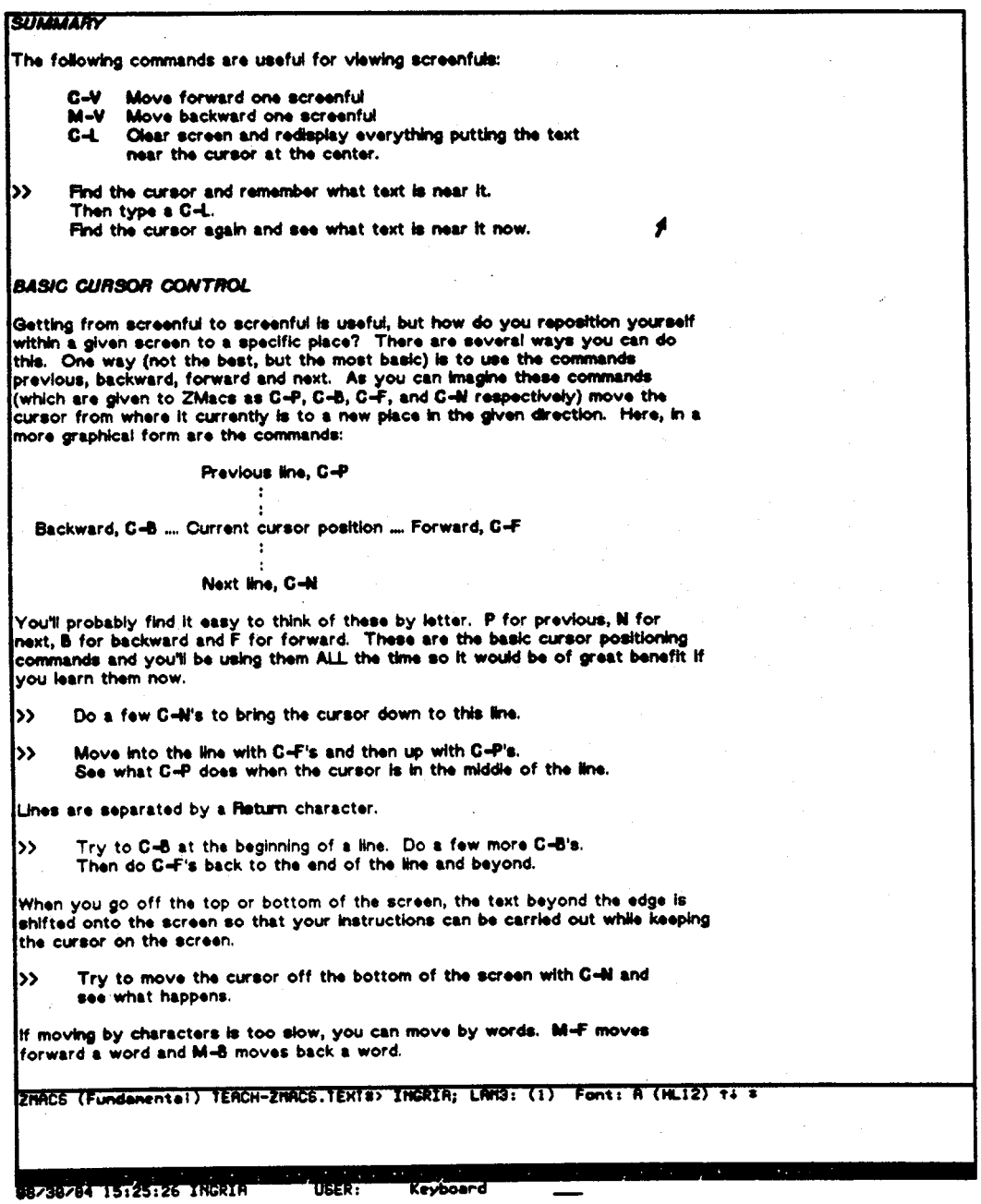


Figure 6. A sample ZMACS buffer

- The name of the currently selected buffer. Normally the name of the buffer will be the same as that of an associated file and will appear in the following form:

  name.type#version directory; host:

  The meaning of these terms is as described above on page 17.
- The version number of the file, enclosed in parentheses.
- The position of the displayed text in the total text of the selected buffer. ZMACS uses arrows to indicate what portion(s) of the buffer are not being displayed. ↑ indicates that there is text above; ↓ indicates that there is text below; ↑↓ indicates that you are in the middle and that there is more information both above and below in the buffer.
- The status of the selected buffer. If * appears in the mode line, the buffer has been modified and should be saved.
- The default font, if you have specified fonts for this buffer. See page 48 for more discussion of fonts.

The Echo Area takes up the remaining lines of the window. ZMACS commands that you type are printed out or "echoed" in this area. If you complete a command quickly, it will not be echoed. However, if you hesitate over a command, it will be echoed here. This echoing facility is useful for novices but will not get in the way of experienced users.

The Echo Area is also the portion of the window where **META-X** extended commands will appear; see page 38 below for discussion.

5.2.2 Major and Minor Modes

ZMACS contains information about many of the types of files that it can be used to edit and provides useful operations for each of these types. Such collections of information are called *major modes*. For example, there are different modes for various programming languages, such as PL1 and LISP, of course. For each language, ZMACS has information on the conventions for formatting code in that language, the comment character, etc. Similar information exists for various text formatters, such as TEX. There is also a Text Mode, for text files that are not designated to be formatted by any particular text formatter.

In addition to major modes, such as LISP Mode or TeX Mode, there are also *Minor Modes* that control more specific aspects of the behavior of ZMACS. For example, the minor mode Auto Fill Mode automatically inserts new lines as you type, so that you do not need to explicitly type (RETURN).

You can specify the major mode that ZMACS should use to edit a particular file in either of two ways:

- By explicitly setting the mode with a ZMACS extended command consisting of **META-X** followed by the name of the mode. For example, to get into TeX Mode, you would type **META-X TeX Mode**. You can use the command completion facility of ZMACS when typing mode changing commands; see page 38 below for a discussion of command completion.
- By specifying the mode of the file in an *attributes list* at the beginning of the file. Such attributes can include the Major Mode of the file, the Package the file belongs to,

the Base, etc. The file attributes list should be the first non-blank line in the file and the attributes list should be delimited by **-*-**. Also, in order to prevent the attributes list from being treated as code to be executed or text to be formatted, it should be preceded by a comment character. An example of an attributes list for a LISP file is:

```
;;;; -*- Mode: LISP; Package: GRAPHICS; Base: 8 -*-
```

You can also specify the readtable a particular file is to use by including **Readtable:** in the file attributes list. It can have either of two values:

Common-LISP
: Selects Common LISP syntax and function definitions.

Traditional
: Selects traditional ZetaLISP syntax and function definitions.

- If you do not specify the major mode explicitly via the file attributes list or a **META-X** command, ZMACS uses the file type as an implicit specification. If it is one that ZMACS recognizes, such as **.LISP**, it will be used to set the major mode. If none of these methods specifies a major mode, the default, **LISP mode**, will be selected.

5.3 Inserting and Deleting Text and Positioning the Cursor

Many text editors have separate "modes" for inserting text, deleting text, and positioning the cursor. (These modes are different from the major and minor modes of ZMACS.) An editor with modes uses them to distinguish the effects that typing a particular key can have. For example, a mode-oriented editor might assign the following meanings to the act of typing the "**f**" key:

in positioning mode:	move the cursor forward one position
in deletion mode:	delete the next character
in insert mode:	insert the letter "f"

In ZMACS any of these operations can be done at any time. In this sense, then, ZMACS is a "modeless" editor.

5.3.1 Control Character Commands

Since ZMACS does not use modes to distinguish between these different functions, there must be some other way to signal ZMACS to insert text, move the cursor, delete some block of text, or perform some other operation. ZMACS accomplishes this with control characters. Normally, typing a key that transmits a printing character (a letter, number, or other character) causes that character to be inserted into the current text being edited. However, simultaneously typing a character key and a modifier key signals ZMACS to perform some other operation. The two modifier keys **CTRL** and **META** are the most frequently used modifier keys in ZMACS, though other modifier keys are also used.

Most commands have a name that is mnemonic of the operation to be performed: for example, **CTRL-F** and **META-F** both move the cursor *F*orward. Typically, the **CTRL** version of a command performs an operation over some small domain, while the **META** version performs it over a larger domain. For example, **CTRL-F** moves the cursor forward one character, while **META-F** moves forward

one word. The following table indicates the most commonly used **CTRL** and **META** cursor movement commands.

Key and Action	CTRL version	META version
F - Move forward	character	word
B - Move backward	character	word
N - Move to next	line	line and add a comment
P - Move to previous	line	line and add a comment
A - Move to beginning	of line	of sentence
E - Move to end	of line	of sentence
] - Move to end	of page	of paragraph
[- Move to beginning	of page	of paragraph

If you wish to execute a ZMACS command more than once, you can prefix it with a numeric argument that specifies the number of times it is to be executed. Numeric arguments are entered by prefixing each number in the entire numeric argument by CTRL. For example, if you wish to move forward 20 lines instead of just one, you would type CTRL-2 CTRL-0 CTRL-N. This would move the cursor down twenty lines. Note that each part of the numeric argument has its own CTRL: typing CTRL-2 0 CTRL-N would cause two 0's to be inserted in the text before the cursor moved down a single line. CTRL-U is a predefined numeric prefix with a value of 4; typing CTRL-U before a command will cause that command to be executed 4 times. Successive CTRL-U's have a multiplicative effect: typing CTRL-U CTRL-U before a command will cause that command to be executed 16 times. CTRL-U can also be used as a prefix to turn numbers into numeric arguments. For example, CTRL-U 20 CTRL-N moves the cursor down 20 lines.

5.3.2 Extended Commands

In addition to single character commands, produced by simultaneously typing a printing character and one or more modifier keys, there are extended commands, produced by typing CTRL-X or META-X. CTRL-X commands are single character extended commands, that is, CTRL-X must be followed by a single character (either a plain character or a single letter control command).

META-X commands are word extended commands, that is, META-X must be followed by one or more words. To make it easier to type word extended commands, there is a *command completion* facility, that will complete different portions of an extended command that has been specified uniquely. Typing (SPACE) will complete the current word of a multi-word command. Typing (ALT MODE) will complete the rest of the command (or, at least, as much as you have uniquely specified) but will not execute it. Typing (RETURN) will complete the current command and cause it to be executed. For example, if you wished to execute the META-X command **Auto Fill Mode** you would first type META-X. The prompt **Extended command:** would then appear in the Mode Line, as shown in Figure 7.

When you type the first two letters of the command, **Au**, they will echo in the Echo Area. You will then have the following completion possibilities:

- Typing (SPACE) will complete the rest of the word causing **Auto** to be displayed in the echo region; you will then be able to type in the rest of the command.
- Typing (ALT MODE) will complete the rest of the command, and will leave the line displayed in the echo area, allowing you time to cancel the command.
- Typing (RETURN) will complete the command and cause it to be executed.

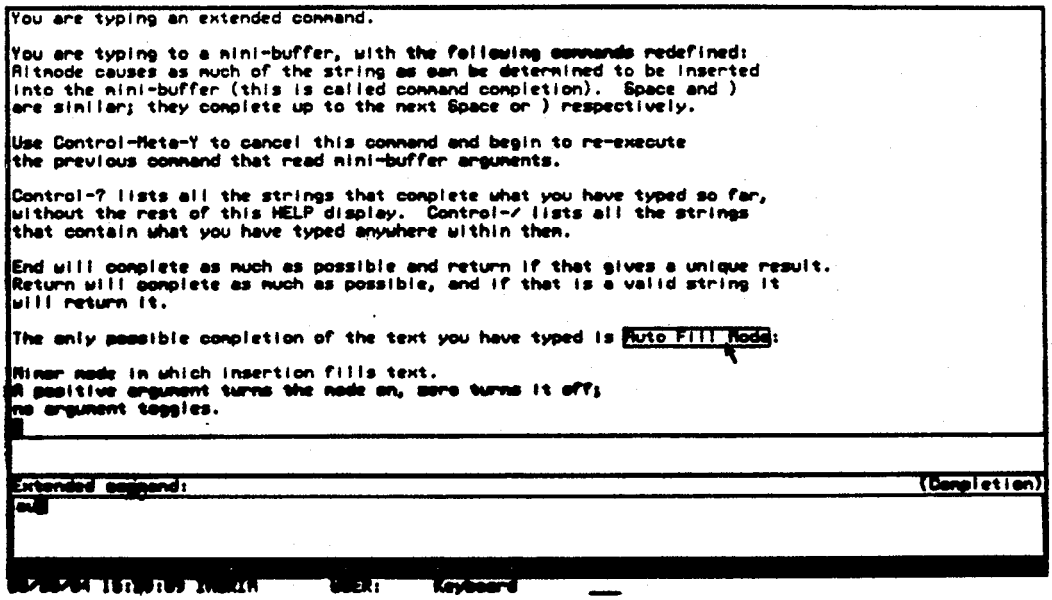


Figure 7. META-X Mode Line, Echo Area and typeout window

At any time you can find out the possible completions of what you have typed by pressing ⟨HELP⟩. The choices will be displayed in a typeout window, as shown at the top of Figure 7. Choices are mouse-sensitive, so you can select one by using the mouse.

5.4 Buffers and Files

Some of the most commonly used extended commands are for manipulating files and buffers. When you edit a file, you do not physically change the file that is on disk. Rather, the file is copied to a region, called a *buffer*, and the copy can then be altered. If you decide to save the alterations you have made, you can then write the contents of the buffer to disk; if you do not, you can discard them.

5.4.1 Saving and Reading in Files

The normal way to read a file into a buffer in ZMACS is by means of the command **CTRL-X CTRL-F**. You will then be prompted for the name of a file (which you should terminate with ⟨RETURN⟩). This reads a file with the specified name into a new buffer and gives that buffer the name of the file. If no file with the specified name already exists, ZMACS creates an empty buffer with that name.

You may enter filenames in either of two orders:

name.type#version directory; host:

This is the order in which filenames are displayed, as mentioned above on page 35. You can also use the order:

host:directory;name.type#version

If you do not specify a version number and the file system being accessed supports version numbers, the highest version number will be assumed. The wildcards ">" and "<" will also work for files on the LISP Machine file system.

ZMACS will normally display a default filename on the mode line. If the host and directory of the file you want to edit are the same as the default, you do not need to specify them. For example, if the default presented is:

lam3:ingria;main.lisp

and you wish to access LAM3:INGRIA;NEW.LISP, you only need to enter **new.lisp**, not the entire file specification.

If you decide to save the changes you have made to the buffer, this can be done with the command **CTRL-X CTRL-S**. This will save the buffer under the original name of the file that was read into the buffer, but with a new version number. **CTRL-X CTRL-W** is similar; it will save the contents of the buffer, but will prompt you for a new filename.

ZMACS keeps track of whether or not changes have been made to a buffer. If you try to save a buffer that has not been altered, ZMACS will respond:

(No changes need to be written.)

ZMACS can contain several buffers at the same time, each with an associated file. There are several ways of finding out what the current buffers are, moving between them, and performing other manipulations. Some of these facilites are available both as commands and as options in the ZMACS System Menu. (See Section 5.9 for more discussion.)

CTRL-X CTRL-B
List Buffers

List the current buffers, with their associated files, and indicate which of them have been modified and need to be saved. A sample display is shown in Figure 8. Buffer names are mouse-sensitive. Clicking [L] will select the buffer. Clicking [R] will pop up a menu of options.

CTRL-X B Allows you to select one of the current buffers. You will be prompted for a buffer name, which should be terminated with RETURN.

CTRL-X K Allows you to kill a buffer. You will be prompted for the name of a buffer to kill, which should be terminated with RETURN. (If you wish to kill the currently selected buffer, simply type RETURN. You will then be prompted for the name of a buffer to select.)

META-X Kill or Save Buffers
Kill or Save Buffers

Allow you to kill or save several buffers at the same time. A menu of all the current buffers, like the one in Figure 9, will then pop up, with a list of operations. You can select the operation(s) to perform on a buffer by clicking on the box(es) in the aprropriate row(s) opposite each buffer. Clicking on **Do It** ■ will cause each buffer to be saved or killed, according to the specifications you have made. Clicking on **Abort** ■ will escape from the menu, leaving all the buffers as they are.

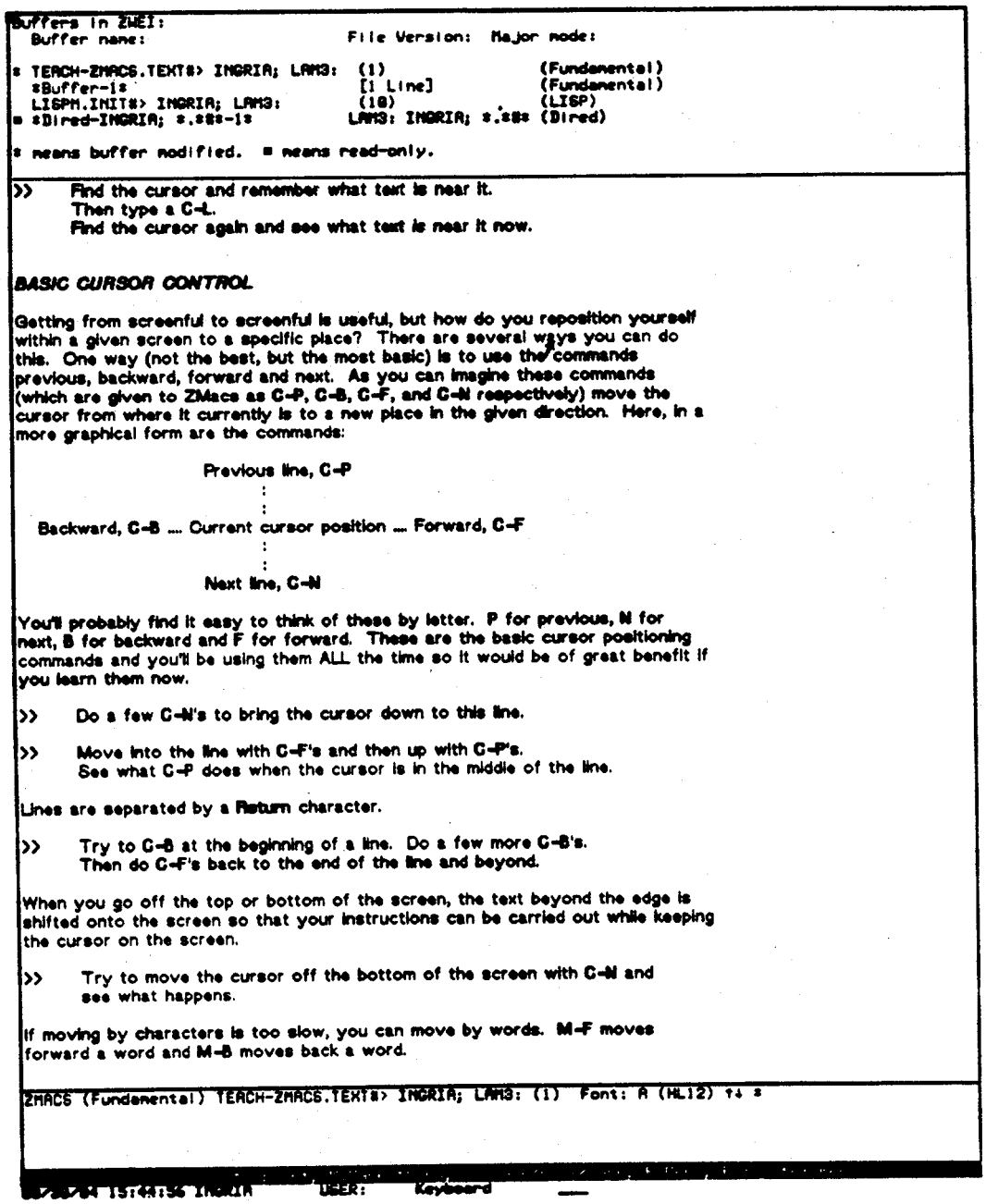


Figure 8. Typeout window from List Buffers

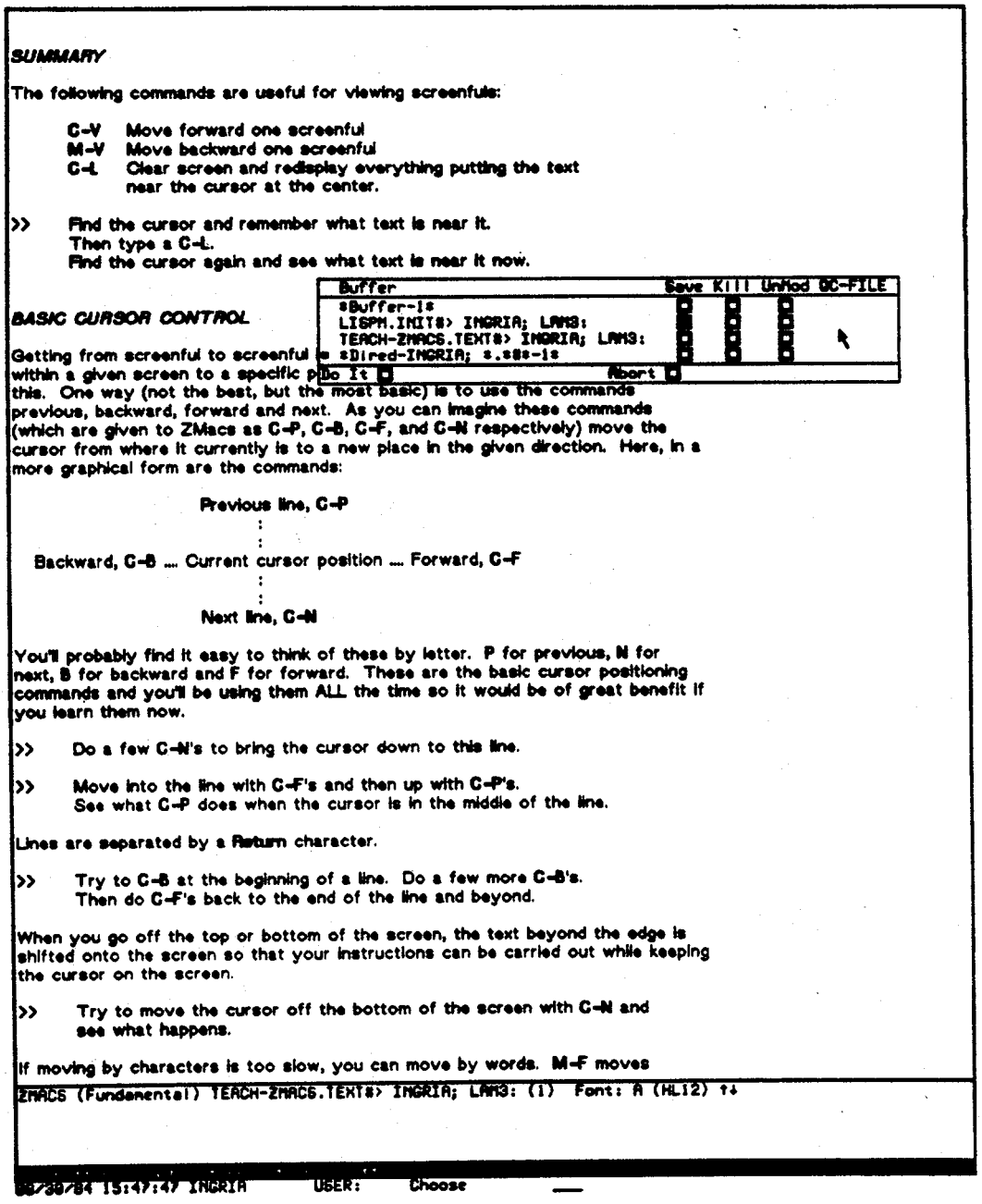


Figure 9. The Kill or Save Buffers menu

META-X Save All Files
Goes through all the modified buffers and ask you if you wish to save the associated files.

5.5 The Undo and CTRL-G Safety Net

There are many commands in ZMACS and it is possible that at some time you may accidentally make an unwanted change to the buffer or start to type in a command and decide that you don't really want to execute it. **Undo** and **CTRL-G** are commands that provide the ability to undo an accidental change or escape from a current command. **Undo**, which can be invoked by **META-X Undo** or ⟨HELP⟩ **U**, will undo the last major change that you have made (except for a "kill", which is undoable in another way; see the Section 5.6 on Killing and Deleting below). Before doing so, you will be given a prompt naming the change to be undone and asking you to confirm that you really wish to undo it.

Undo itself can be undone, if you accidentally undo a change you didn't want to undo or if you decide to keep the results of the last change.

CTRL-G is used to abort an extended command. If you are in the middle of typing a **CTRL-X** command and you decide you really don't wish to complete it, you can use **CTRL-G** to escape from that command. **CTRL-G** must be typed twice to escape from a **META-X** command if you have started to type the extended command. The first **CTRL-G** aborts the string that you have begun to type; the second escapes from **META-X** itself. ⟨ABORT⟩ is similar to **CTRL-G**. However, you need to type only one ⟨ABORT⟩ to terminate an extended command.

5.6 Killing and Deleting

In ZMACS there are two ways of removing text from the currently selected buffer: one way is deleting, the other is killing. Text that is deleted is removed from the buffer and is "gone"; no record is kept of text that is deleted. Text that is killed is placed on a special list called the kill history from which it may later be retrieved. Since there is only one kill history on the Lambda, which can be accessed from any ZMACS buffer or LISP Listener, the kill history can be used to move text from one part of a buffer to another or from one buffer to another, to make multiple copies of a piece of text, and to move text between ZMACS and the LISP Listener. There are only two deleting commands in ZMACS: ⟨RUB OUT⟩, which deletes a previous character, and **CTRL-D**, which deletes the next character. However, if a numeric argument is used with either of these commands, so that more than a single character is removed from the current buffer, ⟨RUB OUT⟩ and **CTRL-D** kill rather than delete text. Common kill commands are:

META-D Kills the next word.

META-⟨RUB OUT⟩
Kills the previous word.

⟨CLEAR INPUT⟩
Kills to the beginning of the current line.

CTRL-X ⟨RUB OUT⟩
Kills to the beginning of the current sentence.

CTRL-K Kills the rest of the current line.

META-K Kills to the end of the current sentence.

5.6.1 The Kill History

The kill history is a list of pieces of text that have been killed. It is possible to retrieve or "yank" back text that is on the kill history by means of the commands **CTRL-Y** and **META-Y**. **CTRL-Y** retrieves the item that is at the top of the kill history, i.e. the most recently killed piece of text, and inserts it in the buffer at the current cursor location. **META-Y** is similar, except that it retrieves and inserts the next-to-last piece of text on the kill history and makes it the current item on the kill history. **META-Y** in ZMACS can only be used after **CTRL-Y** (or **META-Y**).

The kill history maintains a record of all the items that have been killed. **CTRL-Y** with a numerical argument of **0** (zero) (**CTRL-0 CTRL-Y**), will display a menu of the last twenty expression that you have killed. The elements of the kill history are mouse-sensitive. Clicking [L] on one of them will cause it to be yanked into the buffer. If there are more than twenty elements in the kill history, ***n* more elements in the history**. will be displayed at the end of the list. This message is also mouse-sensitive and clicking on it will display the rest of the kill history.

The number of entries that the kill history displays can be altered by changing the value of the variable **History Menu Length**. This is done by means of the **META-X** command **Set Variable**. After typing **META-X Set Variable** you will be prompted for a variable name. Type **History Menu Length** ⟨RETURN⟩. You will then be prompted for a numerical value. Type this, followed by ⟨RETURN⟩. The maximum number of entries that the kill history Menu displays will now be set to the number entered.

The kill history in ZMACS is shared with the LISP Listener. This allows individual ZetaLISP functions to be defined in a ZMACS buffer, placed on the kill history, and yanked into a LISP Listener to be tested. The kill history can also be used to move text from a LISP Listener to a ZMACS buffer.

5.6.2 Regions

It is, of course, possible to kill large portions of text through the use of numeric arguments and commands that kill longer stretches of text, such as **CTRL-K** and **META-K**. However, ZMACS makes this process easier by allowing you to define ***regions*** or large blocks of text that are to be placed on the kill history. Moreover, it is possible to place the text in a region on the kill history without removing it from its present location. This is a convenient facility for those times when you wish to copy a block of text to another part of the current buffer or to another buffer or LISP Listener. You can indicate the text to be moved and then move it to the desired new location(s) without having to kill it and yank it back to its original location.

How to Define a Region

Regions are marked, or *defined*, by indicating the beginning of the text to go into the region and then indicating the end of the text. This can be done with keyboard commands or with the mouse. The keyboard commands **CTRL-**⟨SPACE⟩ and **CTRL-@** (**CTRL-***ATSIGN*) indicate the beginning of region. After marking the beginning of a region with either of these commands, you must then indicate the end of the region by moving the cursor to the desired location. This can be done with keyboard cursor movement commands. A method of defining a region by using only the mouse is to press the left mouse button at the beginning of the region and keep it depressed. Move the mouse cursor to the end of the region and then release the left mouse button. As a region is defined, either by keyboard or mouse commands, ZMACS indicates the region by underlining the

text contained in it, as shown in Figure 10.

Two useful commands for defining large regions are:

CTRL-> Creates a region from the cursor to the end of the buffer.

CTRL-< Creates a region from the cursor to the beginning of the buffer.

Commands Using Regions

Once a region has been defined, there are two commands for placing it on the kill history:

CTRL-W Kills the region and places it on the kill history.

META-W Copies the region onto the kill history without removing it from its current location.

5.7 Commands Especially Useful with LISP

The ability to define regions is useful for moving text around in a single buffer, between buffers, and from a buffer to a LISP Listener. There are other ZMACS commands that are especially useful for defining and testing ZetaLISP functions.

5.7.1 LISP Mode Cursor Movement and Killing Commands

Just as there are commands for moving through and killing various portions of text in Text Mode, there are commands for moving through and killing different portions of LISP code in LISP mode. Among these commands are:

CTRL-META-F
Moves the cursor forward an s-expression.

CTRL-META-B
Moves the cursor backward an s-expression.

CTRL-META-A
CTRL-META-[
Move the cursor to the beginning of the current definitional form; this includes not only **defun**, but also **defflavor**, **defvar**, etc.

CTRL-META-E
CTRL-META-]
Move the cursor to the end of the current definitional form.

CTRL-META-K
Kills the next s-expression.

CTRL-META-RUB OUT
Kills the preceding s-expression.

5.7.2 Formatting and Compiling LISP Forms in ZMACS

In LISP mode, ZMACS will aid you in formatting your LISP code according to standard formatting conventions. An especially useful facility is the treatment of the LINE key. In LISP mode, this will not only start a new line, it will also add the appropriate indentation for the

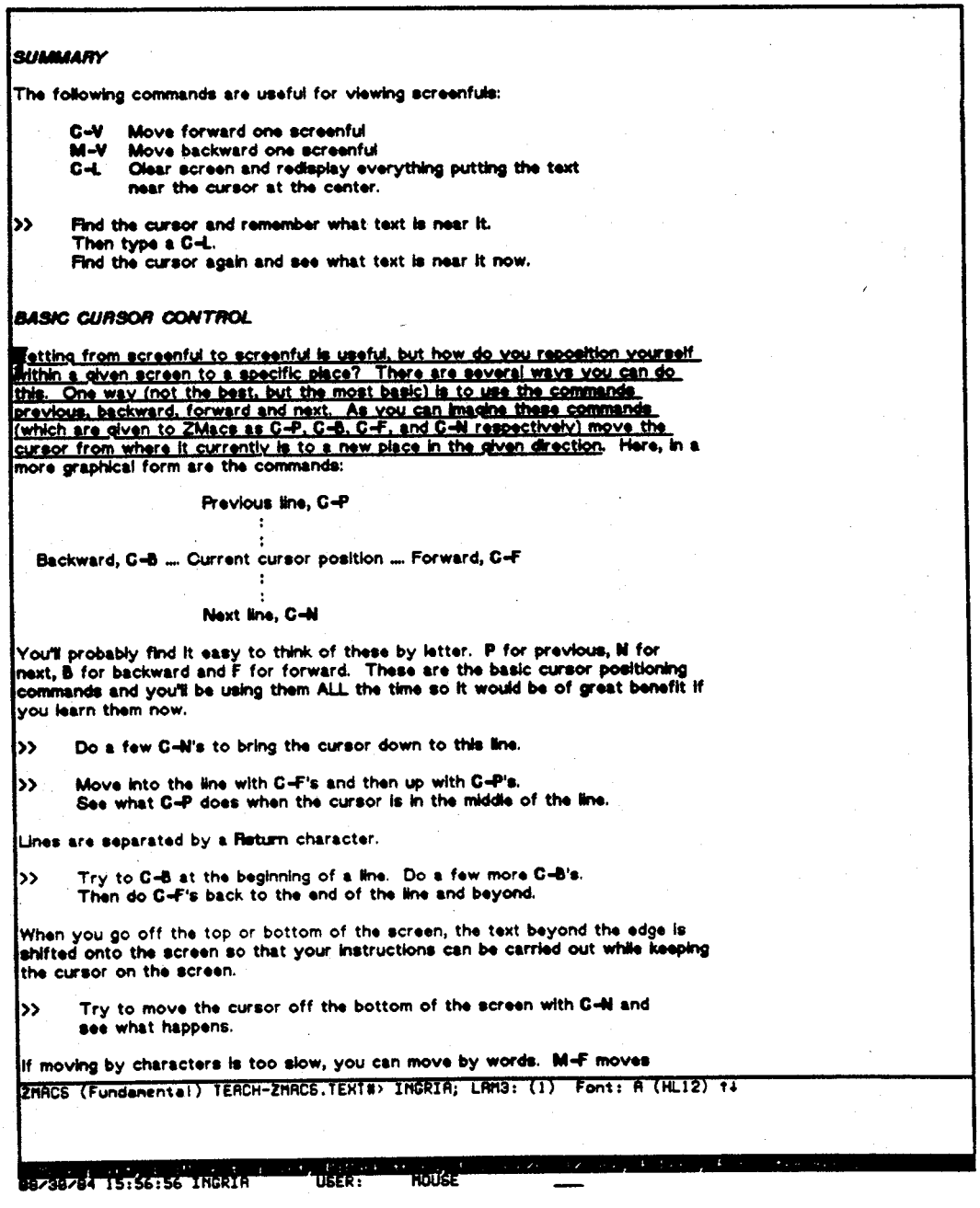


Figure 10. A ZMACS region

current line of LISP code. The command **CTRL-META-Q** will properly indent a LISP form that is not properly formatted.

ZMACS also has commands for compiling LISP forms. **CTRL-HYPER-C** compiles the current region or, if there is no defined region, the current definitional form. **META-X Compile Buffer** compiles the entire buffer. A buffer or region which has been so compiled will be read into the LISP Listener, so that you can use its contents without otherwise explicitly loading it.

A useful way of testing LISP functions that you have defined in ZMACS is to type (BREAK). This will create a LISP Listener typeout window, complete with its own input history, for evaluating LISP forms. Typing (ABORT) will cause this window to disappear and will restore the ZMACS buffer you were previously viewing. If you type (BREAK) later, the new LISP Listener will have the input history of the LISP Listener previously created with (BREAK). Other changes you make in the LISP Listener will also remain constant across invocations. For example, if you set the variables ***print-base*** or ***read-base***, the effects will endure across invocations. However, changing the readtable to Common LISP will not endure. Each time you type (BREAK), the LISP Listener will start with the traditional readtable.

5.7.3 ZTop Mode

Another useful way of testing LISP functions within ZMACS is ZTop Mode, which can be invoked by **META-X ZTop Mode**. This gives the current ZMACS buffer the ability to evaluate LISP forms, like a LISP Listener. The normal ZMACS commands are still available, so you can move the cursor, edit expressions, etc. and use all of the normal ZMACS facilities. ZTop buffers will occasionally display a third cursor, in the shape of an "I-beam", in addition to the normal ZMACS cursor and the mouse cursor. The I-beam cursor indicates that all forms before it have already been evaluated. You should not edit expressions before this cursor, since the edits will not affect evaluation of the form.

A ZTop buffer, like a LISP Listener, has an input history of its own. **CTRL-META-Y** can be used to yank items off of the input history. (Other ZMACS buffers do not have an input history.) You can see the input history of a ZTop buffer by typing **CTRL-META-Y** with a numeric argument of **0**. The items displayed in the history are mouse-sensitive. You can yank an element from the history by clicking [L] on it.

5.7.4 ZMACS Help Functions for LISP Code

ZMACS has various commands that are designed to make writing LISP code easier by providing online, immediate mechanisms for documenting the LISP functions that you are working with. Of these, **META-.** (read as "meta-point", or "meta-period", or "meta-dot", or even "meta-doc") is perhaps the most useful. After you type **META-.**, ZMACS will prompt you for the name of a function, which should be terminated with (RETURN). (ZMACS will present a default function, which is the function to the left of the cursor. If you wish to see this function, simply type (RETURN).) ZMACS will then find the file that contains the definition of this function, read this file into a new buffer, make this buffer the selected buffer, and position the cursor on the specified definition. This is extremely useful for cases where you are editing a function and you find that it calls a function that you do not recognize. You can then use **META-.** to find the definition of this function and, possibly, the definition(s) of functions that that function calls, and so on.

Other useful commands include:

CTRL-HYPER-A
Prints the argument list of the function preceding the cursor.

CTRL-HYPER-D
Prints out brief documentation about the function preceding the cursor.

Both these commands print out in the Echo Area.

HYPER-META-D
Prints out the complete documentation of a ZetaLISP function. **HYPER-META-D** will prompt for the name of the function to be documented (the default is the function to the left of the cursor). Terminate your choice with (RETURN). ZMACS will then display the documentation in a typeout window.

5.8 Fonts

ZMACS allows you to specify different fonts to display the text of your file. You can find out what fonts are available via the command **META-X List Fonts**. This will pop up a list of all fonts. The font names are mouse-sensitive and clicking [L] on the name of a font will display a font sample.

5.8.1 Specifying Fonts for a File

The easiest way to add fonts to a file is with the command **META-X Set Fonts**. You will be prompted to enter a list of fonts in the Echo Area. Terminate them with (RETURN). You will then be asked the question:

```
Change the -*- line of the text as well? (Y or N)
```

If you answer **Y**, the fonts you have specified will be added to the file attributes list of the file and when the file is subsequently read into ZMACS, ZMACS will know what fonts to use.

5.8.2 Changing Fonts

At any time, a buffer with fonts will have a *default font*. This is the font in which newly entered text will be displayed. The default font appears at the end of the Mode Line in the form:

Font: *font letter* (*font name*)

where *font letter* is a single letter, used to select a font in ZMACS, and *font name* is the name of the font. Fonts are associated with font letters based on their position in the fonts list in the file attributes list. For example, given the following fonts list:

```
Fonts: (CPTFONT TR12 TR12I TR12B)
```

CPTFONT would have the font letter **A**, **TR12** would have **B**, etc.

Once you read in a file with a **Fonts:** attribute or use **META-X Set Fonts** to add fonts to a current buffer, you will be able to set the default font and specify fonts for different portions of the text.

ZMACS commands that change fonts include:

META-J Changes the font of the word following the cursor. You can give this a numeric argument to change the font of more than one word.

CTRL-X CTRL-J
CTRL-SHIFT-J
Change the font of the current region. The region must already be defined before you use these commands.

CTRL-META-J
Changes the default font. Newly entered text will be displayed in the font you specify.

You can specify the fonts for all of these commands in any of four ways:

1. By typing a font letter.
2. By typing (ALT MODE), followed by a font name, terminated with (RETURN).
3. By clicking [L] on a character that is already displayed in the desired font.
4. By clicking [R] to pop up a menu of font names. Click [M] on the desired font.

5.9 The ZMACS Main Menu

In ZMACS, clicking [R] will not pop up the System Menu. Instead, it will pop up a menu of its own, like that shown in Figure 11. (Clicking [R] will pop up the main System Menu.) All the entries can also be accessed by ZMACS character or extended commands. As is usual on the Lambda, choose the alternative that suits you best.

Here are the ZMACS Menu options, along with alternate ways of accessing them.

Arglist
CTRL-HYPER-A
Print the argument list of a function. You will be prompted to enter the name of a function in the Echo Area. The default, which is displayed on the mode line, is the function the cursor is over.

Edit Definition
META-. Find the definition of a function, read the file containing it into a ZMACS buffer, and position the cursor at the beginning of the definition. You will be prompted to enter the name of a function in the Echo Area. The default, which is displayed on the mode line, is the function the cursor is over.

List Callers
META-X List Callers
List all the functions that call a specified function in the current package. You will be prompted to enter the name of a function in the Echo Area. The default, which is displayed on the mode line, is the function the cursor is over.

List Sections
META-X List Sections
List all the sections of the current buffer. A *section* is a portion of the buffer containing a **defun** or other definitional form. The sections are displayed in a typeout window. The names of the sections are the names of the defined functions, etc. Each is mouse-sensitive. Clicking [L] on a section will position the definition in the upper left hand corner of the screen, with the cursor positioned at the beginning of the definition.

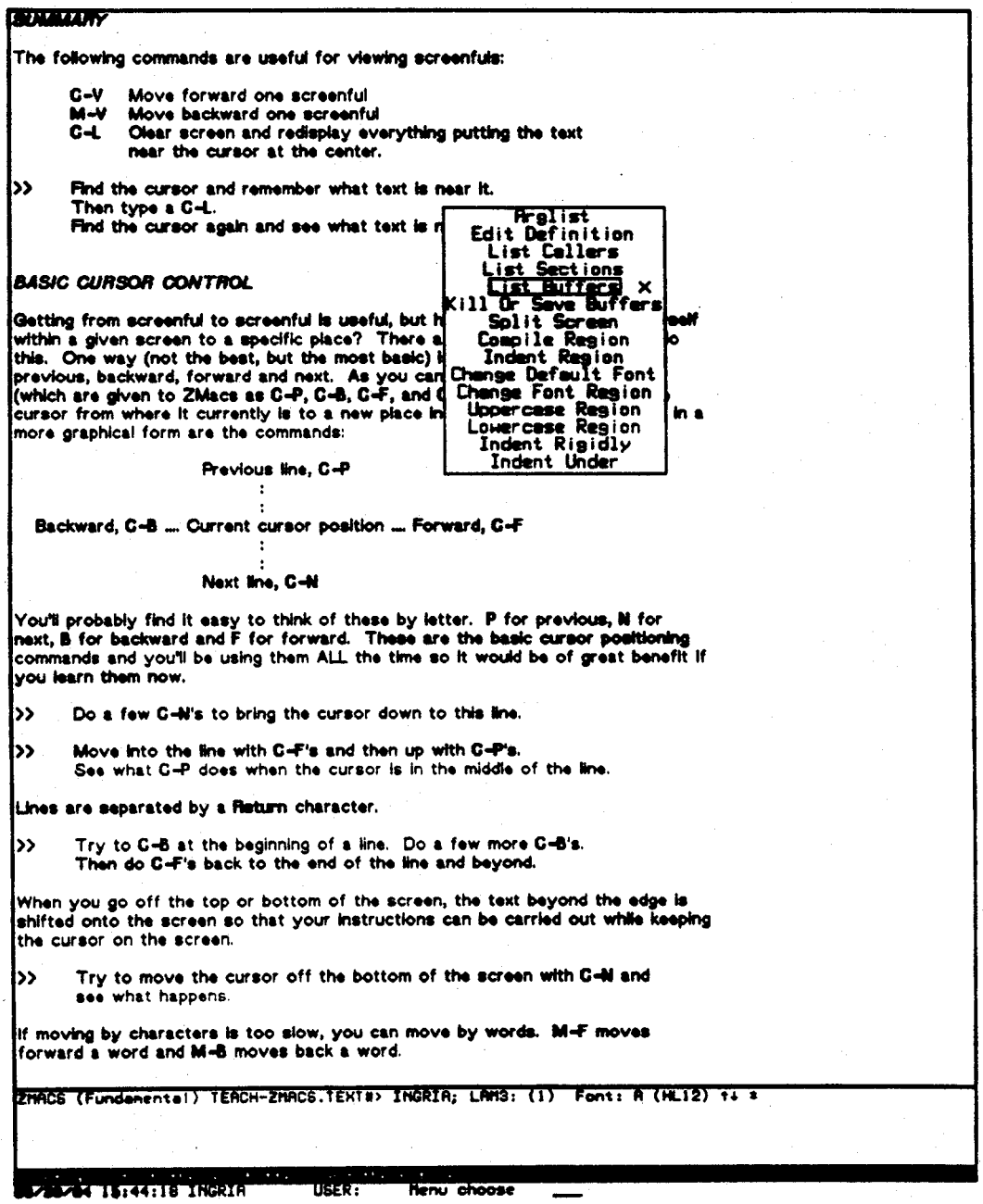


Figure 11. ZMACS main menu

List Buffers
CTRL-X CTRL-B

List all the current buffers in a typeout window. Individual buffer names are mouse-sensitive. Clicking [L] on a buffer name selects it. Clicking [R] pops up a menu of different options.

Kill or Save Buffers
META-X Kill or Save Buffers

List all the current buffers in a menu. This allows you to specify whether each buffer should be saved or killed.

Split Screen
META-X Split Screen

Allow you to split the screen so that more than one ZMACS buffer is visible at once. The current buffers are displayed in a menu like that in Figure 12. Clicking on a buffer causes it to be added to the display. Buffers are added to the display from top to bottom. **New buffer** allows you to create a new buffer as part of the display. **Find file** prompts for the name of a file. It will then be read into a buffer and made part of the display. Clicking on ***Undo*** will remove the last buffer added from the Split Screen display. Clicking on ***Do it*** will cause the screen to be configured as you have specified. Clicking on ***Abort*** will escape from the menu, leaving the screen unchanged.

Compile Region
CTRL-HYPER-C

Compile the functions in the current region. If no region is defined, compile the current definitional form.

Indent Region
CTRL-META-\

Indent each line in the current region. You can specify the amount to indent each line, in terms of spaces in the current font, by providing a numeric argument. A numeric argument of 0 (zero) moves the region back to the left margin. If it is given no numeric argument, the region is indented by the same amount of space that typing TAB would insert. If you use this in a code file, this will produce the right amount of indentation aproporiate for the current piece of code.

Change Default font
CTRL-META-J

Change the default font.

Change Font Region
CTRL-X CTRL-J
CTRL-SHIFT-J

Change the font of the region.

Uppercase Region
CTRL-X CTRL-U

Convert all the text in the region to uppercase.

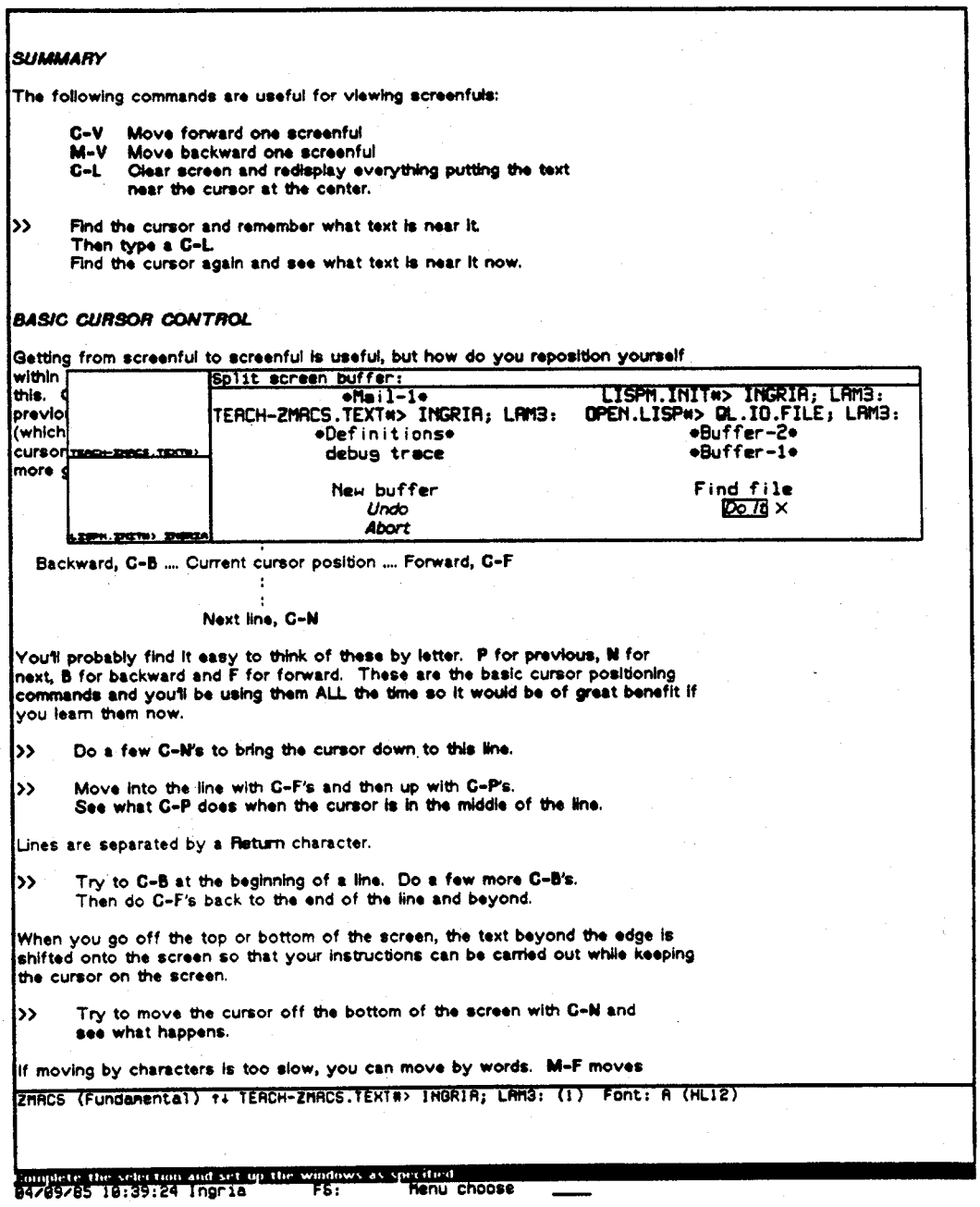


Figure 12. ZMACS Split Screen Menu

Lowercase Region
CTRL-X CTRL-L

Convert all the text in the region to lowercase.

Indent Rigidly
CTRL-X CTRL-I
CTRL-X (TAB)

Indent each line in the current region. The keyboard commands require a numeric argument. Unlike **CTRL-META-**, there is no observation of indenting conventions and no default if a numeric argument is not given. Also, a numeric argument of **0** (zero) does not move the region back to the left margin. To do this you will need to give a negative numeric argument equivalent to the number of spaces to move.

The menu version uses the mouse. Click [L] on the region and hold the left button down. As you move the mouse, the text of the region will move with it. Release the left button when the text is positioned where you want it.

Indent Under
META-X Indent Under

Indent the region under a specified string. The keyboard version only works on the line where the cursor is currently positioned. You will be prompted to enter a string, which should appear earlier in the text than the current line. Terminate it with (RETURN). The left edge of the current line will be aligned with the left edge of the string you specified.

The menu version uses the mouse. Click [L] on any character. The region will move so that is left edge is aligned with the character. If there is no region, the line where the cursor is currently positioned is moved.

5.10 Help

Various help facilities are availalble in ZMACS. Of these, the two most useful are Apropos and Variable Apropos. Apropos, which is invoked by **META-X Apropos** or (HELP) **A**, will prompt you for a substring, which should be terminated with (RETURN). A typeout window will then appear, listing all the ZMACS commands that contain that substring. Each command is listed with the key binding that invokes it as well as a brief description of what it does. Apropos is useful if you cannot remember the exact name of a command, but you can remember some portion of its name. It is also useful if you wish to find out about the ZMACS commands in some particular area. For example, (HELP) **A case** will list all the case conversion commands, as is shown in Figure 13. Apropos, then, provides a convenient online mechanism for learning more about ZMACS.

Note: There is a slight difference in the behavior of **META-X Apropos** and (HELP) **A**. (HELP) **A** lists only the commands that are available in the current ZMACS environment. **META-X Apropos** lists all commands, whether they are available or not. However, unavailable commands will be listed as **CTRL-META-X** commands. Such commands will probably not work in your current ZMACS environment.

Variable Apropos is like Apropos. It is invoked by **META-X Variable Apropos** or (HELP) **V**. It will prompt for a substring, which should be terminated with (RETURN). A typeout window will then appear, listing all the ZMACS variables that contain that substring, along with the current value of each. Such variables control various aspects of the behavior of ZMACS. For example, we

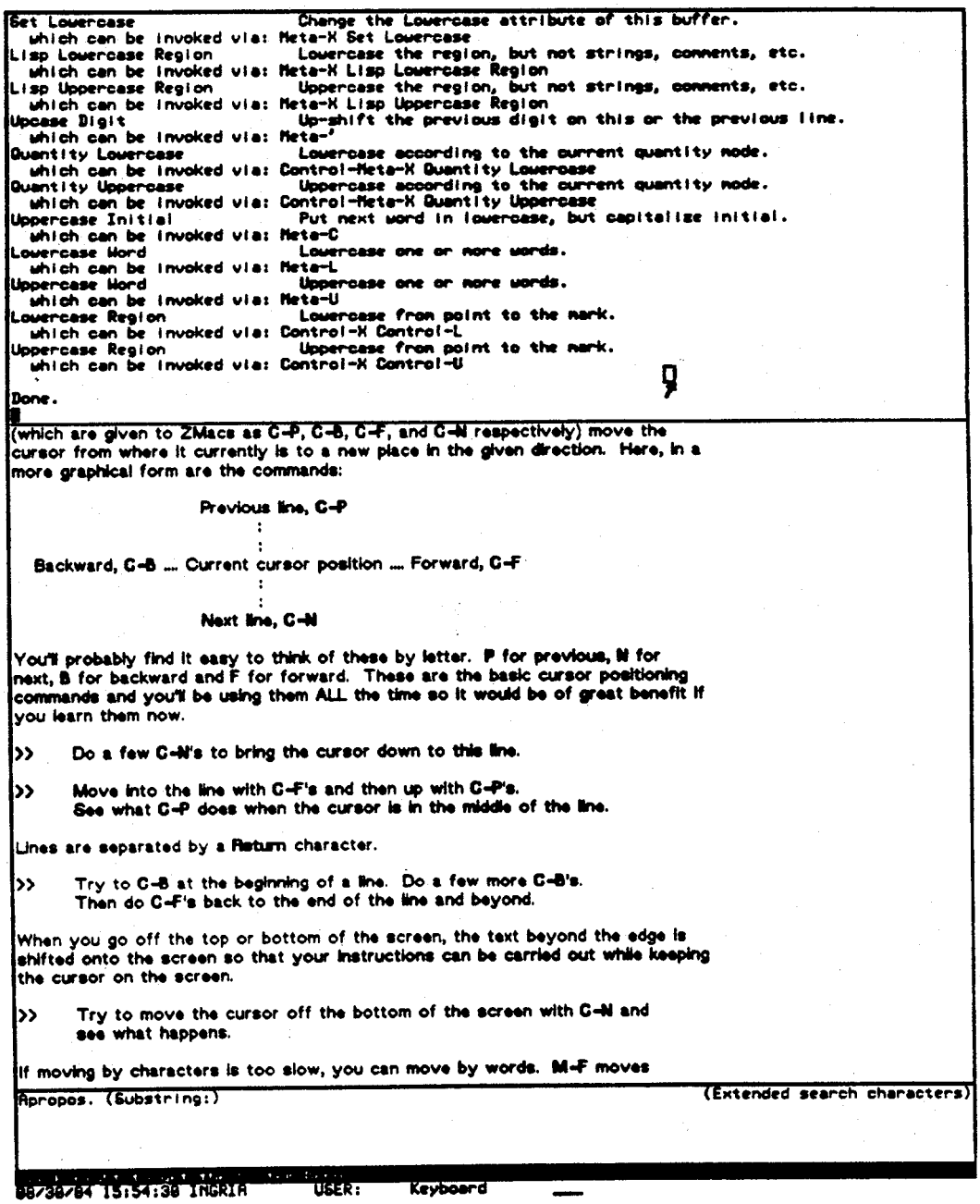


Figure 13. Typeout window from **META-X Apropos case**

have already seen the variable **History Menu Length**, which specifies the number of items that the kill history menu will display. Other ZMACS variables include: **Fill Column**, which specifies the position on the screen where ZMACS will cause text to wrap to the next line; and **Default Major Mode**, which specifies the default mode that ZMACS starts up in, if no other mode is specified. (HELP) V followed by (RETURN) will list all the ZMACS variables and their bindings.

Other useful help commands are:

(HELP) C Prompts for a ZMACS modifier key command, such as **META-F** or **CTRL-HYPER-C**, and prints out the name and description of the ZMACS command invoked by that key.

(HELP) D Prompts for the name of a ZMACS command, such as **Auto Fill Mode** or **Compile Buffer**, and prints out a description of that command.

(HELP) W Prompts for the name of a ZMACS command, such as **Compile Region** or **Forward Word**, and returns the modifier key command, if any, that invokes that command.

6. The Window System

The Window System is the heart of the LMI Lambda; if you understand how it works and how to use it, you can probably make the Lambda do anything you want it to. The Window System can be thought of as the Lambda operating system. Like a more typical operating system, the Window System organizes and controls your access to all the other systems or programs on the computer; unlike such a system, you can never really say that you are on "operating system level".

The main interfaces between you and the Window System are the mouse and the pop-up menus, but there are many system functions that you can access by an alternative method: usually by using either a SYSTEM key, a LISP function, or an extended editor command. For instance you can reach all the main systems through the SYSTEM keys:

SYSTEM E
: Accesses the ZMACS Editor.

SYSTEM I
: Accesses the Inspector.

SYSTEM L
: Accesses the LISP Listener.

SYSTEM M
: Accesses ZMAIL.

SYSTEM P
: Accesses the PEEK system.

The systems listed above are the one's that you are most likely to use. To see a listing of all available systems type SYSTEM HELP.

When you choose a program using the SYSTEM key, the Window System will either select or create a window containing the system you specified. If you want to be sure that you get a new system, type the appropriate letter while pressing the CTRL key. For example, to get a new LISP Listener, type SYSTEM CTRL-L.

The mouse menus are the easiest method to use as you learn the system; use the SYSTEM keys whenever possible when you get more familiar with the Lambda, as they are the quickest method.

There are two reasons why there are multiple ways to accomplish a particular goal. The first is a concern with programmer style: different people like to do things in different ways, and the Lambda is designed to be unobtrusive and allow you to pick the method most natural for you. The second is a concern with the robustness of the system: since all important functions are accessible through both the mouse and the keyboard, you could still reach them in the unlikely event that either of these became non-functional.

6.1 Of Mice and Menus

Pop-up menus are widely used in the Window System. They provide you with a wide choice of options, without requiring that you memorize keywords or long command sequences. To call up a menu, click the appropriate button (usually R) on the mouse. The mouse documentation line will tell you when menus are available from a particular system, and which button to click to access them.

When you press a mouse button to get a menu, the menu will form on the screen around the area where the mouse cursor was. Notice that the mouse cursor now looks like a small "×". Also

notice that the items on the menu are "mouse-sensitive". This means that when the mouse cursor gets near them, the system draws a small box around the item. You can now select an item by "boxing" it and then clicking [L]. Some items take effect immediately; others lead to secondary menus.

If you call up a menu and then decide you don't want it you can usually leave just by moving the mouse cursor outside of the menu boundaries. Some menus have little boxes (called choice boxes) ■ on the bottom labelled **Do It** and **Abort**; in that case move the mouse cursor to **Abort** ■ and click [L]. (Of course if you want to do the action specified by that menu, then click on **Do It** ■.)

6.1.1 Mouse Corners

Mouse corners are the Window System's way of letting you define part of the screen as a window for other Window System commands to act on. Most Window System commands assume that you want to affect the entire screen; some give you a choice. For instance, the [Create] and [Split Screen] options on the main menu both use mouse corners.

When you use mouse corners, the mouse cursor will first turn into something that looks like this: ⌜ an upper left hand corner. Move this to where you want it and click [L]. Then, it will look like this: ⌟ a lower right hand corner. Move this to where you want it and click [L].

There is a quick method for specifying that a window created with [Edit Screen] should take up the whole screen. Place the ⌜ anywhere on the screen and click [L]. Then move the ⌟ to the *left* of the ⌜ and click [L]. The resulting window will take up the whole screen.

6.2 The System Menu

The System Menu provides a covenient interface to many of the facilities of the Lambda. To get the Main System Menu, click [R]. (In ZMACS, and several other programs, you need to click [R2]; the mouse documentation line will always alert you in these cases.) Figure 14 shows this menu.

The main System Menu contains three columns; these are, from right to left: ***Programs***, ***This window***, and ***Windows***. These commands are explained below. If there is an alternate way to execute a command, it is included in parentheses next to the explanation.

Programs

[Lisp] Puts you into a LISP Listener. (Alternative ⟨SYSTEM⟩ L)

[Edit] Puts you into an editor buffer. (Alternative ⟨SYSTEM⟩ E)

[Inspect] Puts you into the Inspector, looking at the specified object.

[Mail] Puts you into ZMail, the Lambda's mail program. (Alternative ⟨SYSTEM⟩ M)

[Trace] Asks for a function name and then offers different tracing options.

[Emergency Break]
Puts you into the "Cold Load Stream". This is a LISP Listener that does not use the Window System for I/O. Sometimes this is necessary if the window software is ailing. Do not use this unless you really need to; it puts the machine into a very fragile state, and if you stay there for too long all your network connections break down. (Alternative ⟨TERMINAL⟩ ⟨CALL⟩)

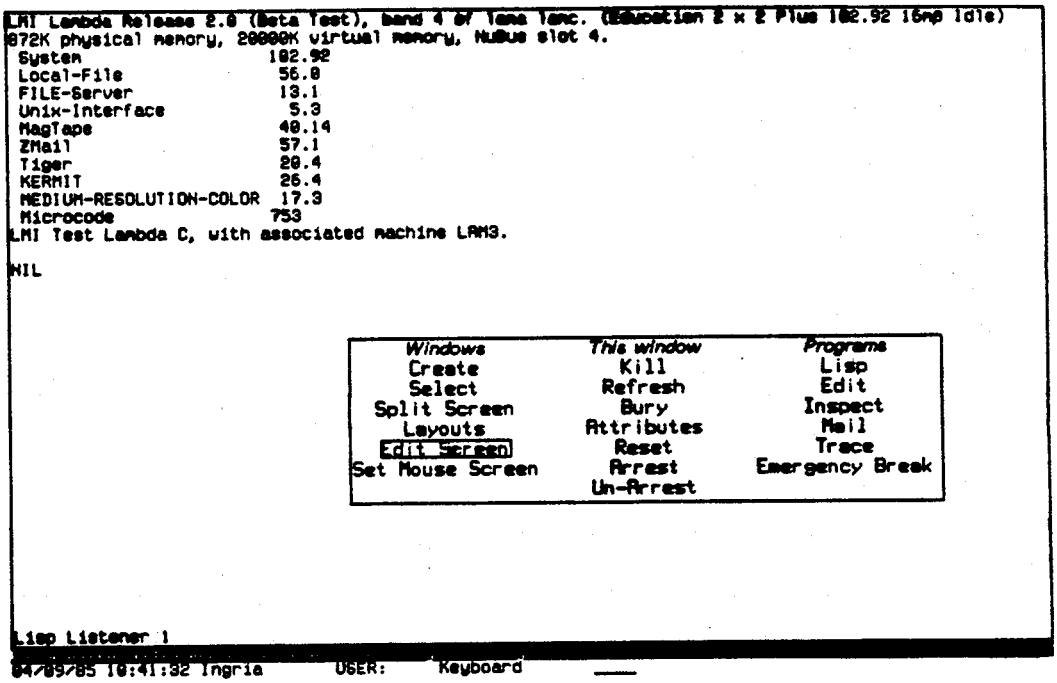


Figure 14. The System Menu

This Window

Kill — Kills both the current window and the process running in it. If you choose this option, it presents you with a small "confirmation menu". This asks if you really meant what you requested. Indicate yes by clicking L in the window, indicate no by moving the mouse cursor away. Unless you confirm your choice, the system will not execute it. (Also available from the Window Hierarchy display in Peek.)

Refresh — This refreshes the display in the currently active window. Use this when something weird has happened to the display. (Also available from the Window Hierarchy display in Peek.)

Bury — Buries the current window; your window will disappear, and you will see whatever was "below" it. (Also available from the Window Hierarchy display in Peek.)

Attributes

Allows you to change the appearance of the current window, by using the menu shown in Figure 15.

If an attribute has only a single value, like "Label", you must enter a new value to change it. Click L on the old value to remove it, then type in a new value and terminate it with RETURN. If an attribute is followed by several possible values, the choice currently in force is displayed in boldface. To change the value, click ⌒ on

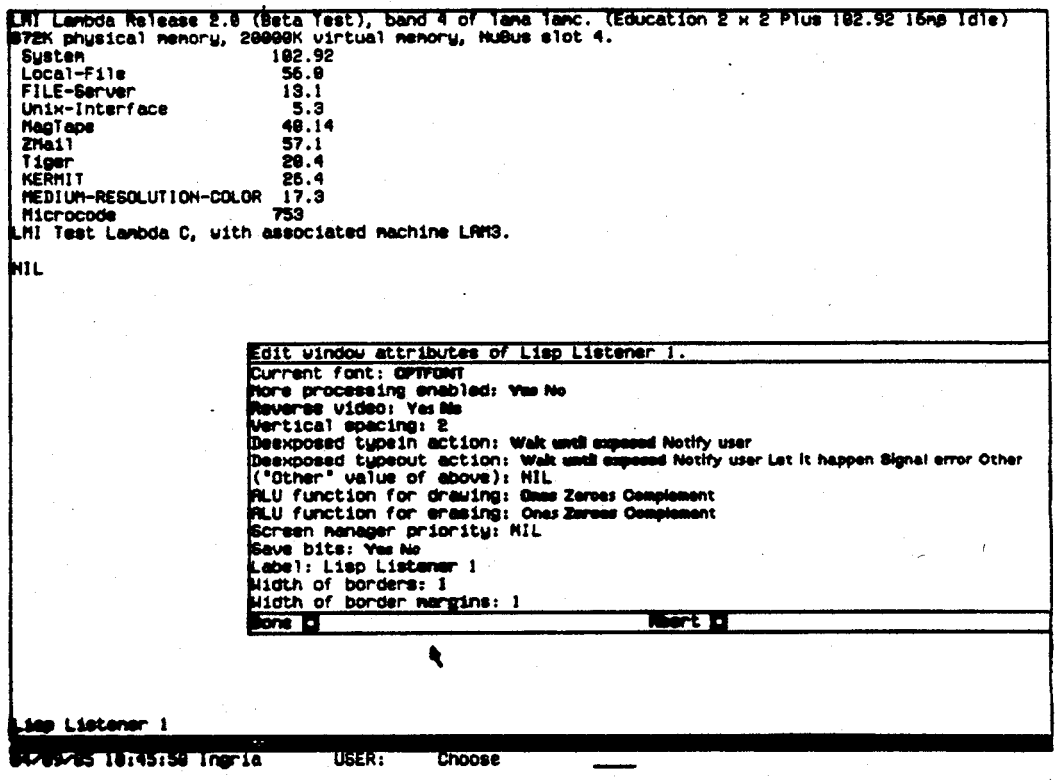


Figure 15. The Attributes Menu

another value.

Reset Starts the process associated with the current window from scratch. Don't do this if you have valuable, unsaved material. However, if the process is wedged, this may be the quickest way to unstick it and start over. (Also available from the **Window Hierarchy** display in Peek.)

Arrest Halts the process associated with the current window. (Also available from the **Window Hierarchy** display in Peek.)

Un-Arrest
Starts the process associated with the current window back up from where it left off. This will always work if the process was arrested from the System Menu; if the process stopped for a different reason this may not restart it. (Also available from the **Window Hierarchy** display in Peek.)

Windows

Create Brings up a sub-menu of types of windows that you can create, then asks you to define the window shape using mouse corners. **Note:** One of the menu options offered to

you is [Any]. If you select this option, you will be prompted for the name of a window flavor, such as **tv:lisp-listener**. You should pick this option only if you know the name of the flavor of window you want to create.

[Select] Brings up a sub-menu of all the available windows so that you can select one.

[Split Screen]
Presents you with a menu, as shown in Figure 16, that allows you to define how you will split your screen, and with what components.

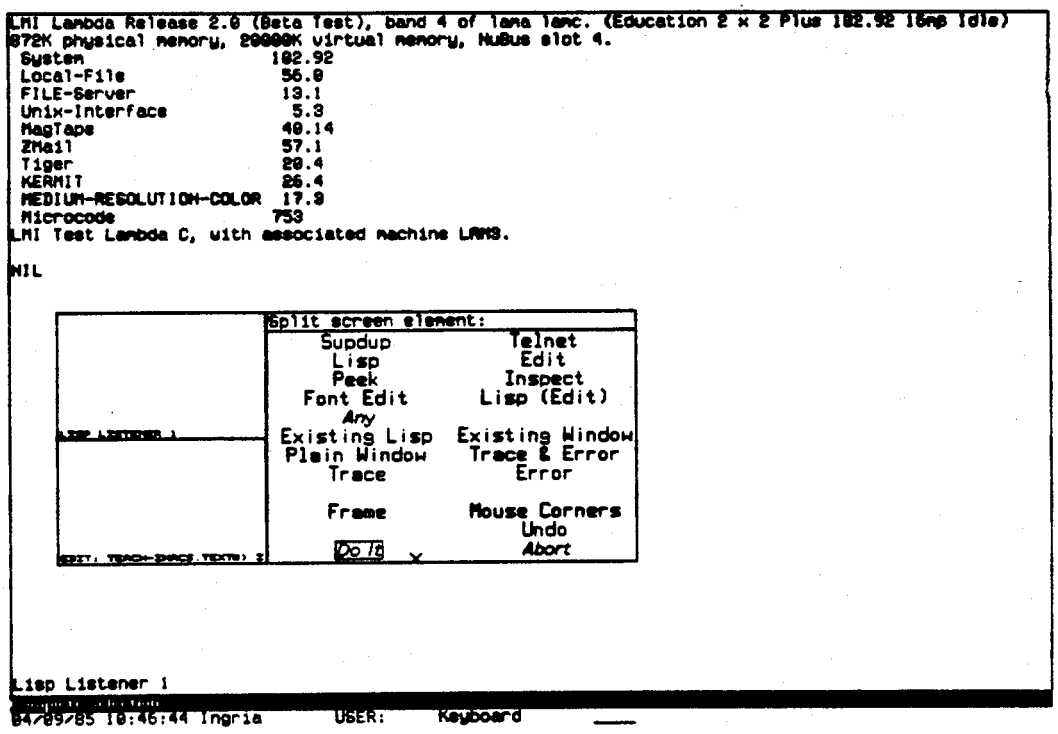


Figure 16. [Split Screen] Menu: Note that it allows you to preview its result, in miniature.

The [Existing Window] option allows you to select from a secondary menu of all the available and already created windows. The [Frame] option allows you to save this window arrangement by "framing" it and giving it a name.

[Layouts] Another way to get the "frame" effect available from the [Split Screen] option. This brings up a menu with at least the two options [Save This] and [Just Lisp]. By picking [Save This] you can save your current screen setup under a name that you choose. Next time you pick layouts, the name of your design will also be on the menu for you to choose.

Edit Screen

Brings up a secondary menu of different ways that you can manipulate all the different windows, as shown in Figure 17. Many of these items are similar to options under **This Window** on the main menu. Note that you cannot leave this option just by moving the mouse cursor out of the menu; you must choose the Exit option in order to leave.

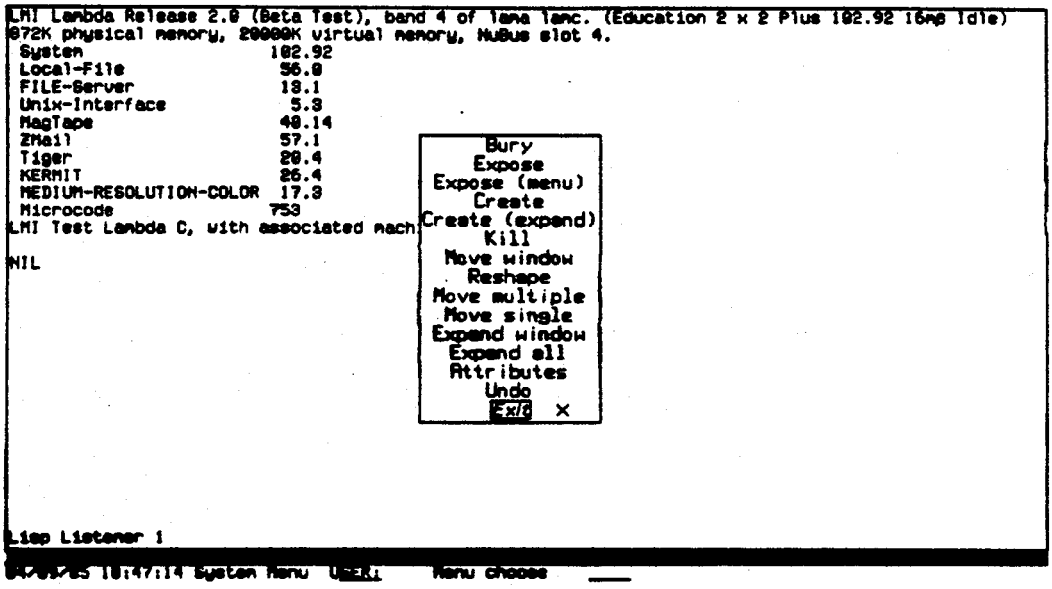


Figure 17. The Edit Screen Menu

Set Mouse Screen

This has an effect only if you have more than one monitor attached to your Lambda. If so, this option allows you to move the mouse cursor to another screen. Usually clicking [L] produces the default option and clicking [R] gives you a menu with all the options. On a two-screen system with a medium resolution color monitor, clicking [L] cycles through all the menu options in turn and clicking [R] presents you with the menu so that you can choose. (Alternative (TERMINAL) ≥)

6.3 Special Windows

Sometimes the Lambda creates windows that you did not explicitly request. This usually happens in one of two circumstances: when you ask the Lambda for help, or when the system needs to give you information about a process running in an unselected or non-visible window.

6.3.1 Temporary Windows

These are the kinds of windows that you see when you ask the system for help; they often say **Press any character to flush** on the bottom. The Lambda creates a window to serve this specific purpose and destroys it when the job has ended. If you press (HELP) and subsequently ask to select a window from the menu, **Help Window** will not be one of the choices.

6.3.2 Typeout Windows

Typeout windows are often used to present you with information that takes up several screens. While most windows have a fixed size, unless you edit them, typeout windows do not. A typeout window starts with its upper edge aligned with the top of the currently selected window and "grows" down the screen, like a window shade being pulled down, as information is output to it. When the bottom of the typeout window reaches the bottom of the screen, ****more**** will sometimes appear at the bottom of the typeout window. This means that the system has more information to display than it can fit on one screen. To see "more", press (SPACE).

6.3.3 Notification

Sometimes the system needs to notify you that an unselected process has got an error or that a background process that has no window to type out to is attempting to type out. The console will beep and the message "**Notifications pending TERMINAL N is one way to see them**" will appear either in square brackets ([]) on the screen or on the mouse documentation line. To select the window referred to by the notification, type (TERMINAL) **OS**; to return to your previous window type, (TERMINAL) **S**.

Notifications get saved; to see old ones again, evaluate **(tv:print-notifications)**.

6.4 The Window Stack

Another way to select windows (probably quicker once you get comfortable with the Lambda) is through the (TERMINAL) *n*S command.

All the currently selectable windows are arranged in a kind of stack (actually a ring) with the selected window on top. Figure 18 is a schematic diagram of the window stack. (Evaluating **(listarray tv:previously-selected-windows)** will give you the current stack order minus the top (currently selected) window.) When you select a window either with the mouse or some version of (TERMINAL) S, the Window System moves it to the top of the stack.

(TERMINAL) **S**
: Gives you the window active before the current window and moves the ex-current window to be the second item on the stack.

(TERMINAL) **1S**
: Gives you the previously active window and moves the old one to the bottom of the stack. Note that although both of these commands access the same window for your use, their stack behavior is different.

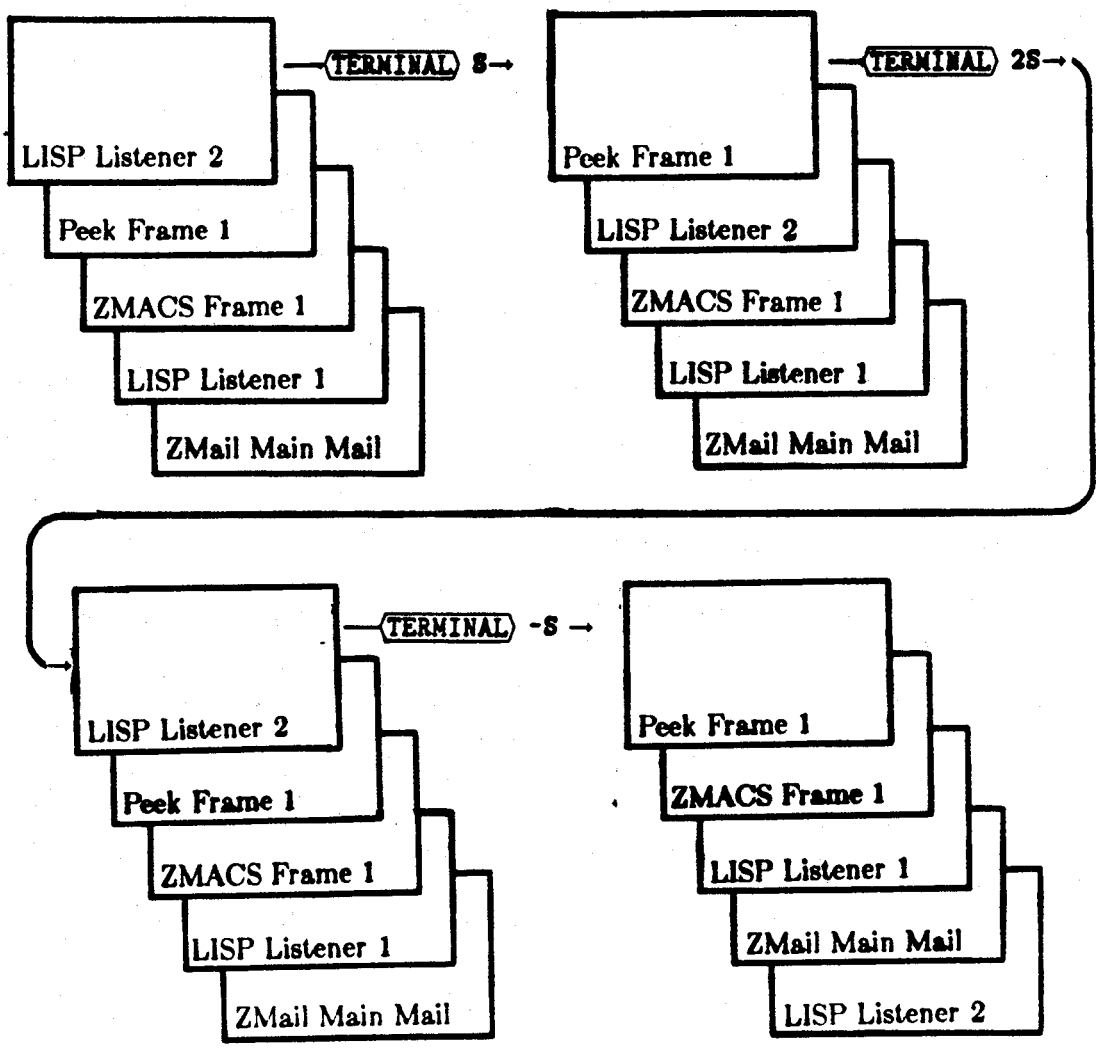


Figure 18. The Window Stack

TERMINAL **-S**
Takes the window on the bottom of the stack and moves it to the top.

TERMINAL ***n*S**
Takes the currently selected window and puts it in the *n*th place on the stack and exposes the window below it. If you do this repeatedly the effect is to cycle through the top *n* items on the stack. (TERMINAL **S** is actually shorthand for TERMINAL **2S**.)

TERMINAL **-*n*S**
Takes the window on the *n*th place on the stack and puts it on top. Used repeatedly, this cycles through the top *n* items on the stack in reverse order from the previous command.

Note that while both the [Select] option from the System Menu and the ⟨TERMINAL⟩ S command update the stack variable, something like cycling through the available LISP Listeners using ⟨SYSTEM⟩ L repeatedly will not. So, if you have been doing that and then access a window using a variant on ⟨TERMINAL⟩ S, you may not get the window you expect.

6.5 Peek

Peek is a program that allows you to look at the status of various system functions. You can also manipulate both processes and windows from within Peek. It is a good way to get rid of extraneous windows and their associated processes. Also, when a process appears to be stopped for no apparent reason, you can often find out why or reset it from Peek.

To get a Peek Window type ⟨SYSTEM⟩ P; the first time you request a Peek Window you will see a display like the one in Figure 19.

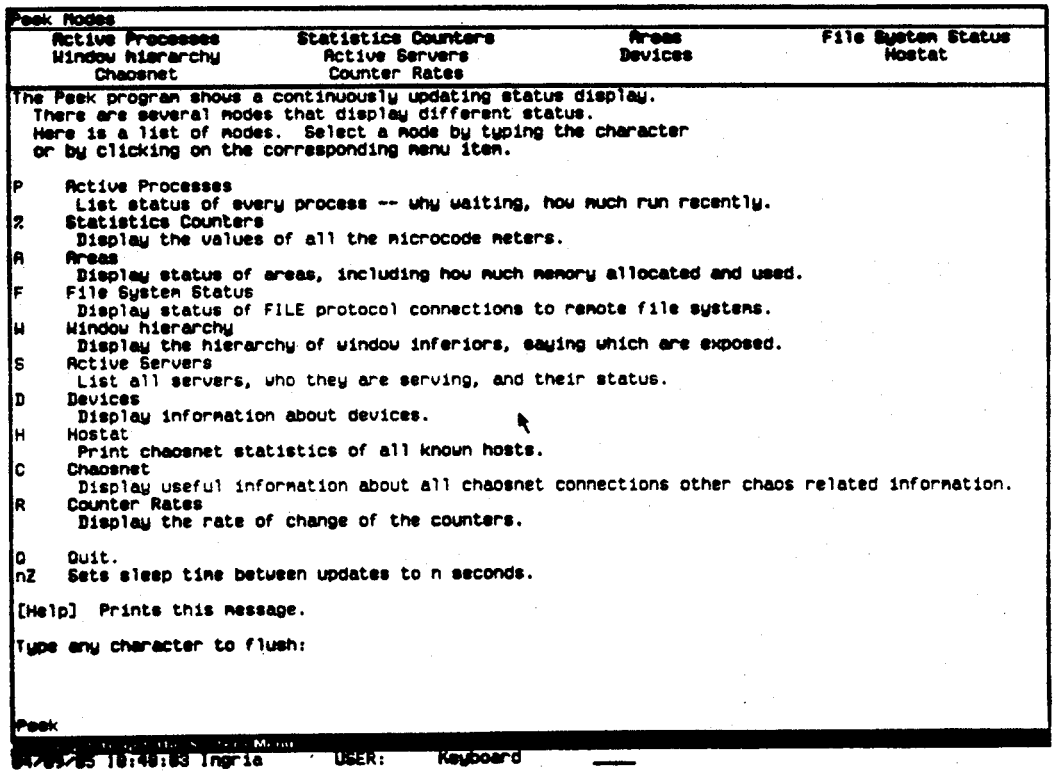


Figure 19. Initial Peek Display

Subsequently, you will return to Peek exactly where you left it. (Of course, you can get a brand new Peek window by typing ⟨SYSTEM⟩ CTRL-P, but it is usually unnecessary to have more than one.)

Q(uit) leaves Peek and returns you to the previous process.

Many of the Peek subsystems update their displays frequently, because they give information on ever changing processes. *n*Z allows you to change the interval at which these updates display. *n* is the number of 60ths of one second. The default update period is five seconds.

The remaining choices are accessible both by single letter commands and by mouse-sensitive options in the command menu at the top of the screen. The two of most general interest are [Active Processes] (p) and [Window Hierarchy] (w), so those will be discussed in greater detail.

6.5.1 Active Processes

The Active Processes display allows you to see all the different processes running on the Lambda and their current state. A typical display looks something like Figure 20.

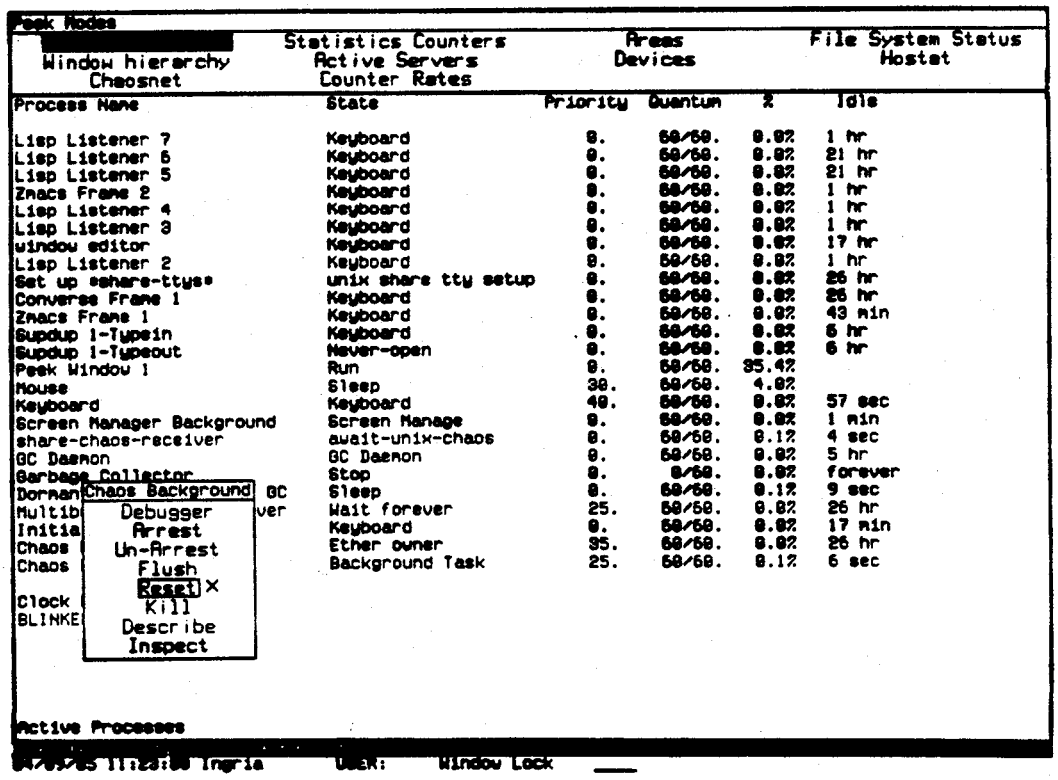


Figure 20. The Active Processes display

Notice that the items on the display are mouse-sensitive; you can "box" a process and then click and get a menu of operations you can carry out on that process.

Many of these choices are operations that you can access from the main menu, but here you can affect any process, not just the current one. Often, Peek is a way to figure out why a process

is hung, and to get it to start again. You can also use the debugger or the Inspector (for more information see Chapter 7) to analyze in more detail what the problem with the process is. If you just want to get it going again, and don't care about the recent state, you can [Reset] the process or [Kill and Un-arrest] it.

6.5.2 Window Hierarchy

This display allows you to see all the active windows and how they relate to one another. For instance, if you have used the [Layouts] or [Split Screen] options to set up a "frame" with several windows in it, they will be displayed as subordinate to the frame. Several system programs are implemented as frames. For example, the Window Error Handler, the Inspector, and Peek itself are frames. ZMACS is also a frame and individual ZMACS buffer windows are subordinate to the ZMACS frame. Finally, all windows are subordinate to the screen that they appear on. A typical [Window Hierarchy] display is shown in Figure 21.

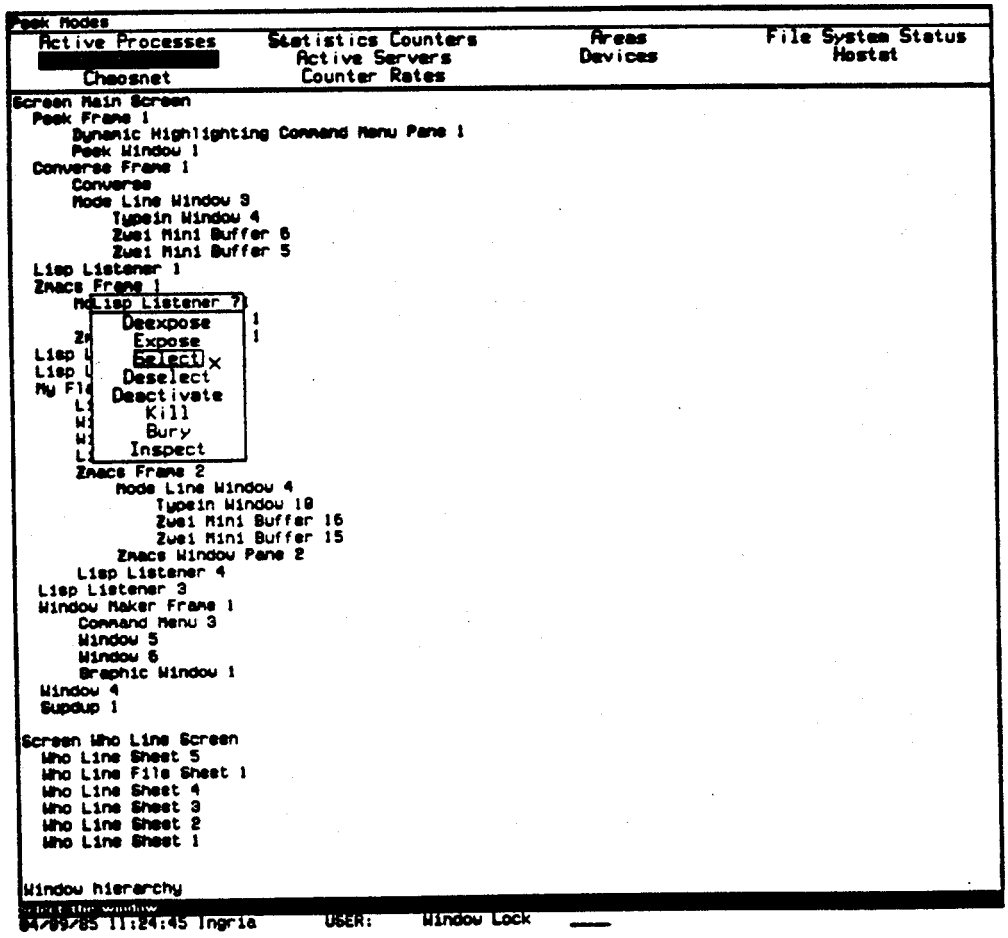


Figure 21. The Window Hierarchy display

Again, the items in this display are mouse-sensitive. You can click on the mouse and get the

small menu displayed in Figure 21; these options are self explanatory.

6.5.3 Mouse Scrolling

Peek and some of the other systems on the Lambda sometimes have more data to display than will fit on the screen. In such cases you can use the mouse scrolling feature to see the rest of the display. There are two mouse scrolling facilities:

Margin Scrolling
: Little italic legends on the screen say ***More Above*** or ***More Below***.

Scroll Bar
: The mouse cursor becomes a heavy black arrow when brought near the left edge of the screen.

Margin Scrolling

When the mouse cursor is near the top or the bottom of the window (and it looks like a heavy up or down arrow) you can scroll the window by moving the mouse slowly in the appropriate direction. To move a whole screenful at a time click ⌒ on the ***More Above*** or ***More Below*** legends.

Scroll Bar

If you move the mouse cursor to the left side of the screen you will see a fat double-headed arrow. This means that using the mouse buttons will scroll the display in various ways:

L — Moves the line next to the mouse to the top of the window.

L2 — Moves the line next to the mouse to the bottom of the window.

R — Moves the top line to where the mouse is.

R2 — Moves the bottom line to where the mouse is.

M — Jumps to a place in the window contents as far, proportionally, from its beginning as the mouse cursor is from the top of the window.

Note that when the mouse cursor has assumed its mouse scrolling form you cannot reach the system menu.

Most windows that have the margin scrolling facility also have the scroll bar facility. However, some windows, such as the displays in Peek, have a scroll bar but do not have margin scrolling.

7. Debugging: When and How

The Lambda programming environment provides several tools for analyzing your programs to locate their problem spots. Foremost among these is the error handler. Others, such as Trace, the Stepper, and the Inspector can be used on a volunteer basis; ZetaLISP will never forcibly place your program in one of these.

When the Lambda system detects an error condition it automatically activates the error handler, also known as the debugger. Sometimes, if the error is trivial, you will be allowed to choose whether or not to enter the debugger and your screen will look something like Figure 22.

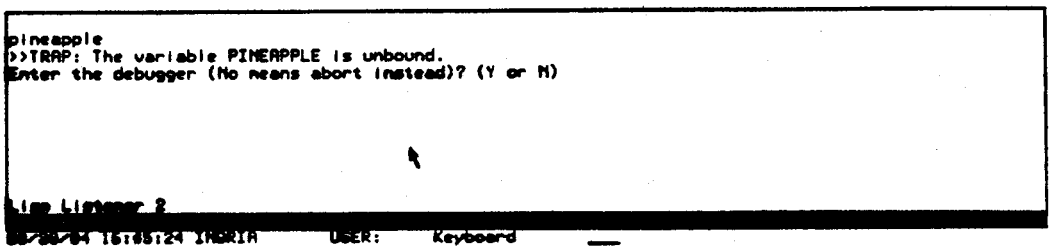


Figure 22. A trivial error

More significant errors can be of two types:

- Syntax or language related errors
- Logic or intent errors

7.1 The Debugger

The debugger can be used both to locate problems with your program, and to fix those problems and proceed with execution. When inside the debugger you can type either a LISP form to be evaluated, or a debugger command. (Debugger commands are explained below.) If you don't really want to be in the debugger you can exit by pressing (ABORT). This brings you back to the LISP Listener and aborts execution of the offending LISP form. (Note that if you type an incorrect LISP form while in the debugger this will cause you to enter a second, deeper level of the debugger. In this case, (ABORT) just brings you up to the previous level.)

When your program enters the debugger you will see a display like the one in Figure 23.

The → (right arrow) prompt indicates that the debugger is waiting for input from you. The number of arrows indicates how many levels deep you are in the debugger.

Many error handler commands can take numeric arguments. If it makes conceptual sense for a command to take such an argument it probably does. To give a numeric argument to a command, type CTRL-*number* followed by the debugger command. For instance:

```
CTRL-5 CTRL-N
```

moves down five stack frames.

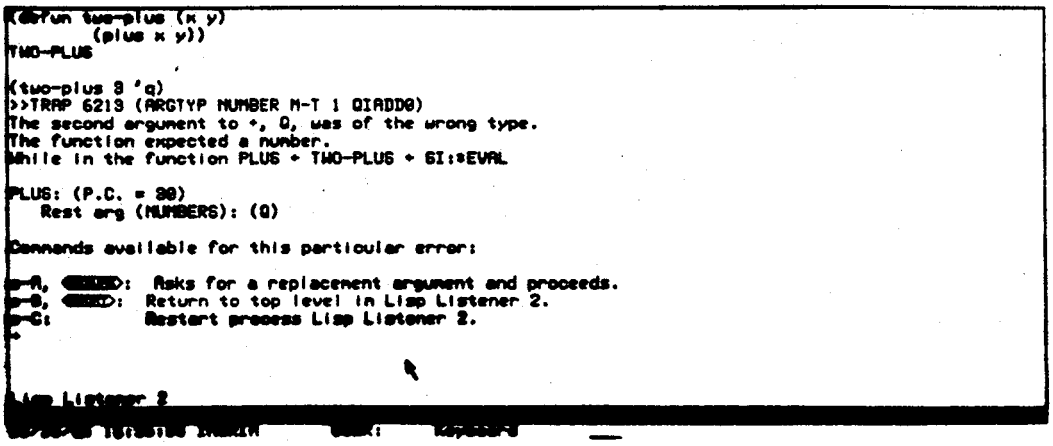


Figure 23. The Debugger

7.1.1 Debugger Commands

The discussion below details the available debugger commands. They are grouped by function rather than strictly alphabetically so that it is clear what each command does, and when it is likely to be useful. (Chapter 30 of the *LISP Machine Manual* contains a more complete explanation and reference.)

7.1.2 Stack Frames

Often, an error happens much earlier in a program than when its effects become apparent and LISP throws your function into the error handler. So it is useful to be able to move "up" and "down" your program to both before and after the point where it enterred the debugger. You do this by giving the debugger commands that change the current stack frame. You can consider a stack frame to be a snapshot of the status of a program at any single point in its execution. A typical stack frame display is show in Figure 24.

The LISP Machine distinguishes between two types of stack frames, "interesting" ones and "uninteresting" ones. Interesting frames result from what you would normally think of as program code. Uninteresting frames involve LISP internal functions such as ***eval** and **prog**. Since it is unlikely that you want to "debug" **eval**, there are stack frame commands that automatically skip uninteresting frames.

The stack frame commands produce output in a pseudo-machine language format. This is the "disassembled" LISP code for which there is no actual assembler. It is fairly straightforward, and you should be able to get information on function calls and variable values just by looking at it. See the *LISP Machine Manual* if you want to understand LISP Machine assembly language in greater depth.

The following are the relevant commands for moving among and viewing stack frames:

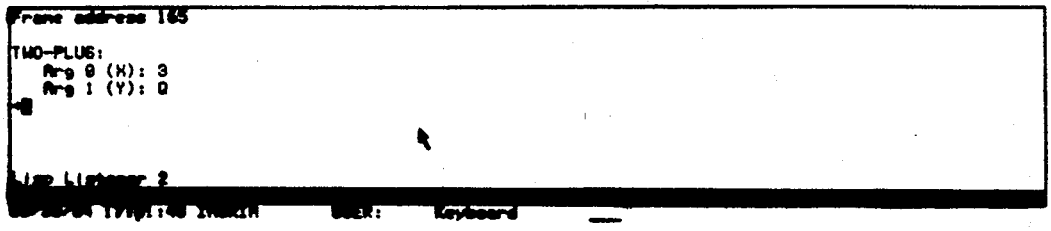


Figure 24. A stack frame

CTRL-A Prints argument list of function in current frame.

CTRL-META-A
Examines or changes the *n*th argument of the current frame.

CTRL-L Redisplays error message and the current frame.

META-L Displays a full-screen typeout of the current frame.

CTRL-META-L
Gets local variable *n*.

CTRL-N
LINE Moves down to next interesting frame.

META-N Moves down to next interesting frame and provides a full-screen frame display.

CTRL-META-N
Moves down to next frame even if it is uninteresting.

CTRL-P
RETURN Moves up to the previous interesting frame.

META-P Moves up to the previous interesting frame and provides a full-screen frame display.

CTRL-META-P
Moves up to the next frame even if it is uninteresting.

CTRL-S Searches for a frame containg a specified function.

META-S Reads the name of a special variable and returns that variable's value in the current frame. Instance variables of **self** may be specified even if not special.

CTRL-META-S
Prints a list of special variables bound by the current frame and their values. If the frame binds **self**, all the instance variables of **self** are listed, even if not special.

CTRL-META-U
Moves up the stack to the previous interesting frame.

META-< Goes to the top of the stack.

META-> Goes to the bottom of the stack.

7.1.3 Backtracing

A backtrace is a concise history of where a function has been. The error handler has several commands that allow you to access this information.

CTRL-B Prints a brief backtrace.

META-B Prints a longer backtrace.

CTRL-META-B
Prints a longer backtrace; does not censor interpreter functions.

7.1.4 Proceed Commands

The following commands allow you to "proceed" with your program after first making changes so that it will (hopefully) run. You can change variable bindings and function definitions, or if you know what the frame with the bug ought to return, you can force it to return that value and then continue.

CTRL-C
(RESUME) Attempts to continue the function, using the first proceed type available.

META-C Attempts to continue after first "permanently" fixing the error. (This command is not available for all errors.)

CTRL-D Attempts to continue, like **CTRL-C**, but traps on next function call.

META-D Toggles the "trap on next function call" flag.

CTRL-E Places you in the editor with source code of the current frame. Return to error handler with **(TERMINAL) S**.

CTRL-META-F
Sets * to the function in the current frame.

CTRL-R Returns a value from the current frame as if it had actually been computed.

CTRL-META-R
Reinvokes the function in the current frame. You use this if you think you have located and fixed the error.

CTRL-T Throws a value to a tag.

CTRL-X Toggles the flag in the current frame that causes a trap on exit or throw through that frame.

META-X Sets the flag causing a trap on exit or throws through the frame for the current frame in addition to the frames outside it.

CTRL-META-X
Clears the flag for the current frame and all the frames outside of it.

SUPER Commands:
SUPER-A, **SUPER-B** etc. are assigned to the various proceed types available for the particular error. Since these are different each time, the debugger always prints them out for you. If you need to see them again type **(HELP) X**.

7.1.5 Miscellaneous Commands

⟨ABORT⟩ In general, this command rescues you from completing your current action. For instance, if you are typing a LISP form to the debugger it aborts the form, and if you are at a particular level in the debugger it moves you up one level.

?

⟨HELP⟩ Prints a really excellent help message.

CTRL-META-Y

Yanks a copy of the top item from the input history.

7.1.6 Window Error Handler

You can enter the Window Error Handler only after you are already in the keyboard level debugger. **CTRL-META-W** is the debugger command that activates the Window Error Handler. The Window Error Handler gives a more graphically oriented view of what is happening with your program; the keyboard error handler is better for manipulating variables and moving up and down the stack. Whether you use one or the other, or both, is largely a matter of taste and programming style. The error handler window is divided into seven panes; Figure 25 shows the Window Error Handler display.

The Window Error Handler has a menu of commands, and in addition many other items are mouse-sensitive. It is easy to use, and the mouse documentation line can tell you what your possibilities are.

7.1.7 Entering the Debugger Manually

Errors of logic will probably not cause your program to enter the debugger. Your program will run; it just won't do what you intended that it do. In cases like this you may find it helpful to use the debugger anyway to help locate errors. There are several ways to do this. The simplest is just to type to the LISP Listener something silly like **pineapple-pie**, which you know is unbound; this will throw you into the debugger with an unbound variable error. You can use **META-**⟨BREAK⟩, which will force the program into the debugger, as long as it is reading from the keyboard; **CTRL-META-**⟨BREAK⟩ will force the program into the debugger immediately. If your program takes off for never never land, this is the way to see what's going on. ⟨BREAK⟩ without modifiers simply enters a "break" loop.

7.2 Other Debugging Aids

In addition to using the debugger, you may find that the Trace and Stepper facilities can provide information helpful in debugging your program.

7.2.1 Trace

The Trace facility allows you to request that LISP take certain special actions upon entering or leaving one or more specified functions. The default action is to print the function name and arguments when the function is called and to print the function name and values when the function

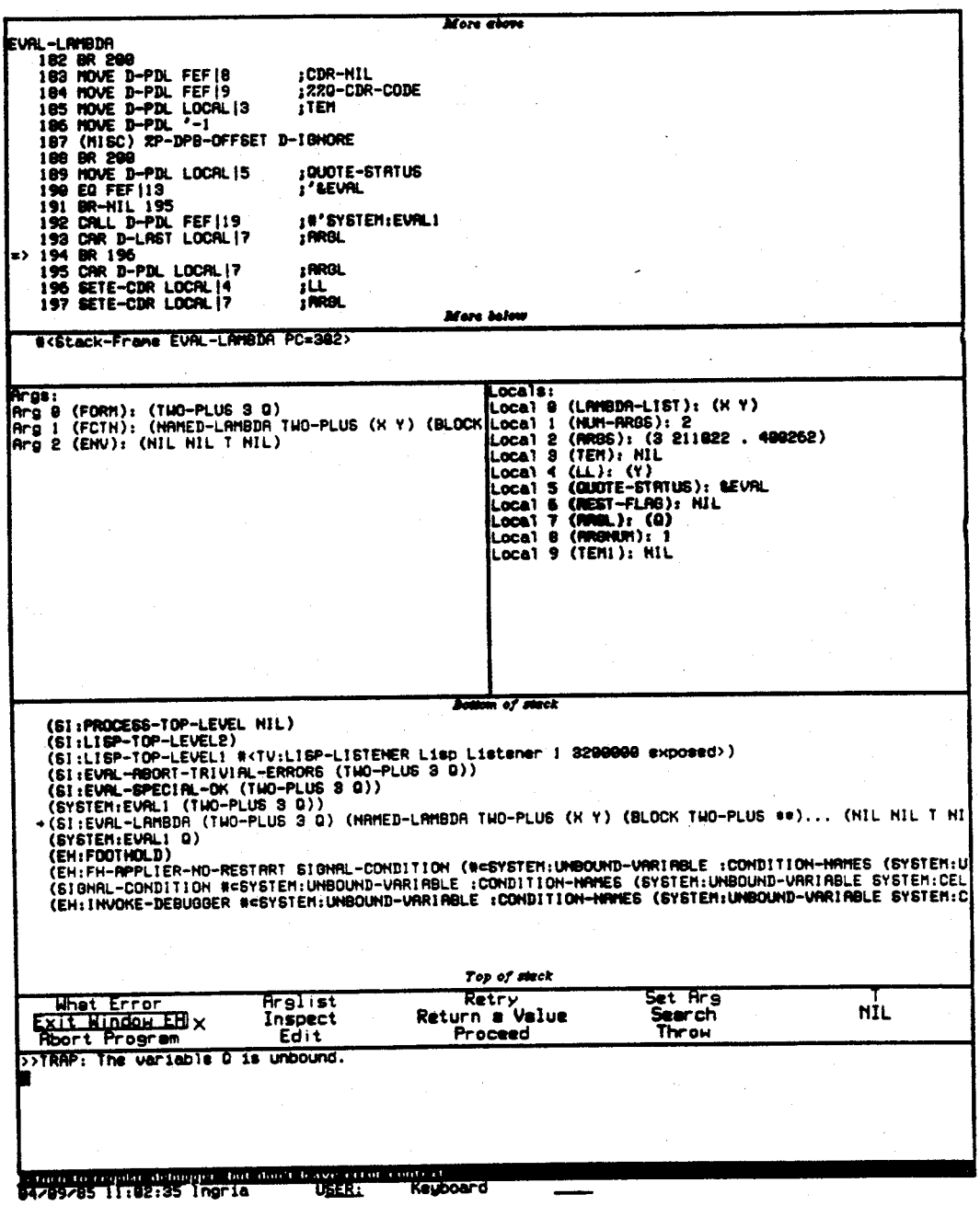


Figure 25. The Window Error Handler display

is exited. Trace can be very helpful when you are trying to track a runaway recursion. The Trace facility can be invoked in several ways. One method is via the **trace** special form.

trace *spec-1 spec-2...* Special form

Some of the forms a *spec* can take are:

A symbol: *function-name*

function-name will be traced in the default manner.

A list of the form: (*function-name option-1 option-2...*)

function-name is traced in a manner controlled by the *options*; some of the more commonly used options are listed below. If an *option* takes an argument, this argument should immediatley follow it.

A list of the form: ((*function-1 function-2..*) *option-1 option-2...*)

This is like the previous form but allows you to specify common options for more than one function.

These various forms can be intermixed in a single invocation of **trace**. However, you don't need to list all the functions that you want traced at once. You can always add more and the system will keep track of all of them.

Here are some of the options. (As usual, the *LISP Machine Manual* provides complete documentation.)

:break *pred*

If *pred* evaluates to non-**nil**, then this causes LISP to enter a breakpoint after printing the Trace entry information, but before executing the function. The example indicates how you would specify this option to Trace.

```
(trace (stew :break (equal meat 'rotten)))
```

:exitbreak *pred*

Similar to **:break** except that the breakpoint occurs after the function is executed and the trace exit information is printed.

:step Causes the function to enter single-step execution every time it is called. **step** is discussed in the following section.

:wherein *function*

Traces the function only when it is called by *function*.

You can also access Trace from the system menu and through the ZMACS extended command **META-X Trace**. Both of these present you with a menu listing the options so that you don't have to memorize the syntax of the **trace** function.

To remove one or more functions from the list of those to be traced by the system, use the special form **untrace**. If **untrace** is given one or more function names as arguments, it stops these functions from being traced. If **untrace** is called with no arguments, it stops all functions from being traced.

7.2.2 The Stepper

The Stepper allows you to single-step through the evaluation of any function. This is similar to the single-step facility found in many machine language debuggers. You cannot use the Stepper

with a compiled function; you can step through interpreted code. If your interpreted function calls a compiled function, then when you reach that point with the Stepper you will see entry and exit only; using the Stepper on compiled code won't hurt either the Stepper or the code, but it won't be very informative either.

The Stepper may be entered either through the **:step** option to **trace** or via the **step** function.

step *form* Function

form should be a function with its argument(s), if any. For example, to trace a function called **myfun** with the arguments **t** and **0**, you should evaluate the form:

```
(step '(myfun t 0))
```

Stepping through a function looks like Figure 26.

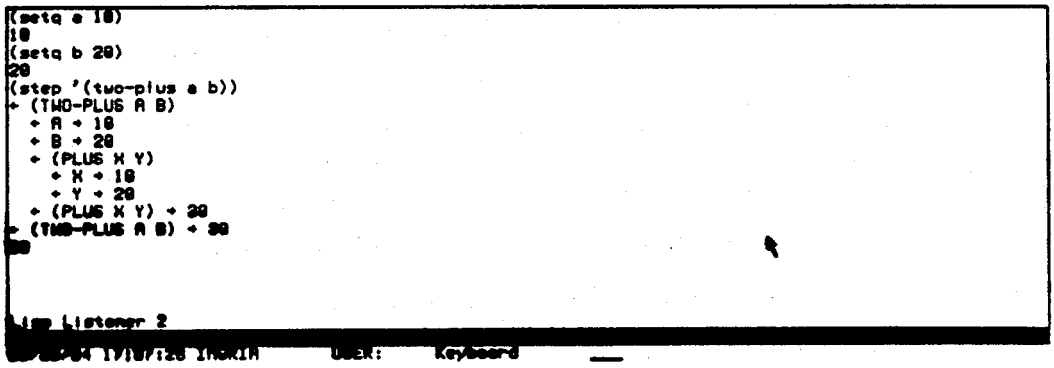


Figure 26. Using the Stepper

Before any form is evaluated it is printed out preceeded by a →. (If it is a long form it may only be partially printed.) The Stepper expands all macros; it signals macro expansions with a double arrow: ↔. A backwards arrow ← will precede the form and values returned by a function. A ∧ character will appear between values if the form returns multiple values. When the stepper has finished evaluating the arguments to a form and is about to apply the function it prints a "λ".

Recursively called functions indent proportionately to their recursion levels so you can follow the process graphically.

Stepper Commands

CTRL-N Steps to the next evaluation. The Stepper continues until the next item to print out, then accepts another command.

SPACE Goes to the next form to be evaluated at this level. All the evaluations at lower levels occur, but are not single-stepped or printed out. This is a quick way to move through parts of the program that don't interest you.

CTRL-A Doesn't show argument evaluation, but pauses before applying the function.

CTRL-U This is like SPACE but more so; it skips over all evaluation on the current level as well as lower levels.

CTRL-G Grinds or "pretty-prints" the current form.

CTRL-T Retypes the current form in full, without any truncation.

CTRL-L Clears the screen and redisplays the last ten pending forms.

META-L Similar to CTRL-L, but doesn't clear the screen.

CTRL-META-L
Redisplays *all* pending forms.

CTRL-E Switches to the editor.

CTRL-B Puts your function in a breakpoint—read-eval-print loop—from which you can examine and set variables and other aspects of the current environment. Several special variables are available:

step-form Is bound to the current form.

step-values
Is bound to a list of returned values.

step-value Is bound to the first returned value.

If you change the value of these variables, it will affect execution.

CTRL-X Exit, finishes the evaluation without any more single-stepping.

HELP
? Prints information on the Stepper commands.

7.2.3 The Inspector

The Inspector is a window-oriented version of **describe**. It lets you investigate the insides of data structures in the same way that the previous tools let you look inside the process of function execution. You can enter the Inspector through the main System Menu, by typing SYSTEM I, or by evaluating (**inspect** *argument*). When you ask to inspect a particular object its components are displayed. What the components are depends on the type of object. For example, the components of a list are its elements, and those of a symbol are its value binding, function definition, and property list. The components of objects are mouse sensitive; you can click on a component to inspect it, and so on, recursively. You can also inspect a new object by typing it into the window in the upper left-hand corner. A typical Inspector window looks like Figure 27.

Using the Inspector is straightforward; the mouse documentation line tells you what you can do. However, there are several items that are not self-evident. Several keyboard commands are active when you are not in the middle of typing a LISP form.

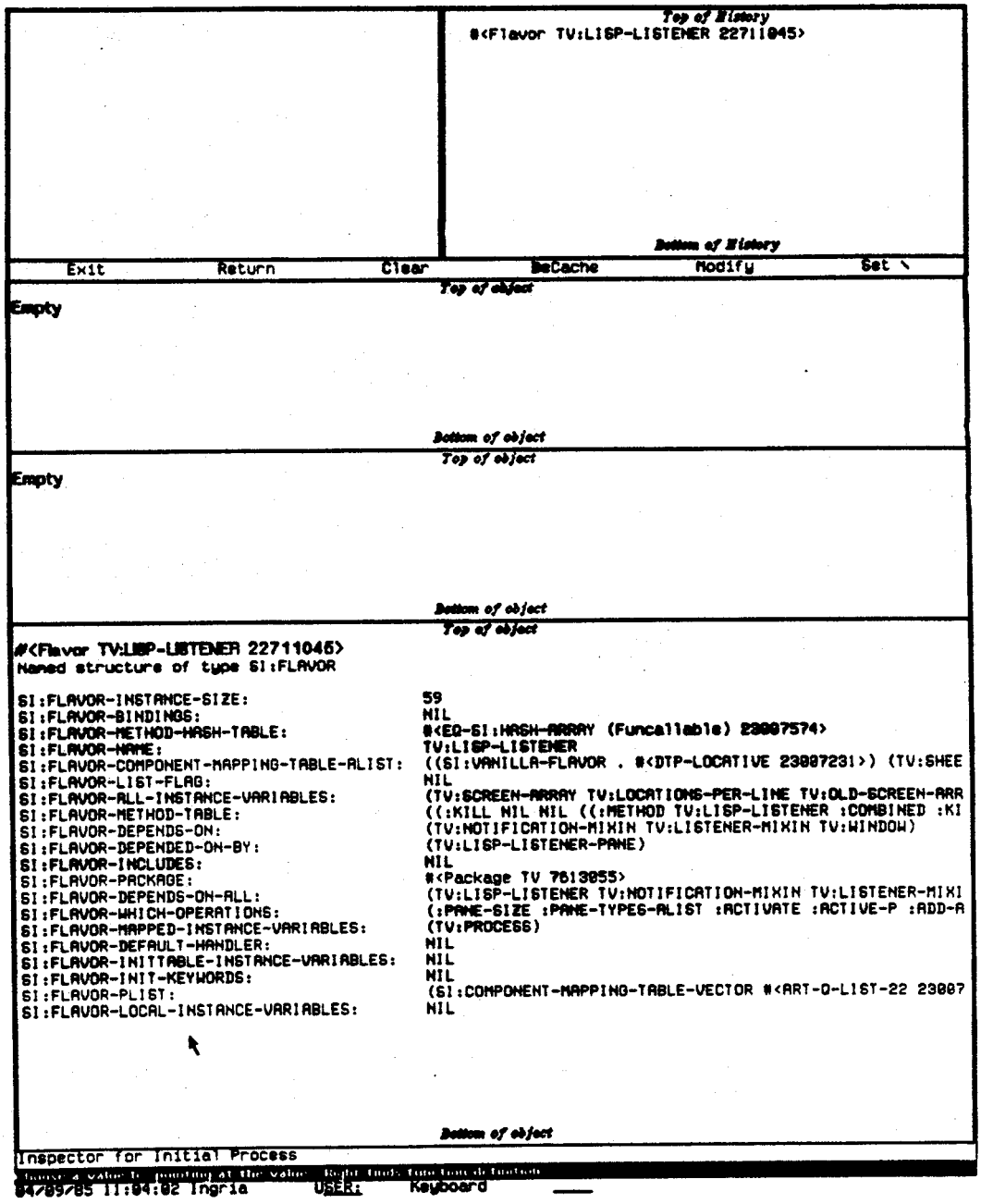


Figure 27. An Inspector Window

CTRL-Z Exits and deactivates the Inspector.

(BREAK) Runs a Break loop in the typeout window.

(QUOTE) Reads and evaluates a form, rather than inspecting it.

Note: if you change the value or structure of an object, the program will not know about it until you **Decache** the object and redisplay it.

8. The UNIX System

The UNIX System is an alternate operating system available from LMI as a separately priced option. It was originally developed at Bell Laboratories in the early 1970s. UNIX is widely used at both commercial and academic sites and is recognized as a standard for high-quality, multiuser operating systems.

We supply Version 7 UNIX (System 5 UNIX will be available in third quarter 1985.), which supports the VI editor, the C programming language, and multiple users. You can have up to three users on UNIX at once, with upgrades available to eight or sixteen users.

8.1 Logging In and Out

UNIX will present you with a login prompt that looks like this:

```
:login:
```

Just respond with your login name. You may also have to provide a password; if so, UNIX will prompt for it.

```
:login: mrc
Password:
```

Your password will not echo on the terminal screen.

To log out, either press **CTRL-D**, or type **logout** followed by (RETURN).

8.2 Filenames and Directory Structure

UNIX filenames should be fourteen or fewer characters long. UNIX is case sensitive and the usual convention is to use lowercase.

The UNIX file system is hierarchically organized with directories arranged in a tree structure. The top, or root, directory is denoted by "/". A typical UNIX file system is shown in Figure 28.

To specify a particular file or directory as the argument to a command you start at the root and work downward, separating directories with additional slashes (/). For example, to refer to the **recipes** subdirectory of a user directory named **ingria**, which is itself a subdirectory of the **usr** directory, you would need to type **/usr/ingria/recipes**.

When you log in you are automatically placed into your "home" directory. This is where your files will be created and live by default. Unless you actively switch to another directory this will be your working directory and you will be able to refer to files in it by filename only; you don't need to give the whole pathname.

To change directories, use the **cd** (change directory) command followed by the pathname of the destination directory. You don't always need to specify the full pathname; you can reach subdirectories by giving the pathname only downward from your current location in the structure. For example, if you are currently at **/usr/mrc/doc**, you can reach **/usr/mrc/doc/intro** by typing:

```
# cd intro
#
```

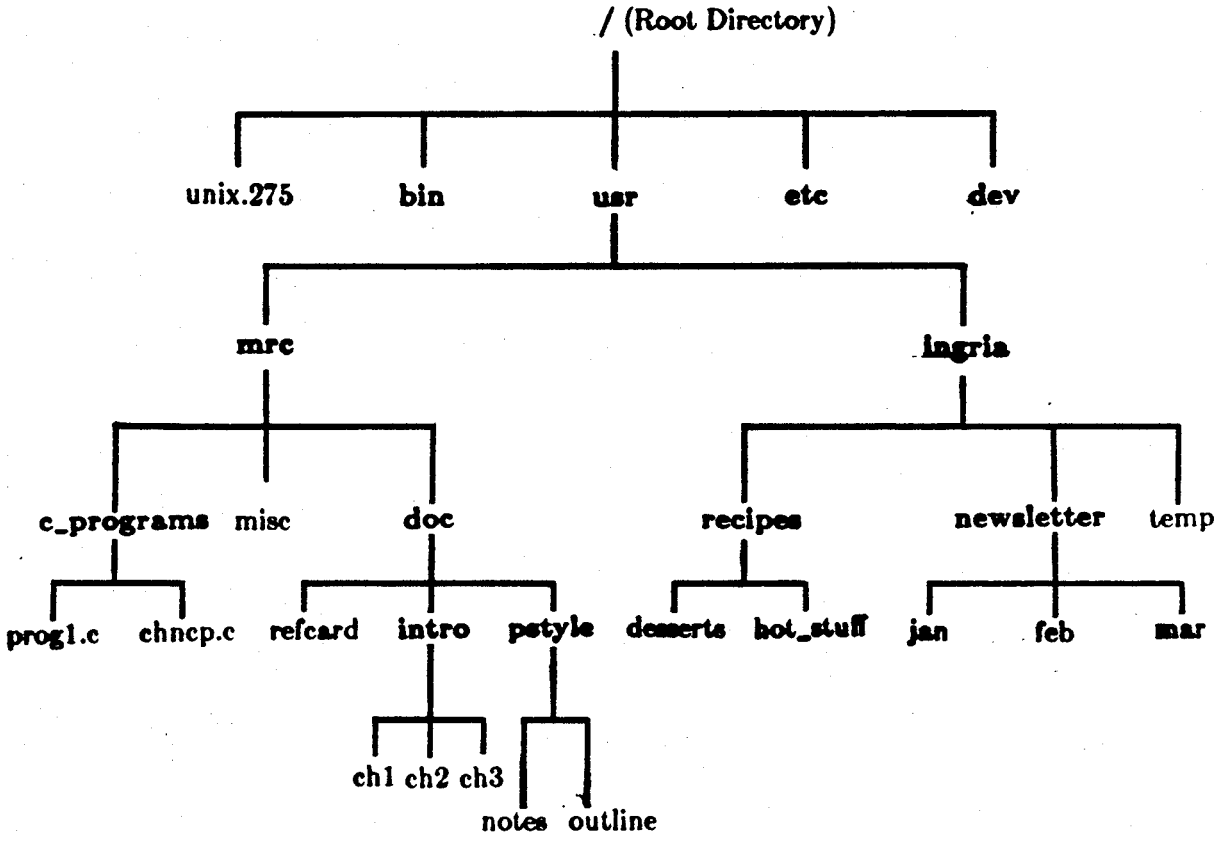


Figure 28. A typical UNIX file system

In addition .. (two dots) is a shorthand way to refer to a parent directory. So that if your current directory is **intro** and you want to move "over" to **pstyle** since they share **doc** as a common parent, you can type:

```
# cd ../pstyle
#
```

This says, move "up" to **doc**, and then "down" to **pstyle**. Once you switch directories you can then refer to files in your new working directory by filename only.

Since UNIX, unlike the Lambda, can be a protected system, there are probably directories that you can't enter and/or read from. If you try, UNIX will respond with an error message.

Some directory commands are:

mkdir *dirname*
: Creates a subdirectory.

cd *pathname*
: Changes your working directory to *pathname*; if you don't specify a pathname this will bring you back to your home directory.

rmdir *pathname*
: Removes a directory; can only be used to remove empty directories.

pwd
: Displays the pathname of your (current) working directory. This is excellent to use if you are exploring and get "lost".

8.3 Some System Commands

The following are just some basic system commands to allow you to get started using UNIX.

cp *file1 file2*
: Copies *file1* into a file named *file2*

mv *file1 file2*
: If both *file1* and *file2* refer to the same directory, then *file1* gets renamed to *file2*. If the names refer to different directories, then *file1* is moved to the directory and filename specified by *file2*.

rm *file(s)*
: Permanently deletes the named file(s).

cat *file(s)*
: Displays the contents of a file on the screen.

more *file(s)*
: Displays the contents of a file on the screen, but stops after each screenful to give you a chance to read the material. To continue and see "more" press (SPACE); to stop the display, type **q**. This is usually more useful for reading text than **cat**.

man *commandname*
: Displays the relevant material from the UNIX reference manual. This is not very useful for learning UNIX—you must know pretty much what you are looking for before you can find it--but it is an excellent reference tool.

man -k *topicname*
: Displays a choice of commands that relate to the given topic. This is the closest that UNIX gets to providing conceptual, noncommand-oriented help.

8.3.1 Wildcards

UNIX has two wildcards, * and ?, that let you specify whole groups of files as the argument to a system command. They can save you a lot of typing, and they can get you into a lot of trouble quickly; use them with care. ? substitutes for exactly one character in a filename. For instance,

```
rm ?at.doc
```

will remove files named **bat.doc**, **cat.doc**, and **mat.doc**. * substitutes for any number of letters. So,

```
rm *at.doc
```

will also delete **splat.doc** and **frat.doc**.

8.4 The Extended STREAMS Interface

With UNIX on your Lambda you can make UNIX and LISP communicate and work for each other by using the Extended STREAMS software included with the UNIX processor.

Extended STREAMS allows you to switch easily from the ZetaLISP-Plus environment to the UNIX environment, and even use LISP from UNIX and call UNIX tools from LISP.

The LISP Processor and the Motorola 68010 processor running UNIX can run concurrently, sharing various system resources. Both processors may access the disk through a common interface. Memory is divided among the host processors based on system configuration information.

There are three ways to use the Extended STREAMS software:

- over Chaosnet.
- using streams and the Share TTY feature.
- directly, accessing shared physical memory.

8.4.1 Chaosnet

You can use Chaosnet between LISP and UNIX sharing a bus just as you would use it between two completely independent machines. In addition, using Chaosnet this way, you can share information and processing responsibility between the LISP processor and any UNIX host.

8.4.2 Share TTY

This allows you to use LISP streams to pass data and control between LISP and UNIX. The (SYSTEM) U command that turns your monitor into a UNIX terminal window was implemented using the share TTY facilities.

8.4.3 Shared Physical Memory

For high-volume, time-critical LISP/UNIX communication, you can use the shared memory area. This is both the most low level, and quickest of the options.

For more detailed information on functions and procedures see *Interprocessor Communication: The Extended STREAMS Interface.*

9. Editing in UNIX

UNIX comes with two editors: they are Ed, a line editor, and VI, a full screen editor. You can write programs or edit files using either one, however since VI is a full-screen editor, which many people find easier to learn and use, it is the one discussed here.

Note: Currently VI can only be used in "open" mode from the high-resolution monitor; if you are using VI from the high-resolution monitor, rather than a terminal like a Z29, use (ALT MODE) as your "escape" key.

Unlike ZMACS, VI has modes. There is an insert mode and an editing mode. When you are in insert mode you can type to the editor just as if it were a typewriter. In edit mode you can move freely around the text and edit it. Because VI distinguishes between characters and commands by the mode it is in, most VI commands are simple single or double character keystrokes without any CTRL or other modifiers.

To use VI type:

```
# vi filename
```

If this is a new file, VI will tell you so, otherwise it will get the appropriate file and put it in a buffer for you to edit.

To exit VI, from command mode type:

```
:wq
```

If you don't want to save your work you can exit by typing:

```
:q!
```

To save and continue editing use `:w` *filename*. The *filename* argument is optional; if you don't give one VI will overwrite the file with the same name as your VI buffer.

9.1 Inserting Text, Moving Around

To go into insert mode press `i`. To leave insert mode and go back to command mode press (ESC). When you use `i` to insert text, text is inserted to the left of the cursor. There are other insertion commands that begin inserting text differently. Some of the more common are:

`I` Inserts text at the beginning of the current line.

`a` (append) Inserts text to the right of the cursor.

`o` Opens a new line for text below the current line.

`O` Opens a new line for text above the current line.

All commands that leave you in insert mode must be ended with (ESC) when you want to return to command mode.

VI offers a wealth of cursor commands; some are presented in the chart below:

In Line Movement:

(SPACE) Moves cursor one character to the right.

`l`	Moves cursor one character to the right.
`BACK SPACE`	Moves cursor one character to the left.
`h`	Moves cursor one character to the left.
`*w`	Moves cursor to the beginning of the next word.
`e`	Moves the cursor to the end of the current word.
`*b`	Moves cursor back a word.
`*0`	Moves cursor to the beginning of the current line.
`*$`	Moves cursor to the end of the current line.

Between Line Movement:

`-`	Moves cursor to the beginning of the previous line.
`CTRL-P`	Moves cursor up one line, but in the same column.
`RETURN`	Moves cursor to the beginning of the next line.
`CTRL-N`	Moves cursor down one line, but in the same column.
`*H`	Moves cursor to the top line of the screen.
`M`	Moves cursor to the middle line of the screen.
`*L`	Moves cursor to the bottom line on the screen.

The commands with asterisks next to them can take numeric arguments. For instance, **3w** will move you 3 words ahead, **4L** moves the cursor four lines from the bottom of the screen, etc.

There are also commands that enable you to move through your file more quickly by scrolling or paging.

`CTRL-D`	Scrolls down through a file.
`CTRL-U`	Scrolls up through a file.
`CTRL-F`	Pages forward through a file.
`CTRL-B`	Pages backward through a file.

Scrolling gives more of a feeling of continuity than paging because it leaves a larger overlap between screens.

You can also use the search commands to move longer distances.

9.2 Deleting and Moving Text

VI offers several ways to change already written text. The simplest way to correct minor mistakes is to use the `RUB OUT` or `DELETE` key. This works even while you are in insert mode. Another way is to switch to writeover mode. The "**R**" command replaces text character for character until you go back to edit mode with `ESC`.

Many of the cursor commands take editing prefixes, so that instead of ***moving*** that distance, they will *edit* on that amount of text.

`d`	Deletes text.
`c`	Changes text and leaves you in **Insert Mode**.
`y`	Yanks, or copies, text.

For instance, **3dw** deletes the next three words; **cw** will change the current word and then leave you in insert mode to add more text if you want. It is equivalent to typing **dw** followed by **i**. The line commands are a slight exception to this general rule: **cc** changes a whole line and **dd** deletes it.

The **d** commands do not actually delete the text, they place it in a buffer from which you can "pull" it back. So, if you want to *copy* text use "**y**", and if you want to *move* text to a new location use "**d**".

p Pulls the text back and places it after the cursor.

P Pulls the text back and places it before the cursor.

9.3 Searching

VI's search facility allows you to look for a particular word or phrase anywhere in your text. This is a good way to move large distances or find phrases that you want to change without going through the text line by line. The only place that VI will not be able to find something is on the current line. This should not be a problem, because you will always be able to see that line on your screen.

/pattern ESC is VI's basic search command. VI will move the cursor to the beginning of the first occurence of the pattern you are searching for. It will search downward in your text from the current cursor position, loop to the top and continue searching until it comes back to your original place. If the pattern does not occur in your text the cursor will remain where it was, and VI will display a message on the bottom of your screen.

n Asks VI for the next occurence of the pattern downward in the text.

N Asks VI for the next occurence of the pattern upward in the text.

You can also use the search facility to define arbitrary regions of text. For instance, **d***/foofraw* ESC will delete everything from the current cursor position to the next occurence of *foofraw*.

9.4 Other Editors

Mince, a version of EMACS, is also available under UNIX. For further information, see *Introduction to Mince.*

Appendix A. A Sample INIT File

An example of a portion of an init file containing useful options is given below. Superscript numerals, such as[1], refer to the numbered sections following the sample forms, not text you type in on your Lambda.

```
;; -*- Mode LISP Mode -*-

(login-forms[1]
            zwei:(set-comtab[2] *standard-comtab*
                                '(;; Set up hand keys to work like arrow keys
                                #\hand-up com-up-real-line[3]
                                #\hand-down com-down-real-line
                                #\hand-left com-backward
                                #\hand-right com-forward
                                ;; Make Roman keys do useful things
                                #\roman-i com-backward-paragraph
                                #\roman-ii com-forward-paragraph
                                #\roman-iii com-beginning-of-line
                                #\roman-iv com-end-of-line
                                ;; Make delete do rubouts
                                #\delete com-rubout))
            #-common[4]
            (progn
                  ;; Set base to decimal
                  (setq-globally[5] base 10.)
                  (setq-globally ibase 10.)
                  ;; Get rid of that annoying trailing decimal point
                  (setq-globally *nopoint t))
            )

(tv:white-on-black)[6]

(push[7] '(tv:black-on-white)[6] logout-list)
```

1. **login-forms** is a LISP macro that evaluates the forms passed to it and arranges for them to be undone when you logout. Forms passed to **login-forms** should have an **:undo-function** property. You will need to add this property yourself to any functions of your own that you call in your **.INIT** file. However, common ZetaLISP function, including **setq**, **pkg-goto-globally**, **setq-globally**, **add-initialization**, **deff**, **defun**, **defsubst**, **macro**, **advise**, and **zwei:set-comtab**, already have an **:undo-function** property. If one of the forms in your **login-forms** does not have an **:undo-function** property, the following message will be printed when you login:

```
[A name form is supposed to be undone at logout, but this is not implemented.
The form's effects will be permanent.]
```

If you define an **:undo-function** property in your **LOGIN.INIT** file, the definition must appear *earlier* in the file than the **login-forms** that contains the function. See the *LISP Machine Manual*, pages

801–802 for a discussion of **login-forms** and an example of adding the **:undo-function** property to a function.

Another way to arrange for a command to be undone at logout is to explicitly add a LISP form to the **logout-list** variable. **logout-list** is a variable bound to a list of LISP expressions that are evaluated at logout. (See **7.** below.) Conscientious use of **login-forms** and **logout-list** will help clean up the Lambda environment for other users when you logout.

2. **zwei:set-comtab** is a LISP form that sets up initialization options for ZMACS. You may also notice the syntactic device here of placing a package prefix (**zwei** in this case) outside of a form, when all the symbols in the form are to be evaluated as symbols in that package. This syntax is supported in ZetaLISP but not in Common LISP.

3. Names of ZMACS commands that you wish to specify in your initialization file should be preceded by **com-** and should be written as single symbols, with hyphens (-) appearing where spaces appear in the normal name of a command. If you know the modifier key sequence that invokes a given command but do not know the name of the command, you can use (HELP) C in ZMACS, followed by the modifier key sequence, to find out the name of the command.

4. **#-** and **#+** are useful conditionalization macros. There is a global variable, called ***features***, that contains a list of features of the current LISP environment. **#+** and **#-** allow you to conditionalize LISP forms based on the presence or absence of a feature in ***features***. **#+***feature* evaluates the following LISP form if *feature* is present in ***features***. **#-***feature* evaluates the following LISP form if *feature* is *not* present in ***features***. In both cases, if the test fails, the form is not evaluated at all, so no value is returned. The example form here sets the base to 10 if the LISP environment does not contain **:common**, i.e. if Common LISP is not supported. (**:common** indicates the presence of Common LISP features.)

You can add features of your own to ***features*** to conditionalize you own code. Symbols added to ***features*** are normally in the keyword package. That is, they begin with a colon (:).

5. **setq-globally** is used to set the global binding of a variable. This binding is accessible not only in the context in which your **.INIT** file is loaded, but "globally", so that variables set by **setq-globally** have that value for all programs that use them.

6. **tv:white-on-black** and **tv:black-on-white** control the display mode of the Lambda. (See the discussion of display mode above on page 9.) You may find it useful to put one or another of these forms in your **.INIT** file if you prefer one of these display modes.

7. Note here the scheme of explicitly placing forms to be evaluated at logout on **logout-list**. You should **push** forms onto **logout-list** rather than **setqing** **logout-list** to a list of forms, since other parts of your **.INIT** file may already have caused other forms to be placed on **logout-list**. Using **setq** would remove the earlier forms, while **push** merely adds the new forms.

Appendix B. Help and Keyboard Facilities

This appendix mentions some system help facilities and built-in keyboard functions that you should find useful while programming, and when you have minor problems with the Window System.

⟨HELP⟩ The ⟨HELP⟩ key can provide useful brief messages in several contexts; the most common are mentioned below. You should also always try using ⟨HELP⟩ when inside an application (for example, the debugger). Often ⟨HELP⟩ is context sensitive and will provide information on the particular application that you are using.

The following two help requests can be made from anywhere in the Window System environment.

⟨SYSTEM⟩ ⟨HELP⟩
Provides a list of all currently available programs that have been defined to be "systems" and assigned a letter to use in combination with the ⟨SYSTEM⟩ key; for instance the LISP Listener is assigned to ⟨SYSTEM⟩ **L**.

⟨TERMINAL⟩ ⟨HELP⟩
Provides information on the use of many of the function keys.

⟨TERMINAL⟩ takes several arguments. Some of the more useful are:

⟨TERMINAL⟩ **C**
Complements the display mode of the screen. That is, switches from black-on-white mode to white-on-black mode, and vice versa.

⟨TERMINAL⟩ **M**
Complements **MORE** processing. That is, turns **MORE** processing on if it is off, and vice versa. When **MORE** processing is enabled, output pauses when text reaches the bottom of the screen, and the legend **more** is displayed, just as in a typeout window.
You may also explicitly turn **MORE** processing on or off:

⟨TERMINAL⟩ **1M**
Turns **MORE** processing on.

⟨TERMINAL⟩ **0M**
Turns **MORE** processing off.

⟨TERMINAL⟩ **Q**
Hardcopies the screen on the default output device.

⟨TERMINAL⟩ **1Q**
Hardcopies the selected window on the default output device.

⟨TERMINAL⟩ ⟨CLEAR SCREEN⟩
Clears all visible windows, does not alter the LISP environment.

⟨TERMINAL⟩ ⟨HOLD OUTPUT⟩
Exposes the window currently producing the **Output Hold** run state on the who line.

Other useful function keys are:

⟨ABORT⟩ Halts the process and brings you back to command level, but only if the process is waiting for keyboard input.

CTRL-⟨ABORT⟩
Halts the process and brings you back to command level immediately. This is good to use if you've started something you didn't mean to, or if the current process appears to be hung.

CTRL-META-⟨BREAK⟩
Puts process immediately into the error handler.

⟨CLEAR INPUT⟩
Forgets the current, unevaluated input.

⟨CLEAR SCREEN⟩
Clears the currently selected window, does not alter the LISP environment.

From the LISP Listener **CTRL-**⟨HELP⟩ produces a listing of ZMACS commands that can be used to edit LISP expressions while in the Listener. You can get information on the use of a function or a listing of its arguments by typing an open parenthesis, then the function and then pressing either **CTRL-SHIFT-D** or **CTRL-SHIFT-A**, respectively.

From ZMACS, ⟨HELP⟩ ⟨HELP⟩ provides a listing with brief explanations of the help options available in ZMACS.

Appendix C. Further Reading

Here are some suggestions for more reading on topics covered in this manual. Most of these are manuals written or distributed by LMI; some are high quality books that are readily available.

LISP Culture

More information on the history of the LISP Machine can be found in these two papers:

Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, and Daniel Weinreb "The LISP Machine", in Patrick Henry Winston and Richard Henry Brown, eds., *Artificial Intelligence: An MIT Perspective, Volume 2*, The MIT Press, Cambridge, Massachusetts, 1982, pp. 345–373. [Discusses the CONS.]

Richard Greenblatt, Thomas F. Knight, Jr., Jack Holloway, David A. Moon, and Daniel L. Weinreb "The LISP Machine", in David R. Barstow, Howard E. Shrobe, and Erik Sandewall, eds., *Interactive Programming Environments*, McGraw-Hill Book Company, New York, 1984, pp. 326–352. [Discusses the CADR.]

Chapter 1

The hardware information is relevant to some degree.

Chapter 2

The *Release Notes* included in your latest software release provide further information on booting.

Chapter 3

See Chapter 24 of the *LISP Machine Manual* for further discussion of pathnames.

Chapter 4

The LISP Machine Manual is, of course, the final authority on ZetaLISP-Plus. For a conceptually oriented introduction to LISP read:

David S. Touretzky, *LISP: A Gentle Introduction to Symbolic Computation*, Harper and Row, 1984.

Lambda Users Supplement to Touretzky is an LMI pamphlet enumerating the differences between the dialect of LISP used in this book and ZetaLISP. LMI will send this free upon request.

Two other books that are especially useful as sources of Common Lisp information are:

Patrick H. Winston and Berthold K. P. Horn, *LISP*, Addison-Wesley, 1984.

Guy L. Steele Jr, *Common Lisp: the Language*, Digital Press, 1984.

Chapter 5

The ZMACS Introductory Manual and the forthcoming *ZMACS Reference Manual* both provide more detailed discussion of the capabilities of ZMACS as a programming tool.

Chapter 6

The Window System Manual discusses aspects of ZetaLISP and the LISP Machine that you need to know about in order to program using windows. *Introduction to the Window System* provides a conceptual introduction.

Chapter 7

The LISP Machine Manual and the forthcoming *LISP Machine Programming: Style and Technique* both discuss debugging in greater depth.

Chapters 8 and 9

The *UNIX I* and *UNIX II* volumes are essentially reference manuals and cover UNIX and C programming in detail. For a basic introduction to the C language try:

Brian W. Kernigan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

All of the following books provide an introduction to the UNIX operating system; the first title is somewhat broader and more theoretical, the next two more practical.

James R. Groff and Paul N. Weinberg, *Understanding UNIX: A Conceptual Guide*, Que Corporation, 1983.

Rebecca Thomas and Jean Yates, *A User Guide to the UNIX System*, Osborne/McGraw-Hill, 1982.

S. R. Bourne, *The UNIX System*, Addison-Wesley, 1983.

Concept Index

D

E

F

M

N

O

P

R

S

T

U

V

W

Y

Z

LISP Index

Introduction to the Lambda:
A Programmer's Guide to Getting Started
[Preliminary Version: August, 1984]

Meryl Cohen
Robert Ingria

Published by LMI 6033 W. Century Blvd. Los Angeles CA 90045
USA

This is a *preliminary* version of the *Introduction to the Lambda*.

Please help us to make LMI documentation work better for you! Send comments via your customer dialup mail line to Sarah Smith (username SWRS) or by U.S. Mail to:

Dr. Sarah Smith
Manager, Documentation
LMI
1000 Massachusetts Avenue
Cambridge, MA 02139

Table of Contents

Preface

Purpose of This Manual

This manual is an introduction to the LMI Lambda programming system. It concentrates on software rather than hardware facilities, but hardware issues are mentioned when necessary. By the time you finish reading this Introduction and following the examples at the terminal, you should be able to use the Lambda as a serious programming tool, limited only by your ability as a LISP programmer.

LISP Machine Inc. welcomes all comments and suggestions for improvement; contact Sarah Smith, Documentation Manager, at (617) 876-6819.

Audience

The *Introduction to the Lambda* is written to meet the needs of most beginning users of this machine. No previous LISP Machine experience is assumed, but some familiarity with basic LISP concepts, syntax, and terminology is.

Other Information Resources

This manual is intended neither as a guide to LISP, nor as a comprehensive guide to the Lambda. For more information on both the hardware and software in the Lambda system consult the sources described below.

More Manuals

Appendix C contains a chapter by chapter guide to manuals or books that will give you more detailed information on the material covered in each chapter. With a few exceptions these materials are provided when you purchase the Lambda.

Telephone Support

LISP Machine Inc. provides free telephone software and hardware support to its clients. This service is available during normal business hours (Eastern Standard Time) Monday through Friday. To get help just call (617) 876-6819. You can also dial in to our VAX computer and leave mail for customer service staff. For an account call Dawna Provost, Manager of Customer Service, at the above number.

Classes

LMI currently offers beginning and intermediate level ZetaLISP-Plus classes. Other classes will be added in the future. For information and reservations contact Pam Renowden (213) 642-1116.

Documentation Conventions

There are many operations that can be performed at the LISP Machine console using the keyboard and mouse. In order to describe these operations unambiguously, the following notational conventions have been adopted in this document.

Modifier keys are blue keys with black labelling. Like the SHIFT key on an ordinary terminal keyboard, these keys do not actually transmit a character, but rather modify a printing character. The proper usage of such a key is to depress it simultaneously with the key to be modified. The following modifier keys are available on the LISP Machine keyboard:

TOP
GREEK
HYPER
SUPER
META
CTRL

Function keys are blue keys with white labelling. These keys actually transmit a character. The proper usage of such a key is to depress it and then depress the following character. Function keys are indicated by the following box: ⟨_____⟩. Some common function keys are:

⟨ABORT⟩
⟨TERMINAL⟩
⟨SYSTEM⟩
⟨HELP⟩
⟨END⟩
⟨STATUS⟩

Mouse buttons are the buttons on the mouse. [ ] indicates clicking any mouse button. [ ] with one of the letters L, M, or R indicates clicking the left, middle, or right mouse buttons, respectively. Any of these button indicators can appear with a following 2, indicating that that button should be clicked twice quickly in succession. For example:

[L]	indicates that the left mouse button should be clicked
[R2]	indicates that the right mouse button should be clicked twice

Mouse-sensitive items are objects that appear in windows that may be pointed at with the mouse. A mouse sensitive object appears in a box, such as [Split Screen]. To indicate that a particular mouse button should be clicked while the mouse is pointing at a mouse-sensitive item, a mouse button indicator appears before the mouse-sensitive item. For example:

[R2][Split Screen]	indicates that the right mouse button should be clicked twice while the mouse is pointing at

Split Screen

Function Conventions

to be supplied

History of the LISP Machine

The LISP Machine was originally designed at the Artificial Intelligence Laboratory at MIT to provide an efficient, single-user programming environment for the development of large LISP programs. The need for such an environment arose from the problems of running multiple LISP programs on time-sharing systems. The earliest computers on which LISP was developed were expensive machines that were originally designed to be single-user systems. Since it was not cost-effective to devote such machines to only one user at a time, time-sharing systems were developed, which allowed several researchers to use a single machine at the same time. Project MAC, at MIT, developed such a time-sharing (or "time-stealing", as it was then called) system, called ITS, for PDP-6 and PDP-10 computers. In addition, a new dialect of LISP, MacLISP was developed, based on LISP 1.5. As more and more researchers worked on LISP programs on individual ITS machines, system resources were strained and response-time was slowed considerably, hindering program development.

The earliest computers were much more expensive than the cost of programmer time, so it made sense to have as many users on a single system as possible. However, in recent years the cost of computing time has become relatively inexpensive compared to the cost of wasted programmer time, so that it made sense to have highly efficient single-user programming environments. The availability of integrated circuits and the relative cheapness of systems with massive computing power made it possible to develop a single-user system for LISP development.

The first LISP machine was called CONS and was operational in 1975. This was succeeded by the CADR, in 1978, and the Lambda, in 1983. All of these machines were designed for compatability with AI programmiing environment at MIT. The dialect of LISP that runs on the LISP Machine, ZetaLISP, is descended from MacLISP. Many of the features of the LISP Machine are based on analogous facilities in ITS: the WHOLINE, PEEK, **MORE** processing, and the structure of the file system are all developments of ITS. The ZMACS editor is an extended version of the EMACS editor, originally written in the TECO language on ITS. The multi-tasking of the LISP machine, where a user can move from a LISP Listener to an editor to any other program, also has its origins in ITS, which also supports multiple jobs.

The Lambda may be seen as a natural outgrowth of the programming environment that had been developed at MIT: extensions to LISP that had proved useful and aspects of the ITS operating system that had provided support for program development were re-implemented in an expanded and more efficient manner, creating an optimum environment for LISP programming and development.

The LMI Lambda Philosophy

LMI has designed the Lambda to be the ideal programmer's workshop. All the best programming tools and facilities are immediately available for you to use. ZetaLISP-

Plus is an extremely powerful and natural programming language; it is easy to design and code programs the way that the problem appears to you. FLAVORS give you the power to use Smalltalk-like, object oriented programming techniques within the Lambda environment. You can use ZetaLISP-Plus as either an interpreted or compiled language. This gives you the running speed of compiled code and the ease of development of an interpreted language.

While LMI believes that ZetaLISP-Plus offers an ideal programming environment, both as a language and an operating system, we also offer other languages (Prolog, Fortran, C, and Pascal) and the UNIX operating system. You can use these separately or in conjunction with LISP.

3. The Lambda Environment

The Hardware

The hardware in your LMI Lambda can be thought of as consisting of two groups: the hardware that runs the computer, and the hardware that you use to talk to the machine. This first group includes:

- one cabinet containing:
  - a disk drive
  - a tape drive
  - Lambda boards, memory, and an optional UNIX processor.
- one terminal used for booting.

Depending upon how your site is arranged, you may never (or rarely) see this equipment. It can be physically distant from the second group of hardware—the user console. This consists of:

- a high-resolution black and white monitor.
- a medium resolution color monitor (optional).
- the AI keyboard.
- a mouse and optical pad.

The High-resolution Monitor

The high-resolution monitor included with your Lambda provides an easy-to-read display for both graphics and text applications. When text reaches the bottom of the screen it wraps around again to the top; text does not scroll. You can adjust the viewing angle by gently pushing the monitor housing. The accordion base will move easily, and then hold your new angle.

If you prefer a reverse video display, you can change from normal to reverse video (and back again) by typing ⟨TERMINAL⟩ C.

PICTURE of a screen-display approx. half-size with arrows and signs to various features.

Information Lines

At the bottom of the screen are two lines that always provide you with information about the status of the system. The reverse video upper line is the mouse documentation line. This tells you what the mouse buttons do in the current environment. If you are looking at a pop-up menu, this line tells you the effect of the currently "boxed" choice. (For a more complete discussion see Chapter 5, The Window System.)

The line below the mouse documentation line is called the "who line". This contains several kinds of important information. The leftmost entry is the current date; the one just to the right of the date gives the time. Next in line is the username of the person currently logged in. Following this is the name of the current package (for details see Packages in Chapter 3). Next is usually a description of the state of the current process. The area on the far right displays different information depending on the circumstances. Usually the area is empty. However, when you are reading a file from disk or writing to disk, it will display the filename and the progress of the operation. When the mouse hasn't been clicked or a key pressed for at least 5 minutes, it displays the message `console idle for X minutes`.

Below the who line to the middle right are three small lines called run bars; they can give you a sense of what the Lambda is doing. You will see the leftmost one when garbage collection occurs, the middle one during disk accesses, and the rightmost bar while any program is being run. You should see the middle bar when reading and writing files; if you see it too frequently at other times there may be a problem with your machine. You should see the rightmost one quite often. One way to tell if your machine has crashed is to move the mouse rapidly and look for the run bar to appear. If you don't see anything, your machine is probably no longer among the living, also, the clock inthe left corner will have stopped.

The Color Monitor

The color monitor can be used in either a color or a black and white mode. It is designed so that you can distinguish different layers of an image. This makes it useful for both CAD/CAM and AI vision-oriented work. (For complete documentation see *LMI Lambda Medium Resolution Color Board and Monitor.*)

The AI Keyboard

The AI keyboard that comes with your Lambda has many more keys than a typical typewriter or terminal keyboard; it can be quite imposing when you first encounter it. There are three types of keys on the keyboard: character keys, modifier keys, and function keys.

PICTURE: Keyboard picture with 3 different tones of shading.

Character Keys

These are the small grey keys in the middle of the keyboard (********** in figure ****). These include standard typewriter keys as well as other keys which produce mathematical symbols, or have special uses in certain programs.

Modifier Keys

The modifier keys are the blue keys near the bottom on the right and left with black lettering (******** in figure ****). Almost all of these work like shift keys: you press one of these keys together with a character key to modify the behavior of the character key. For instance, the TOP modifier key and a letter key will produce the symbol printed above the letter on the character key. The two exceptions to this are: the small Caps Lock and Alt Lock keys on the left and right respectively.

Caps Lock acts like the Shift Lock on a typewriter, except that it only shifts the letter keys. For instance, if you activate the Caps Lock and then press the "1" key you will still get a "1" (one), not a "!" (exclamation point).

Alt Lock currently has no effect.

Function Keys

The function keys are the large blue keys with white lettering. (The ********** in the figure.) Some, like ⟨TERMINAL⟩ and ⟨SYSTEM⟩, are used throughout the Lambda environment; others are used in specific programs, and still others currently are not assigned to any function. These keys will be discussed more fully in the context of individual programs. (Appendix B contains a reference to all the ⟨TERMINAL⟩ and ⟨SYSTEM⟩ key commands.)

Use the ⟨RUB OUT⟩ key to delete single characters to the left. The ⟨DELETE⟩ key is currently unassigned.

Mouse and Optical Pad

The Mouse, like the keyboard, is an input device, a way for you to speak to the computer. LMI uses an optical, rather than a mechanical, mouse because they are much more reliable (the wheels don't fall off... there aren't any). The Mouse's companion is the optical pad, a rectangular silvery mat with blue grid lines. The little eyes on the

bottom of the mouse use those lines to figure out where it is. You should orient the optical pad to match your monitor screen.

The Mouse can transmit information to the computer in two ways: through the three buttons on its back, and by changes of position on the optical pad. Information from the buttons usually calls up or chooses items from the Pop-up mouse menus. The mouse documentation line usually tells which buttons are active and what they will do. Position information is indicated on the screen by a small arrow called the mouse cursor. Programs such as ZMACS and the screen editor use this to allow you to move text and windows around the screen. (For more information on the Mouse and pop-up menus see Chapter 5, "The Window System".)

Software Features

The Lambda system includes software tools that together create a sophisticated programming environment. These include:

- ZetaLISP-Plus, a powerful and flexible dialect of LISP that includes FLAVORS.
- ZMACS, a text editor tailored to take full advantage of the Lambda environment.
- The Window System, a user-friendly, yet unobtrusive user interface.
- The Error Handler and the Inspector, two high-function debugging aids.
- UNIX, a popular operating system used primarily in academia and software development. (optional)
- Other Languages: (optional)
  - Prolog
  - Fortran
  - C
  - Pascal

The greatest strength of the LMI Lambda lies in its modeless operating environment. All the software is instantly accessible; you can never get "stuck" someplace that you don't want to be.

This manual introduces you to the Lambda's software and to how its different subsystems interact, so that you can develop your own personal working environment.

4. Starting Up

Booting the Lambda

To "boot" a computer means to clear out memory and restart the machine with the default environment. If you are from a mainframe environment, you have probably never done this unless you are a computer operator, or a systems programmer. If you come from the microcomputer world, you do this every time you start the machine. The Lambda fits in between these two groups, so, depending upon your site, you may have to boot every time you use the machine or only very rarely. The information below explains the standard procedure to boot your Lambda.

NOTE for Lambda-Plus users: The console method will boot only the LISP processor; if you need to boot UNIX also, boot from the SDU. If only LISP has crashed, but it won't respond to a console boot, you will have to bring down UNIX before you can boot from the SDU.

There are two main reasons to boot the Lambda: either your machine has crashed, or you want a clean LISP environment to work with. The method explained below that allows you to boot from the user console may not work if your system has crashed. If so, you will need to boot from the SDU (Z29), and the procedure is more involved. For more detailed instructions see *The LMI Lambda Field Service Manual.*

Cold Booting

To cold boot press all five of the following keys at once.

CTRL-META-CTRL-META-(RUB OUT)

Note that you need to press *both* CTRL and META keys.

After you boot, you will notice that there are active run bars in the lower right quarter of the screen. Do not use the keyboard or the mouse until the run bars have returned to their usual location and the message cold-booted appears on the who-line.

Warm Booting

A warm boot resets the software, but does not clear memory. The only time to use this is when the machine crashes, or is behaving strangely and you can't save your files. Once you warm boot, save your files immediately and then cold boot. Warm booting does not leave the Lambda in a very stable condition.

To warm boot press the following keys all together.

CTRL-META-CTRL-META-(RETURN)

How and When to Log In

Under many circumstances, especially while you are learning to use the Lambda system, you will not need to log in. Since the Lambda is a single-user, low-security system, anyone can walk up to a machine and just begin using it. You do not need to have an account set up for you. You cannot carry out file operations unless somebody (anybody) is logged in, but if you encounter such a situation the Lambda will prompt you to log in and then carry out your request.

The main reason to log in is to activate any functions or defaults that you have placed into your lispm.init file. This file can help you setup the Lambda environment to your satisfaction. A sample file with some common defaults is in Appendix A.

To log in, you should get to a LISP Listener (use (SYSTEM) L), then type:

```
(login "loginame")
```

loginame can be any name that makes you happy. If you don't have a `lispm.init` file the system will look for one and get very unhappy if it can't find one; to prevent this, login like this:

```
(login "loginame" t)
```

To logout type:

```
(logout)
```

When a new person logs in, any previous user automatically gets logged out by the Lambda.

Directory Creation

You will probably also want to have your own directory to store files in. If you want the Lambda to be able to locate your lispm.init file, your directory name and your login name should be identical.

Again, it is not strictly necessary to have your own directory; you can store files in any directory you want since this is an open system, but it does help organize matters.

To create a directory, type:

(fs:create-directory "*dirname*")

You can create a sub-directory by typing:

(fs:create-directory "*dirname.subdir*")

Note that it is impossible to ever delete a top level directory, so you shouldn't create one unless you are sure you need one.

5. Using the LISP Listener

After you have booted your Lambda, the first environment the computer will present you with is a LISP Listener. You should see the words

```
Lisp Listener 1
```

in the bottom left corner of your console screen. (If you are not using a freshly booted Lambda, then your screen may not look like this. Get yourself a fresh LISP Listener by pressing ⟨SYSTEM⟩ CTRL-L. Your LISP Listener will have a number higher than one, but you will be able to follow the material in this chapter equally well.)

The LISP Listener is the Lambda sub-system that probably most closely resembles other LISP systems you have used. You can type LISP expressions to it and it will evaluate any expression as soon as it is completed. On the most basic level the LISP Listener consists of a read-eval-print loop. You can use it in this simple way or you can take advantage of the many additional features that ZetaLISP-Plus offers. Some of these additional features are presented and explained in this chapter. You can simply read the material if you want, but most people learn best when they practice, so you will probably get more out of the following material if you sit at the Lambda console and try out the material as it is presented. (Be adventurous! Don't be afraid to try things out for yourself to answer questions that this manual does not cover.)

The cursor in the LISP Listener is a blinking box. The LISP Listener does not have a prompt that indicates when it is ready for new input, but you can still tell easily when it is ready; the process information word in the status line will say **keyboard**. If you type before the Listener is ready for you, the Lambda saves your input and will deal with it when it can.

The default base for numbers on the Lambda is octal. Putting a decimal point after a number will force the LISP to read it as a base ten number; however, it will still respond in octal. (If you want to change the default radix, then **base** and **ibase** are the relevant variables for output and input respectively.)

Editing in the LISP Listener

Often, you realize in the middle of typing an expression that you want to change something. The ZetaLISP-Plus environment provides many editing features. You can use many of the function keys from the Listener. Press ⟨CLEAR INPUT⟩ to delete any expression that you have not yet completed; to delete an expression one character at a time use ⟨RUB OUT⟩. ⟨CLEAR SCREEN⟩ cleans off your screen but does not affect the LISP environment.

ZMACS Commands

A few ZMACS editing commands are available from the Listener. Use these to edit a LISP expression that you have typed into the LISP Listener. Some of the most useful are:

CTRL-F	Moves the cursor forward one character.
CTRL-B	Moves the cursor backward one character.
CTRL-D	Deletes the character under the cursor.
CTRL-P	Moves the cursor up one line.
CTRL-N	Moves the cursor down one line.
CTRL-K	Deletes the rest of the line after the cursor; if you press this twice, the line will also disappear.
CTRL-META-F	Moves the cursor forward one set of matching parentheses.
CTRL-META-B	Moves the cursor backward one set of matching parentheses.
CTRL-META-D	Deletes the material inside the paretheses.
CTRL-O	Opens a new line at the cursor.
CTRL-⟨STATUS⟩	Prints the contents of the kill ring.
CTRL-Y	Yanks a copy of the top item from the kill ring to the Listener.
META-Y	Cycles through the kill ring yanking copies of all items in order.

To find out more about editing commands available in the Listener, press ⟨HELP⟩. (For a more complete explanation of ZMACS commands, and syntax see Chapter 4 or the *ZMACS Introductory Manual.*)

Editing Previous Expressions

Press the ⟨STATUS⟩ key to get a list of up to the last sixteen expressions that you have typed to the Listener. CTRL-C reprints the last expression you typed without the last parentheses so that you can modify it. This can save you a great deal of repetitive typing when rectifying trivial errors.

PICTURE: screen picture showing use of status and ctrl and meta-c

META-C (which must be used immediately after CTRL-C) replaces the previous expression with progressively older expressions each time you use it.

The following are several variables that read-eval-print keeps track of that you can use as shortcuts when typing LISP forms. All of these are active while a form is actually being evaluated by the read-eval-print loop.

+ is bound to the previous form read by the loop.
++ holds the previous value of +.
+++ holds the previous value of ++.

* is bound to the result printed the last time through the loop. (If several values were printed, * is bound to the first value.)

** holds the previous value of *.

*** holds the previous value of **.

// is bound to a LIST of the results printed the last time through.

Useful LISP Functions

This Introduction is not intended to teach LISP programming; however, here are just a few functions that should make programming immediately easier to accomplish. All of the functions in this section are discussed in greater detail in the *LISP Machine Manual*.

`print-herald`

Prints information about the software available on your Lambda and the name of the associated file server. The system calls this as part of the cold-booting procedure, so this is what you see just after booting a machine.

`load` *"pathname"*

Loads the file specified by the pathname. This works properly with both source and compiled (`qfasl`) files.

`compile` *function*

Compiles the specified function.

`describe` *symbol*

Tries to give you all the interesting information about the specified object. It does not give array contents. `describe` returns its argument so that you can do something else with it.

`who-calls` *symbol*

symbol can be a symbol or a list of symbols. This function tries to find all of the functions in the LISP world that call symbol as a function, or use symbol as a variable or constant.

The symbol `unbound-function` means something special to `who-calls`; if you use this as *symbol* then `(who-calls 'unbound-function)` searches the compiled code for undefined variables or functions. This is useful for finding errors such as misspelled or forgotten functions.

`grindef` *Function*

Pretty-prints (displays a nicely formatted version of) the definition of a function. This always works for interpreted code; it will work for compiled code, but only if

the interpreted version was in force at some point.

Packages

Every symbol in the LISP environment is (and needs to be) unique. If you define a symbol that has already been defined in the environment, the previous definition (or value) gets smashed and yours takes over. This could be very embarrasing if, for example, you redefined something like `setq`. To avoid the problem of different people wanting to use the same symbol names, ZETALISP-PLUS has a feature called the "package" system. This allows the existence of more than one obarray (this is where all the symbols live). You can then differentiate between different symbols with the same name by including the package prefix. For instance:

`pie:blueberry` is in package `pie` whereas
`jam:blueberry` is in package `jam`

You don't need to use the package prefix for anything that is in the current default package. Note that a colon (`:`) signals a package name to ZETALISP and separates it from the name of the symbol.

A newly booted Lambda always starts out with package user as the default readtable (or package). As a new user, you will probably need to use only package `'user` . However, if your Lambda is not freshly booted when you want to use it, someone else may have changed the default package. You can return to user by using the function `(pkg-goto)`. If you want to change to another package then use `(pkg-goto '`*packagename*`)`. When you begin writing large applications and think that using packages would be helpful, first read the chapter on packages in the *LISP Machine Manual*.

Getting a Record of Your Lambda Session

Sometimes it is useful to make a record of your programming session with the Lambda. In ZetaLISP-Plus you can create a "dribble" file which will record all of what you typed and LISP returned. The three functions that you need to know about are:

`dribble-start` *filename* &optional *editor-p*
: Opens filename as a "dribble" or "wallpaper" file, this records all terminal interaction; if *editor-p* is non-NIL then *filename* goes to a ZMACS buffer rather than a disk file.

`dribble-all` *filename* &optional *editor-p*
: Same as `dribble-start`, except this also records break loops, and debugger sessions etc.

`dribble-end`
: Closes files opened by the previous two functions.

For more detailed information on these functions see the *LISP Machine Manual.*

Function Information

Sometimes, you want to use a LISP function and don't remember exactly what arguments it takes, or quite what it does. Two LISP Listener commands can help with this. `CTRL-SHIFT-A` prints the function's arguments, `CTRL-SHIFT-D` prints the function's documentation string. To use either, first type a "(" (left parentheses) to open the expression, and then the function you want information on, follow this with either `CTRL-SHIFT-A` or `CTRL-SHIFT-D`.

PICTURE: of screen showing use of dribble function and info. commands.

6. Using ZMACS—The Editor

ZMACS is a full-screen editor, based on the EMACS editor developed at the MIT Artificial Intelligence Laboratory. ZMACS is written in ZetaLISP and can be used to edit text files as well as files containing LISP functions (and those in other programming languages, as well). This chapter provides a brief overview of ZMACS. For a more detailed introduction, see the *ZMACS Introductory Manual*; for full documentation of ZMACS, see the *ZMACS Reference Manual.*

Additional Information: Teach-ZMACS

In addition to the introductory and reference manuals, which provide written documentation of ZMACS, there is also an on-line tutorial that introduces ZMACS in an interactive, "hands on" manner. This tutorial can be accessed in two ways:

- In a LISP Listener, evaluate the form `(teach-zmacs)`.
- In ZMACS, execute the extended command `CTRL-META-X Teach ZMACS`. That is, simultaneously hold down the `CTRL`, `META` and `X` keys and then type `Teach ZMACS`.

6.1 ZMACS Background Concepts

Accessing ZMACS

The ZMACS editor can be invoked in three different ways:

- By typing (SYSTEM) E.
- By evaluating the form `(ed)` in a LISP Listener.
- By clicking [R] to cause the system menu to appear and then clicking [L|Select] and selecting [Editor]

A ZMACS window will then be selected (and created, if one does not already exist) and you will be in a ZMACS buffer, which will look like the following:

need to add screen dump of a ZMACS pane

The largest part of the buffer is devoted to the editing area proper. When you read in the contents of a file, they will be displayed in this area. If you create a new file, using an empty buffer, the text that you enter will appear in this area.

Below this, and separated by a line that runs the full width of the window, is an area consisting of the *Mode Line* and *Echo Area*. The Mode Line displays useful information about the selected buffer. This information includes:

- The mode that ZMACS is in. ZMACS contains information about many of the types of files that it can be used to edit and provides useful operations for each of these types. For example, there are different modes for various programming languages, such as Pll and Midas (PDP-10 assembly language), and, for each language, ZMACS has information on the conventions for formatting code in that language, the comment character, etc. Similar information exists for various text formatters, such as TeX and Bolio. There is also a Text Mode, for text files that are not designated to be formatted by any particular text formatter. In addition to *Major Modes*, such as LISP Mode or TeX Mode, there are also *Minor Modes* that control more specific aspects of the behavior of ZMACS. For example, the minor mode Auto Fill Mode automatically inserts new lines as you type, so that you do not need to explicitly type ⟨RETURN⟩. The default mode for a new file created in ZMACS, if no mode is specified, is LISP Mode.

  The mode that ZMACS should use to edit a particular file can be specified in either of two ways:

  - By explicitly setting the mode with a ZMACS extended command consisting of `META-X` followed by the name of the mode. For example, to get into TeX Mode, you would type `META-X TeX Mode`. You can use the command completion facility of ZMACS when typing mode changing commands; see page ??? below for a discussion of command completion.

  - By specifiying the mode of the file in an *attributes list* at the beginning of the file. Such attributes include the Major Mode of the file, the package the file belongs to, the base, if appropriate, etc. The file attributes list should be the first non-blank line in the file and the attributes list should be delimited by `-*-`. Also, in order to prevent the attributes list from being treated as code to be executed or text to be formatted, it should be preceded by a comment character. An example of an attributes list for a LISP file is:
    ```
    ;;;; -*- Mode: LISP; Package: GRAPHICS; Base: 10 -*-
    ```
    An example of an attributes list for a TeX file is:
    ```
    % -*- Mode: TeX -*-
    ```

- The name of the currently selected buffer. Normally the name of the buffer will be the same as that of an associated file and will appear in the following form:
  name.type#> ***directory***; *host*:
  host is the physical machine where the file is stored. ***directory*** is the directory on that host where the file is located. Subdirectories are allowed and are indicated by the use of a period (".") *name* is the first part of the filename proper of the file; this will usually be some descriptive name. *type* indicates the type of file this is, such as LISP for a LISP file or TEX for a TeX file. #> indicates that this is the most recent version of the file.

The syntax just described is that of the internal file system of LISP Machines. There are related syntaxes for other file systems. UNIX file names do not support version numbers. In addition, *directory* will consist of a series of names separated by slashes ("/"). Directories in UNIX are terminated with "/" rather than with ";". Though ZMACS displays filename information in the sequence given above, the user may type in filenames in the order:
host:directory;name.type

If no version number is specified and the file system being accessed supports version numbers, the highest version number will be assumed.

- The version number of the file, enclosed in parentheses.

- The position of the current window in the selected buffer. Typically, the entire contents of a buffer cannot be displayed in a single window, even one that extends the full height of the screen. At any given time, some of the contents of the buffer will be displayed and others will not. ZMACS employs three symbols to indicate what portion(s) of the buffer are not being displayed. ↑ indicates that the end of the buffer is being displayed and that there is more information earlier in the buffer; ↓ indicates that the beginning of the buffer is being displayed and that there is more information later in the buffer; ↑↓ indicates that an intermediate portion of the buffer is being displayed and that there is more information earlier and later in the buffer.

- The status of the selected buffer. If * appears in the mode line, the buffer has been modified and should be saved.

The Echo Area takes up the remaining lines of the window. ZMACS commands that you type are printed out or "echoed" in this area. If you complete a command quickly, it will not be echoed. However, if you hesitate over a command, it will be echoed here. This echoing facility is useful for novices but will not get in the way of experienced users. The Echo Area is also the portion of the window where `META-X` extended commands will appear; see page ??? below for discussion.

Inserting and Deleting Text and Positioning the Cursor

Many text editors have separate "modes" for inserting text, deleting text, and positioning the cursor. (These modes are different from the major and minor modes of ZMACS.) In ZMACS any of these operations can be done at any time (as well as more advanced operations not found in many text editors). In this sense, then, ZMACS is a "modeless" editor. An editor with modes uses them to distinguish the effects that typing a particular key can have. For example, a mode-oriented editor might assign the following meanings to the act of typing the "f" key:

in positioning mode:	move the cursor forward one position
in deletion mode:	delete the next character

in insert mode: insert the letter "f"

Control Character Commands

Since ZMACS does not use modes to accomplish this, there must be some other way for the user to signal ZMACS to insert text, move the cursor, delete some block of text, or perform some other operation. ZMACS accomplishes this with control characters. Normally, typing a key that transmits a printing character (a letter, number, or other character) causes that character to be inserted into the current text being edited. However, by typing a character key and a modifier key simultaneously, ZMACS is signalled to perform some other operation. The two modifier keys CTRL and META are the most frequently used modifier keys in ZMACS, though other modifier keys are also used. Most commands have a name that is mnemonic of the operation to be performed: for example, CTRL-F and META-F both move the cursor *F*orward. Typically, the CTRL version of a command performs an operation over some small domain, while the META version performs it over a larger domain. For example, CTRL-F moves the cursor forward one character, while META-F moves forward one word. The following table indicates the most commonly used CTRL and META cursor movement commands.

KEY and ACTION	CTRL version	META version
F - Move forward	character	word
B - Move backward	character	word
N - Move to next	line	line and add a comment
P - Move to previous	line	line and add a comment
A - Move to beginning	of line	of sentence
E - Move to end	of line	of sentence
] - Move to end	of page	of paragraph
[- Move to beginning	of page	of paragraph

If you wish to execute a ZMACS command more than once, you can prefix it with a numeric argument that specifies the number of times it is to be executed. Numeric arguments are entered by prefixing each number in the entire numeric argument by CTRL. For example, if you wish to move forward 20 lines instead of just one, you would type CTRL-2 CTRL-0 CTRL-N. This would move the cursor down twenty lines. Note that each part of the numeric argument has its own CTRL; typing CTRL-2 0 CTRL-N would cause two 0's to be inserted in the text before the cursor moved down a single line. CTRL-U is a predefined numeric prefix with a value of 4; typing CTRL-U before a command will cause that command to be executed 4 times. Successive CTRL-U's have a multiplicative effect: typing CTRL-U CTRL-U before a command will cause that command to be executed 16 times. CTRL-U can also be used as a prefix to turn numbers into numeric arguments. For example, CTRL-U 20 CTRL-N moves the cursor down 20 lines.

Extended Commands

In addition to single character commands, produced by simultaneously typing a printing character and one or more modifier keys, there are extended commands, produced by typing CTRL-X or META-X. CTRL-X commands are single character extended commands, that is, CTRL-X must be followed by a single character (either a plain character or a single letter control command). META-X commands are word extended commands, that is, META-X must be followed by one or more words. To make it easier to type META-X extended commands, there is a *command completion* facility, that will complete different portions of an extended command that has been specified uniquely. Typing ⟨SPACE⟩ will complete the current word of a multi-word command. Typing ⟨ALT MODE⟩ will complete the rest of the command (or, at least, as much as you have uniquely specified) but will not execute it. Typing ⟨RETURN⟩ will complete the current command and cause it to be executed. For example, if you wished to execute the META-X command Auto Fill Mode you would first type META-X. The following screen display would then appear:

Sample echo area of screen with Extended command: in left side and (Completion) in right

If you then type the first two letters of the command, Au, you will then have the following completion possibilities:

- typing ⟨SPACE⟩ will complete the rest of the word, causing Auto to be displayed in the echo region; you will then be able to type in the rest of the command;
- typing ⟨ALT MODE⟩ will complete the rest of the command, and will leave the line displayed in the echo area, allowing you time to cancel the command; and
- typing ⟨RETURN⟩ will complete the command and cause it to be executed.

At any time you can find out the possible completions of what you have typed by typing ⟨HELP⟩

probably insert screen dump of what the scroll window that comes up looks like.

Buffers and Files

Some of the most commonly used extended commands are for manipulating files and buffers. When you edit a file, you do not physically change the file that is on disk. Rather, the file is copied to a region, called a *buffer*, and the copy can then be altered. If you decide to save the alterations you have made, you can then write the contents of the buffer to disk or, if you do not, you can discard them. In ZMACS, individual buffers are separate windows, which have the main ZMACS window as their superior.

Saving and Reading in Files

The normal way to read a file into a buffer in ZMACS is by means of the command CTRL-X CTRL-F. You will then be prompted for the name of a file (which you should terminate with ⟨RETURN⟩). This reads a file with the specified name into a new buffer and gives that buffer the name of the file. If no file with the specified name already exists, ZMACS creates an empty buffer with that name.

If you decide to save the changes you have made to the buffer, this can be done with the command CTRL-X CTRL-S. This will save the buffer under the original name of the file that was read into the buffer, but with a new version number. CTRL-X CTRL-W is similar; it will save the contents of the buffer, but will prompt you for a new filename.

screen dumps with examples of the echo region with CTRL-X CTRL-S and CTRL-X CTRL-W

ZMACS keeps track of whether or not changes have been made to a buffer. If you try to save a buffer that has not been altered, ZMACS will respond (No changes need to be written.).

ZMACS can contain several buffers at the same time, each with an associated file. There are several ways of finding out what the current buffers are, moving between them, and performing other manipulations:

- CTRL-X CTRL-B will list the current buffers, with their associated files, and will indicate which of them have been modified and need to be saved. An alternative way of getting a list of the current buffers is as follows. [R] in ZMACS will cause a pop up menu to appear. If you mouse on [List Buffers], a temporary window will pop up, listing all the current buffers.

  screen dump of this

- CTRL-X B allows you to select one of the current buffers. You will be prompted for a buffer name, which should be terminated with ⟨RETURN⟩. An alternative way to select a different buffer is to select [List Buffers] from the ZMACS main menu. You can then select a buffer from the list of buffers. Each buffer name is mouse sensitive. If you [L] on any one of these, it will be selected as the current buffer.

- CTRL-X K allows you to kill a buffer. You will be prompted for the name of a buffer to kill, which should be terminated with ⟨RETURN⟩. (If you wish to kill the currently selected buffer, simply type ⟨RETURN⟩. You will then be prompted for the name of a buffer to select.) An alternative way of killing several buffers at the same time is to select [Kill Buffers] from the ZMACS main menu. You can then select the buffers you wish to kill by clicking the box in the row labelled Kill opposite each buffer you wish to kill. Mousing on Do It will cause each buffer to be saved or killed,

according to the specifications you have made. Mousing on Abort will escape from that menu, leaving all the buffers as they are.

Screen dump of Kill Buffers menu

- META-X Save All Files will go through all the modified buffers and ask you if you wish to save the associated files.

The Undo and CTRL-G safety net

There are many commands in ZMACS and it is possible that at some time you may accidentally make an unwanted change to the buffer or start to type in a command and decide that you don't really want to execute it. Undo and CTRL-G are commands that provide the ability to undo an accidental change or escape from a current command. Undo, which can be invoked by META-X Undo or ⟨HELP⟩ U, will undo the last major change that you have made (except for a "kill", which is undoable in another way; see the section on Killing and Deleting below). Before doing so, you will be given a prompt naming the change to be undone and asking you to confirm that you really wish to undo it.

screen dump of echo area with prompt, etc.

Undo itself can be undone, if you accidentally undo a change you didn't want to undo or if you decide to keep the results of the last change.

CTRL-G is used to abort an extended command. If you are in the middle of typing a CTRL-X command and you decide you really don't wish to complete it, you can use CTRL-G to escape from that command. CTRL-G must be typed twice to escape from a META-X command if you have started to type the extended command. The first CTRL-G aborts the string that you have begun to type; the second escapes from META-X itself.

6.2 Killing and Deleting

In ZMACS there are two ways of removing text from the currently selected buffer: one way is deleting, the other is killing. Text that is deleted is removed from the buffer and is "gone"; no record is kept of text that is deleted. Text that is killed is placed on a special list called the Kill Ring from which it may later be retrieved. This is a useful feature for moving text from one part of a buffer to another, from one buffer to another, or for making multiple copies of a piece of text. There are only two deleting commands in ZMACS: ⟨RUB OUT⟩, which deletes a previous character, and CTRL-D, which deletes the next character. However, if a numeric argument is used with either of these commands, so that more than a single character is removed from the current buffer, ⟨RUB OUT⟩ and CTRL-D kill rather than delete text. Common kill commands are:

META-D	Kills the next word

META-⟨RUB OUT⟩	Kills the previous word
CTRL-K	Kills the rest of the current line
META-K	Kills to the end of the current sentence
CTRL-X ⟨RUB OUT⟩	Kills to the beginning of the current sentence

The Kill Ring

The Kill Ring is a list of pieces of text that have been killed. It is possible to retrieve or "yank" back text that is on the Kill Ring by means of the commands CTRL-Y and META-Y. CTRL-Y retrieves the item that is at the top of the Kill Ring, i.e. the most recently killed piece of text and inserts it in the buffer at the current cursor location. META-Y is similar, except that it retrieves and inserts the next-to-last piece of text on the Kill Ring and makes it the current item on the Kill Ring. META-Y in ZMACS can only be used after CTRL-Y (or META-Y). This is the same behavior as META-Y in the PDP-10 implementation of EMACS but it is different from the behavior of META-Y in STEVE, which is not so restricted.

The default number of entries that the Kill Ring can contain is 8. This number can be altered by changing the value of the variable Kill Ring Max. This is done by means of the META-X command Set Variable. After typing META-X Set Variable you will be prompted for a variable name. Type Kill Ring Max ⟨RETURN⟩. You will then be prompted for a numerical value. Type this, followed by ⟨RETURN⟩. The maximum number of entries that the Kill Ring can contain will now be set to the number entered.

The Kill Ring in ZMACS is shared with the LISP Listener. This allows individual ZetaLISP functions to be defined in a ZMACS buffer, placed on the Kill Ring, and yanked into a LISP Listener to be tested.

Regions

It is, of course, possible to kill large portions of text through the use of numeric arguments and commands that kill longer stretches of text, such as CTRL-K and META-K. However, ZMACS makes this process easier by allowing you to define *regions* or large blocks of text that are to be placed on the Kill Ring. Moreover, it is possible to place the text in a region on the Kill Ring without removing it from its present location. This is a convenient facility for those times when you wish to copy a block of text to another part of the current buffer or to another buffer or LISP Listener. You can indicate the text to be moved and then move it to the desired new location(s) without having to kill it and yank it back to its original location.

How to Define a Region

Regions are marked, or *defined*, by indicating the beginning of the text to go into the

region and then indicating the end of the text. This can be done with keyboard commands or with the mouse or with a combination of the two. The keyboard commands CTRL-⟨SPACE⟩ and CTRL-@ indicate the beginning of a region. After marking the beginning of a region with either of these commands, you must then indicate the end of the region by moving the cursor to the desired location. This can be done with keyboard cursor movement commands. An alternative method to move the cursor over a long distance quickly is to move the mouse cursor to the desired spot and then [L]. This will move the cursor to the position indicated by the mouse cursor. A method of defining a region by using only the mouse is to press the left mouse button at the beginning of the region and keep it depressed. Move the mouse cursor to the end of the region and then release the left mouse button. As a region is defined, either by keyboard or mouse commands, ZMACS indicates the region by underlining the text contained in it.

Probably it would be good to have an example of a screen with a region underlined

Commands Using Regions

Once a region has been defined, there are two commands for placing it on the Kill Ring:

- CTRL-W kills the region and places it on the Kill Ring.
- META-W places the region on the Kill Ring without removing it from its current location.

6.3 Commands Especially Useful with LISP

The ability to define regions is useful for moving text around in a single buffer, between buffers, and from a buffer to a LISP Listener. There are other ZMACS commands that are especially useful for defining and testing ZetaLISP functions.

LISP Mode Cursor Movement and Killing Commands

Just as there are commands for moving through and killing various portions of text in Text Mode, there are commands for moving through and killing different portions of LISP code in LISP mode. Among these commands are:

- CTRL-META-F moves the cursor forward an s-expression.
- CTRL-META-B moves the cursor backward an s-expression.

- CTRL-META-A and CTRL-META-[move the cursor to the beginning of the current definitional form; this includes not only DEFUN, but also DEFFLAVOR, DEFVAR, etc.

- CTRL-META-E and CTRL-META-] move the cursor to the beginning of the current definitional form.

- CTRL-META-K kills the next s-expression.

- CTRL-META-⟨RUB OUT⟩ kills the preceding s-expression.

Formatting and Compiling LISP Forms in ZMACS

In LISP mode, ZMACS will aid you in formatting your LISP code according to standard formatting conventions. An especially useful facility is the treatment of the ⟨LINE⟩ key. In LISP mode, this will not only start a new line, it will also add the appropriate indentation for the current line of LISP code. The command CTRL-META-Q will properly indent a LISP form that is not properly formatted.

ZMACS also has commands for compiling LISP forms. CTRL-HYPER-C (also CTRL-ε = CTRL-GREEK-E) compiles the current region or, if there is no defined region, the current definitional form. META-X Compile Buffer compiles the entire buffer. A buffer or region which has been so compiled will be read into the LISP Listener, so that you can use its contents without otherwise explicitly loading it.

ZMACS Help Functions for LISP Code

ZMACS has various commands that are designed to make writing LISP code easier by providing on-line, immediate mechanisms for documenting the LISP functions that you are working with. Of these, META-. (read as "meta-point", or "meta-period", or "meta-dot", or even "meta-doc") is perhaps the most useful. After you type META-., ZMACS will prompt you for the name of a function, which should be terminated with ⟨RETURN⟩. (ZMACS will present a default function, which is the function to the left of the cursor. If you wish to see this function, simply type ⟨RETURN⟩.) ZMACS will then find the file that contains the definition of this function, read this file into a new buffer, make this buffer the selected buffer, and position the cursor on the specified definition. This is extremely useful for cases where you are editing a function and you find that it calls a function that you do not recognize. You can then use META-. to find the definition of this function and, possibly, the definition(s) of functions that that function calls, and so on.

Other useful commands include CTRL-HYPER-A, which prints the argument list of the function preceding the cursor, and CTRL-HYPER-D, which types out brief documentation about the function preceding the cursor. (Both these commands print out in the Echo

Area.) HYPER-META-D prints out the complete documentation of a ZetaLISP function. HYPER-META-D will prompt for the name of the function to be documented (the default is the function to the left of the cursor). Terminate your choice with ⟨RETURN⟩. ZMACS will then display the documentation in a temporary window.

probably some screen examples of this.

6.4 The ZMACS System Menu

need screen picture

Explain options.

In text above, menu alternatives to frequently used commands are given; do we need separate section on menu proper???

6.5 File Manipulation

Dired

The **Dired** command (for *DIR*ectory *ED*it) is used to examine and modify a directory. **Dired** can be used to examine a directory on either the local machine where you are working or on a remote host accessed over a network. **Dired** can be invoked by the command META-X **Dired**. ZMACS will then prompt for a directory name and will indicate a default choice. The syntax for directory names is like that for file names given above on page ???:
host:*directory*;*.*#>

Again, *directory* can contain "." to indicate subdirectories. In the case of directories on UNIX file systems, the directory name should be terminated with "/" rather than ";" and only "*" should follow the "/" that terminates the directory name. Here is an example of a valid directory name on a LISP Machine file system:
LAM3:INGRIA;*.*#>
and a valid directory name on a UNIX file system:
CAP:/lmi/mrc/*

Once you have specified a directory name, it should be terminated with ⟨RETURN⟩. (If you wish to examine the default directory that ZMACS presents, simply press ⟨RETURN⟩.) ZMACS will then create a special **Dired** buffer. The top line of this buffer will contain the name of the host and the directory as well as any other information that the host file system normally presents. Below this will be an alphabetically ordered list of all the files

in that directory. In addition, each file will be followed by the following information:

- The version number of the file (on file systems that support version numbers).
- The size of the file in blocks, pages, or some other (file system dependent) measure.
- The length of the file in bytes followed by the size of each byte (in bits), enclosed in parentheses.
- Indicators of various flags that have been set in the file specification:
  - "!" if the file has not been backed up on tape.
  - "$" if the file has been marked as protected from being "reaped" or "migrated", i.e. moved off disk to tape if it has not been accessed recently.
  - "@" if the file is protected from deletion.
- The creation date and creation time of the file.
- The date the file was last read, enclosed in parentheses. (This is file system dependent.)
- The username of the creator of the file.

Dired allows you to perform various operations on the files in the specified directory. ZMACS will display a full list of them if you type ⟨HELP⟩ M. **Dired** commands are single characters and affect the file on the line where the cursor is currently positioned. To move the cursor forward one line in a **Dired** buffer, type ⟨SPACE⟩ or CTRL-N; to move backward, type CTRL-P. Some of the more useful commands are:

- D to mark the file for deletion. Actual deletion takes place when you exit **Dired**. After you have typed D to mark the file for deletion, a D will appear in the left margin on the same line as the file. In addition, K, CTRL-D, and CTRL-K also mark a file for deletion.
- U to undelete the file.
- P to print the file on the default local hardcopy device.
- C to copy the file. ZMACS will prompt you to enter a file name, which should be terminated with ⟨RETURN⟩.
- R to rename the file. ZMACS will prompt you to enter a file name, which should be

terminated with ⟨RETURN⟩.

- E to edit the file. ZMACS will create a new buffer and read the file into it.

- V to view the contents of the file. This is useful is you want to see what is in a file without changing the contents.

You can exit from Dired by typing Q. If you have marked any files for deletion, ZMACS will list them in a temporary window and present you with the options: Delete? (Q, E, Y, or N) Y deletes the files but does not expunge them; E deletes and expunges the files. (Some file systems, such as that on Tops-10 and Tops-20 systems, as well as that on the LISP Machine itself, support two stages of deletion: deleted files can be recovered easily and are not physically removed from the disk, while expunged files are physically removed and cannot easily be restored.) N returns you to Dired. Q aborts from Dired.

screen picture of temporary prompt window?

6.6 Help

In addition to the help options available in particular sub-programs within ZMACS, such as the help facilities available in Dired, the facilities for command completion, etc., global help facilities are available. Of these, the two most useful are Apropos and Variable Apropos. Apropos, which is invoked by META-X Apropos or ⟨HELP⟩ A, will prompt you for a sub-string, which should be terminated with ⟨RETURN⟩. A temporary window will then appear, listing all the ZMACS commands that contain that substring. Each command is listed with the key binding that invokes it as well as a brief description of what it does. Apropos is useful if you cannot remember the exact name of a command, but you can remember some portion of its name. It is also useful if you wish to find out about the ZMACS commands in some particular area. For example, ⟨HELP⟩ A buffer will list all the buffer manipulation commands. Apropos, then, provides a convenient on-line mechanism for learning more about ZMACS.

Screen dump of Apropos window; what's a good one? one with MORE processing?

Variable Apropos is similar to Apropos. It is invoked by META-X Variable Apropos or ⟨HELP⟩ V. ZMACS will then prompt for a sub-string, which should be terminated with ⟨RETURN⟩. A temporary window will then appear, listing all the ZMACS variables that contain that substring, along with the current binding of each. Such variables control various aspects of the behavior of ZMACS. For example, we have already seen the variable Kill Ring Max, which specifies the number of items that the Kill Ring can contain. Other ZMACS variables include: Fill Column, which specifies the position on the screen where ZMACS will cause text to wrap to the next line; and Default Major Mode, which specifies the default mode that ZMACS starts up in, if no other mode is specified. ⟨HELP⟩ V

followed by ⟨RETURN⟩ will list all the ZMACS variables and their bindings.

Other useful help commands are:

- ⟨HELP⟩ C, which prompts for a ZMACS modifier key command, such as META-F or CTRL-HYPER-C, and prints out the name and description of the ZMACS command invoked by that key.

- ⟨HELP⟩ D, which prompts for the name of a ZMACS command, such as Auto Fill Mode or Compile Buffer, and prints out a description of that command.

- ⟨HELP⟩ W, which prompts for the name of a ZMACS command, such as Compile Region or Forward Word, and returns the modifier key command, if any, that invokes that command.

some example screens of this???

7. The Window System

The Window System is the heart of the LMI Lambda; if you understand how it works and how to use it, you can probably make the Lambda do anything you want it to. The Window System can be thought of as the Lambda operating system. Like a more typical operating system, the Window System organizes and controls your access to all the other systems or programs on the computer; unlike such a system, you can never really say that you are on "operating system level".

The main interfaces between you and the Window System are the mouse and the pop-up menus, but there are many system functions that you can access by an alternative method: usually by using either a ⟨SYSTEM⟩ key, a LISP function, or an extended editor command. For instance you can reach all the main systems through the ⟨SYSTEM⟩ keys:

⟨SYSTEM⟩ E accesses the ZMACS Editor.

⟨SYSTEM⟩ I accesses the Inspector.

⟨SYSTEM⟩ L accesses the LISP Listener.

⟨SYSTEM⟩ M accesses ZMAIL.

⟨SYSTEM⟩ P accesses the PEEK system.

When you choose a function using the ⟨SYSTEM⟩ key, the Window System will either call up or create a window containing the system you specified. If you want to be sure that you get a new system, type the appropriate letter while pressing the CTRL key. For example, to get a new LISP Listener, type ⟨SYSTEM⟩ CTRL-L.

The mouse menus are the easiest method to use as you learn the system; use the ⟨SYSTEM⟩ keys whenever possible when you get more familiar with the Lambda, as they are the quickest method.

There are two reasons why there are so many ways to accomplish a particular goal. The first is a concern with programmer style: different people like to do things in different ways, and the Lambda is designed to be unobtrusive and allow you to pick the method most natural for you. The second is a concern with the robustness of the system: since all important functions are accessible through both the mouse and the keyboard, you could still reach them in the unlikely event that either of these became non-functional.

Of Mice and Menus

Pop-up menus are central to the design of the Window System. They provide you

with a wide choice of options, without requiring that you memorize keywords or long command sequences. To call up a menu, click the appropriate button (usually the right button) on the mouse. The mouse documentation line will tell you when menus are available from a particular system, and which button to click to access them.

When you press a mouse button to get a menu, the menu will form on the screen around the area where the mouse cursor was. Notice that the mouse cursor now looks like a small "×". Also notice that the items on the menu are "mouse sensitive". This means that when the mouse cursor gets near them, the systen draws a small box around the item. You can now select an item by "boxing" it and then clicking left on the mouse. Some items take effect immediately; others lead to secondary menus.

PICTURE: of do it type menu showing X

If you call up a menu and then decide you don't want it you can usually leave just by moving the mouse cursor outside of the menu boundaries. Some menus have little boxes on the bottom that say "Do It" and "Abort"; in that case move the mouse cursor to the Abort box, and click left. (Of course if you want to do the action specified by that menu, then click on "Do It".)

Mouse Corners

Mouse corners are the Window System's way of letting you define part of the screen as a window size for other Window System commands to act on. Most Window System commands assume that you want to effect the entire screen, some give you a choice. For instance, the create and split screen options on the main menu both use mouse corners.

When you use mouse corners, the mouse cursor will first turn into something that looks like this: ⌊ an upper left hand corner, move this to where you want it and click [L]. Then, it will look like this: ⌋ a lower right hand corner. Move this to where you want it and click [L].

The System Menu

To get the Main System Menu, click right on the mouse. (In ZMACS, and several other situations, you need to click [R2]; the mouse documentation line will always alert you in these cases.) **Figure ******** shows this menu.

Picture: From a screen dump of the main menu blown up to about 2x life-size.

The main menu contains three columns; these are, from right to left: `Programs`, `This window`, `Windows`. These commands are explained below. If there is an alternate,

prefered way to execute a command this is included next to the explanation.

Programs

LISP ⟨SYSTEM⟩ L	Puts you into a LISP Listener.
EDIT ⟨SYSTEM⟩ E	Puts you into an editor buffer.
INSPECT	Puts you into the Inspector, looking at the specified object.
TRACE	Asks for a function name and then offers different tracing options. (Be careful about package names; sometimes even functions in the user package must be preceeded with a : (colon).)
EMERGENCY BREAK ⟨TERMINAL⟩ ⟨CALL⟩	Puts you into the "Cold Load Stream"; this is a LISP Listener that does not use the Window System for i/o. Sometimes this is necessary if the window software is ailing. Do not use this unless you really need to; it puts the machine into a very fragile state, and if you stay there for too long all your network connections break down.
HARDCOPY	Presents you with a menu to specify what you want printed and how you want it to look.

PICTURE: of hardcopy menu

This Window

KILL (from Peek)	Kills both the current window and the process running in it. If you choose this option, it presents you with a small "confirmation menu". This asks if you really meant what you requested. Indicate yes by clicking left in the window, indicate no by moving the mouse cursor away. Unless you confirm your choice, the system will not execute it.
REFRESH (from Peek)	This refreshes the display in the currently active window. Use when something weird has happened to the display.
BURY (from Peek)	Buries the current window; your window will disappear, and you will see whatever was "below" it.
ATTRIBUTES	Allows you to change the appearance of the current window, by using the following menu.

PICTURE: of Attributes menu

	Some items, such as "Label" can be filled in. Click left to remove the old entry, then type and finish your entry with a RETURN. Other items are multiple choice; the boldface item is the choice currently in force.
RESET (from Peek)	Starts the process associated with the current

window from scratch. Don't do this if you have valuable, unsaved material. However, if the process is wedged, this may be the quickest way to unstick it and start over.

ARREST (from Peek) — Halts the process associated with the current window.

UN-ARREST (from Peek) — Starts the process associated with the current window back up from where it left off. This will always work if the process was arrested from the system menu; if the process stopped for a different reason this may not restart it.

Windows

CREATE — Brings up a sub-menu of types of windows that you can create, then asks you to define the window shape using mouse corners.

SELECT — Brings up a sub-menu of all the available windows so that you can select one.

SPLIT SCREEN — Presents you with a menu that allows you to define how you will split your screen, and with what components.

PICTURE: of split screen menu, example of using it boxed nearby.

The "existing window" option allows you to select from a secondary menu of all the available and already created windows. The "frame" option allows you to save this window arrangement by "framing" it and giving it a name.

LAYOUTS — Another way to accomplish the "frame" effect available from the split screen option. This brings up a menu with at least the two options "Save this" and "Just Lisp". By picking Save This you can save your current screen setup under a name that you choose. Next time you pick layouts, the name of your design will also be on the menu for you to choose.

EDIT SCREEN — Brings up a secondary menu of different ways that you can manipulate all the different windows. Many of these items are similar to options under "This window" on the main menu.

PICTURE: of edit screen menu

Note that you cannot leave this option just by moving the mouse cursor out of the menu; you must choose the "exit" option in order to leave.

SET MOUSE SCREEN	This has an effect only if you have more than one monitor attached to your Lambda. If so, this option allows you to move the mouse cursor to another screen. Usually clicking left produces the default option and clicking right gives you a menu with all the options. On a two-screen system with a medium resolution color monitor, clicking left cycles through all the menu options in turn and clicking right presents you with the menu so that you can choose.

Special Windows

Sometimes the Lambda creates windows that you did not explicitly request. This usually happens in one of two circumstances: when you ask the Lambda for help, or when the computer needs to give you information about a process running in an unselected or non visible window.

Typeout Windows

These are the kinds of windows that you see when you ask the system for help; they often say "press any character to flush" on the bottom. The Lambda creates a temporary window to serve this specific purpose and destroys it when the job has ended. If you press ⟨HELP⟩ and subsequently ask to select a window from the menu, "Help Window" will not be one of the choices.

Often you will see **more** at the bottom of such a window; this means that the system has more information to display than it can fit on one screen. To see "more" press the space bar, to leave type "n".

Notification

Sometimes the system needs to notify you of an error in an unselected process, or an attempt to type out by a background process that has no window to type out to. The console will beep and the message "Notifications pending TERMINAL N is one way to see them" will appear either on the screen in square brackets ([]), or on the mouse documentation line. To select the window referred to by the notification type ⟨TERMINAL⟩ OS; to return to your previous window type ⟨TERMINAL⟩ S.

Notifications get saved; to see old ones again evaluate `tv:print-notifications`.

The Window Stack

Another way to select windows (probably quicker once you get comfortable with the Lambda) is through the ⟨TERMINAL⟩ nS command.

All the currently selectable windows are arranged in a kind of stack (actually a ring) with the selected window on top. (Evaluating `(listarray tv:previously-selected-windows)` will give you the current stack order minus the top (currently selected) window.) When you select a window with the mouse (or some version of ⟨TERMINAL⟩ S), the Window System moves it to the top of the stack.

PICTURE: of window stack with TERMINAL commands that will get that particular window alongside.

⟨TERMINAL⟩ S gives you the window active before the current window and moves the ex-current window to be the second item on the stack.

⟨TERMINAL⟩ 1S gives you the previously active window and moves the old one to the bottom of the stack. Note that although both of these commands access the same window for your use, their stack behavior is different.

⟨TERMINAL⟩ -S takes the window on the bottom of the stack and moves it to the top.

⟨TERMINAL⟩ nS takes the currently selected window and puts it in the Nth place on the stack and exposes the window below it. If you do this repeatedly the effect is to cycle through the top *n* items on the stack. (⟨TERMINAL⟩ S is actually shorthand for ⟨TERMINAL⟩ 2S.)

⟨TERMINAL⟩ -nS takes the window on the Nth place on the stack and puts it on top. Used repeatedly, this cycles through the top *n* items on the stack in reverse order from the previous command.

Note that while both using the select option from the main menu and using the ⟨TERMINAL⟩ S command update the stack variable, something like cycling through the available LISP Listeners using ⟨SYSTEM⟩ L repeatedly will not. So, if you have been doing that and then access a window using a variant on ⟨TERMINAL⟩ S, you may not get the window you expect.

Peek

Peek is a program that allows you to look at the status of various system functions. You can also manipulate both processes and windows from within Peek. It is a good way to get rid of extraneous windows and their associated processes. Also, when a process appears to be stopped for no apparent reason, you can often find out why or reset it from Peek.

To get a Peek Window type (SYSTEM) P; the first time you request a Peek Window you will see something like this:

PICTURE: from a screen print of the beginning Peek setup

Subsequently, you will return to Peek exactly where you left it. (Of course, you can get a brand new Peek window by typing (SYSTEM) CTRL-P, but it is usually pointless to have more than one.)

Q(uit) leaves Peek and returns you to the previous process.

Many of the Peek sub-systems update their displays frequently, because they give information on ever changing processes. nZ allows you to change the interval at which these updates display. n is the number of 60ths of one second. The default update period is five seconds.

The remaining choices are accesible both by letter command and by selecting them from the mouse menu at the top of the screen. The two of most general interest are Active Processes and Window Hierarchy, so those will be discussed in greater detail.

Active Processes

The "Active Processes" display allows you to see all the different processes running on the Lambda, and their current state. A typical display looks something like this:

PICTURE: screen dump of peek/ap window

Notice that the items on the display are mouse sensitive; you can "box" a process and then click and get a menu of operations you can carry out on that process.

PICTURE: small screen dump of menu

Many of these choices are operations that you can access from the main menu, but here you can affect any process, not just the current one. Often, Peek is a way to figure out why a process is hung, and to get it to start again. You can also use the debugger or the Inspector (for more info see Chapter 6) to analyze in more detail what the problem with the process is. If you just want to get it going again, and don't care about the recent state, you can reset, or Kill and Un-arrest the process.

Window Hierarchy

This display allows you to see all the active windows and how they relate to one

another. For instance, if you have used layouts or split screen to set up a "frame" with several windows in it, they will be displayed as subordinate to the frame.

PICTURE: of peek/wh display w/options window

Again, the items in this display are mouse sensitive. You can click on the mouse and get the small menu displayed in figure ****; these options are self explanatory.

Mouse Scrolling

Peek and some of the other systems on the Lambda sometimes have more data to display than will fit on the screen. In such cases you can use the mouse scrolling feature to see the rest of the display. There are two ways to tell if this feature is active:

- Little italic legends on the screen say "more above" or "more below".
- The mouse cursor becomes a heavy black arrow when brought near any of the edges of the screen.

When the mouse cursor is near the top or the bottom of the window (and it looks like a heavy up or down arrow) you can scroll the window by moving the mouse slowly in the appropriate direction. To move a whole screenful at a time click on the "more above" or "more below" legends. If you move the mouse cursor to one of the sides of the screen you will see a fat double-headed arrow. This means that using the mouse buttons will scroll the display in various ways:

L	Move the line next to the mouse to the top of the window.
L2	Move the line next to the mouse to the bottom of the window.
R	Move the top line to where the mouse is.
R2	Move the bottom line to where the mouse is.
M	Jump to a place in the window contents as far, proportionally, from its beginning as the mouse cursor is from the top of the window.

Note that when the mouse cursor has assumed its mouse scrolling form you cannot reach the system menu.

8. Debugging: When and How

When the Lambda system detects an error condition it automatically activates the error handler, also known as the debugger. Sometimes, if the error is trivial, you will be allowed to choose whether or not to enter the debugger and your screen will look something like this:

PICTURE: of trivial error that throws to debugger option with a useful caption and circles and arrows.

More significant errors can be of two types:

- Syntax or language related errors
- Logic or intent errors

The Lambda programming environment provides several tools for analyzing your programs to locate their problem spots. Foremost among these is the error handler. Others, such as Trace, Step, and the Inspector can be used on a volunteer basis; ZetaLISP will never forcibly place your program in one of these.

The Debugger

The debugger can be used both to locate problems with your program, and to fix those problems and proceed with execution. When inside the debugger you can type either a LISP form to be evaluated, or a debugger command. (Debugger commands are explained below.) If you don't really want to be in the debugger you can exit by pressing ⟨ABORT⟩. This brings you back to the LISP Listener and aborts execution of the offending LISP form. (Note that if you type an incorrect LISP form while in the debugger this will cause you to enter a second, deeper level of the debugger. In this case, ⟨ABORT⟩ just brings you up to the previous level.)

When your program enters the debugger you will see:

PICTURE: with little arrow prompt and SUPER options

The → (right arrow) prompt indicates that the debugger is waiting for input from you. The number of arrows indicates how many levels deep you are in the debugger.

Many error handler commands can take numeric arguments. If it makes conceptual sense for a command to take such an argument it probably does. To give a numeric argument to a command, type CTRL-*NUMBER* followed by the debugger command. For instance,

```
CTRL-5 CTRL-N
```

moves down five stack frames.

Debugger Commands

Here are some of the most useful debugger commands. They are presented by type rather than alphabetically so that it is clear what they do and when they are likely to be useful. (Chapter 27 of the LISP Machine Manual contains a more complete explanation and reference.)

Stack Frames

Often, an error happens much earlier in a program than when its effects become apparent and LISP throws your function into the error handler. So, it is useful to be able to move "up" and "down" your program to both before and after the point where it enterred the debugger. You do this by giving the debugger commands that change the current stack frame. You can consider a stack frame to be a snapshot of the status of a program at any single point in its execution.

PICTURE: of simple stack frame example

The LISP Machine distinguishes between two types of stack frames, "interesting" ones and "uninteresting" ones. Interesting frames result from what you would normally think of as program code. Uninteresting frames involve LISP internal functions such as `*eval` and `prog`. Since it is unlikely that you want to "debug" `eval`, there are stack frame commands that automatically skip uninteresting frames.

The stack frame commands produce output in a pseudo-machine language format. This is the "disassembled" LISP code for which there is no actual assembler. It is fairly straightforward, and you should be able to get information on function calls and variable values just by looking at it. See the *LISP Machine Manual* if you want to understand the LISP Machine Assembler in greater depth.

The following are the relevant commands for moving among stack frames:

`META-L`	Displays a full-screen typeout of the current frame.
`CTRL-N` or ⟨LINE⟩	Moves down to next interesting frame.
`META-N`	Moves down to next interesting frame and provides a full-screen frame display.
`CTRL-META-N`	Moves down to next frame even if it is uninteresting.
`CTRL-P` or ⟨RETURN⟩	Moves up to the previous interesting frame.
`META-P`	Moves up to the previous interesting frame and provides a full-screen frame display.

CTRL-META-P	Moves up to the next frame even if it is uninteresting.
CTRL-S	Searches for a frame containg a specified function.
CTRL-META-U	Moves up the stack to the previous interesting frame.
META-<	Goes to the top of the stack.
META->	Goes to the bottom of the stack.

Backtracing

A backtrace is a concise history of where a function has been. The error handler has several commands that allow you to access this information.

CTRL-B	Prints a brief backtace.
META-B	Prints a longer backtrace.
CTRL-META-B	Prints a longer backtrace; does not censor interpreter functions.

Proceed Commands

The following commands allow you to change variable bindings and function definitions so that when you try to restart the function it may work.

CTRL-C or ⟨RESUME⟩	Attempts to continue the function, using the first proceed type available.
META-C	Attempts to continue after first "permanently" fixing the error. (This command is not available for all errors.)
CTRL-D	Attempts to continue, like CTRL-C, but traps on next function call.
META-D	Toggles the "trap on next function call" flag.
CTRL-E	Places you in the editor with source code of the current frame. Return to error handler with ⟨TERMINAL⟩ S.
CTRL-META-F	Sets * to the function in the current frame.
CTRL-R	Returns a value from the current frame as if it had actually been computed.
CTRL-META-R	Reinvokes the function in the current frame. You use this if you think you have located and fixed the error.
CTRL-T	Throws a value to a tag.
CTRL-X	Toggles the flag in the current frame that causes a trap on exit or throw through that frame.
META-X	Sets the flag causing a trap on exit or throws through the frame for the current frame in addition to the frames outside it.
CTRL-META-X	Clears the flag for the current frame and all the frames outside of it.

super Commands: SUPER-A, SUPER-B etc. are assigned to the various proceed types available for the particular error. Since these are different each time, the debugger always prints them out for you. If you need to see them again type ⟨HELP⟩ X.

Miscellaneous Commands

(ABORT) In general, this command rescues you from completing your current action. For instance, if you are typing a LISP form to the debugger it aborts the form, and if you are at a particular level in the debugger it moves you up one level.

? or (HELP) Prints a really excellent help message.

Window Error Handler

You can enter the Window Error Handler only after you are already in the keyboard level debugger. CTRL-META-W is the debugger command that activates the Window error handler. The Window Error Handler gives a more graphically oriented view of what is happening with your program; the keyboard error handler is better for manipulating variables and moving up and down the stack. Whether you use one or the other, or both, is largely a matter of taste and programming style. The error handler window is divided into seven panes; the figure below shows the Window Error Handler display with the contents of each pane labeled.

PICTURE: of Window Error Handler display, with circles and arrows!

The Window Error Handler has a menu of commands, and in addition many other items are mouse sensitive. It is easy to use, and the mouse documentation line can tell you what your possibilities are.

Entering the Debugger Manually

Errors of logic will probably not cause your program to enter the debugger. Your program will run, it just won't do what you intended that it do. In cases like this you may find it helpful to use the debugger anyway to help locate errors. There are several ways to do this. The simplest is just to type to the LISP Listener something silly like pineapple-pie, which you know is unbound; this will throw you into the debugger with an unbound variable error. You can use META-(BREAK), which will force the program into the debugger, as long as it is reading from the keyboard; CTRL-META-(BREAK) will force the program into the debugger immediately. If your program takes off for never never land, this is the way to see what's going on. (BREAK) without modifiers simply enters a "break" loop.

Other Debugging Aids

In addition to using the debugger, you may find that the Trace and Step functions can provide information helpful in debugging your program.

Trace

The Trace facility allows you to request that LISP take certain special actions upon entering or leaving one or more specified functions. The default action is to print the function name and arguments when the function is called and to print the function name and values when the function is exited. Trace can be very helpful when you are trying to track a runaway recursion. Trace is a regular LISP function; the general appearance of the form is:

`(trace ((`*fun-name1 fun-name2 ...*`) `*option1 option2 ...*`))`

If you do not specify any options the function will be traced in the default manner. You don't need to list all the functions that you want traced at once. You can always add more and the system will keep track of all of them. `(trace` *fun-name1 fun-name2 ...*`)` is a simplified form that you can use if you don't want any of the options.

Here are some of the options. (As usual, the *LISP Machine Manual* provides complete documentation.)

`:break` *pred*
: If *pred* evaluates to non-NIL, then this causes LISP to enter a breakpoint after printing the trace entry information, but before executing the function. The example indicates how you would specify this option to trace.

```
(trace (stew :break (equal meat 'rotten)))
```

`:exitbreak` *pred*
: Similar to `:break` except that the breakpoint occurs after the function is executed and the trace exit information is printed.

`:step`
: Causes the function to enter single-step execution every time it is called. Step is discussed in the following section.

`:wherein` *function*
: Traces the function only when it is called by the function named after `:wherein`.

You can also access trace from the system menu, and through the ZMACS extended command `META-X Trace`. Both of these present you with a menu listing the options so that you don't have to memorize the syntax of the trace function.

To remove one or more functions from the list of those to be traced by the system, use the function Untrace. `(untrace` *fun-name*`)` stops a particular function from being traced, while `(untrace)` by itself stops all functions from being traced.

Step

Step allows you to single-step through the evaluation of any function. This is similar to the single-step facility found in many machine language debuggers. You cannot use Step with a compiled function; only interpreted code can be stepped. If your interpreted function calls a compiled function, then when you reach that point with the Stepper you will see entry and exit only; using the Stepper on compiled code won't hurt either the Stepper or the code, but it won't be very informative either.

To use the Stepper type:

`(step '(`*function arg1 arg2 ...*`))`

This is what stepping through a function looks like:

PICTURE: of some of a stepping session that includes all arrows etc.

Before any form is evaluated it is printed out preceeded by a →. (If it is a long form it may only be partially printed.) The stepper expands all macros; it signals macro expansions with a double arrow: ↔. A backwards arrow ← will preceed the form and values returned by a function. A ^ character will appear between values if the form returns multiple values. When the stepper has finished evaluating the arguments to a form and is about to apply the function it prints a "λ".

Recursively called functions indent proportionately to their recursion levels so you can follow the process graphically.

Stepper Commands

CTRL-N Steps to the next evaluation. The stepper continues until the next item to print out, then accepts another command.

⟨SPACE⟩ Goes to the next form to be evaluated at this level. All the evaluations at lower levels (more (parentheses)) occur, but are not single-stepped or printed out. This is a quick way to move through parts of the program that don't interest you.

CTRL-A Doesn't show argument evaluation, but pauses before applying the function.

CTRL-U This is like ⟨SPACE⟩ but more so; it skips over all evaluation on the current level as well as lower levels.

CTRL-G Grinds or "pretty-prints" the current form.

CTRL-T Retypes the current form in full, without any truncation.

CTRL-L Clears the screen and redisplays the last ten pending forms.

META-L Similar to CTRL-L, but doesn't clear the screen.

CTRL-META-L Redisplays ALL pending forms.

CTRL-E Switches to the editor.

CTRL-B Puts your function in a breakpoint—read-eval-print loop—from which you can examine and set variables and other aspects of the current environment. Several special variables are available:

step-form is bound to the current form.
step-values is bound to a list of returned values.
step-value is bound to the first returned value.

If you change the value of these variables, it will affect execution.

CTRL-X Exit, finishes the evaluation without any more single-stepping.

⟨HELP⟩ or ? Prints information on the Stepper commands.

The Inspector

The Inspector is a window-oriented version of **describe**. In other words, it lets you investigate the insides of data structures the way that the previous tools let you look inside the process of function execution. You can enter the Inspector through the main system menu, ⟨SYSTEM⟩ I, or by typing (**inspect** *argument*). When you ask to inspect a particular object its components are displayed. What the components are depends on the type of object; for example, the components of a list are its elements, and those of a symbol are its value binding , function definition, and property list. A typical Inspector window looks like this:

PICTURE: of inspector with panes labelled.

Using the Inspector is straightforward; the mouse documentation line tells you what you can do. However, there are several items that are not self-evident. Several keyboard commands are active when you are not in the middle of typing a LISP form.

CTRL-Z exits and deactivates the Inspector.
⟨BREAK⟩ runs a Break loop in the typout window.

(QUOTE) reads and evaluates a form, rather than inspecting it.

Note that if you change the value or structure of an object, the program will not know about it until you decache the object and redisplay it.

9. The UNIX System

The UNIX System is an alternate operating system available from LMI as a separately priced option. It was originally developed at Bell Laboratories in the early 1970s. UNIX is widely used in both commercial and academic sites and is recognized as a standard for high-quality, multi-user operating systems.

We supply System 5 UNIX which supports VI, the C programming language, and multiple users. You can have up to three users on UNIX at once, with upgrades available to eight or sixteen users.

Logging In and Out

UNIX will present you with a login prompt that looks like this:

```
:login:
```

Just respond with your login name. You may also have to provide a password; if so, UNIX will prompt for it.

Filenames and Directory Structure

UNIX filenames should be fourteen or fewer characters. UNIX is case sensitive and the usual convention is to use lowercase.

The UNIX file system is hierarchicaly organized with directories arranged in a tree structure. The top, or root directory is denoted by "/". A typical UNIX file system will look like this:

PICTURE: of a tree shaped directory

To specify a particular file or directory as the argument to a command you start at the root and work downward, separating directories with additional slashes (/). For example, if .....**EXAMPLE***************

When you login you are automatically placed into your "home" directory. This is where your files will be created and live by default. Unless you actively switch to another directory this will be your working directory and you will be able to refer to files in it by filename only; you don't need to give the whole pathname. If you do switch directories you can then refer to files in your new working directory by filename only.

Since UNIX, unlike the Lambda, is a protected system, there are probably directories

that you can't enter and/or read from. If you try, UNIX will give you an error message.

Some directory commands are:

`mkdir` *dirname*	creates a subdirectory
`cd` *pathname*	changes your working directory to *pathname*; if you don't specify a pathname this will bring you back to your home directory.
`pwd`	displays the pathname of your (current) working directory. This is excellent to use if you are exploring and get "lost".

Some System Commands

The following are just some basic system commands to allow you to get started using UNIX. UNIX has two wildcards * and ? that let you specify whole groups of files as the argument to a system command. They can save you alot of typing, and they can get you into alot of trouble quickly; use with care. ? substitutes for exactly one character in a filename. For instance,

```
rm ?at.doc
```

will remove files named `bat.doc`, `cat.doc`, and `mat.doc`. * substitutes for any number of letters. So,

```
rm *at.doc
```

will also delete `splat.doc` and `frat.doc`.

`cp` *file1 file2*	copies *file1* into a file named *file2*
`mv` *file1 file2*	if both *file1* and *file2* refer to the same directory, then *file1* gets renamed to *file2*. If the names refer to different directories, then *file1* is moved to the directory and name specified by *file2*.
`rm` *file(s)*	permanently deletes the named file(s).
`cat` *file(s)*	displays the contents of a file on the screen.
`more` *file(s)*	displays the contents of a file on the screen, but stops after each screenful to give you a chance to read the material. To continue and see "more" press the spacebar. This is usually more useful for reading text than `cat`.
`man` *topicname*	displays the relevant material from the UNIX reference manual. This is not very useful for learning UNIX—you must know pretty much what you are looking for before you can find it—but it is an excellent reference tool.

Extended-STREAMS

One of the most wonderful things about having UNIX on your Lambda is that you can make UNIX and LISP work for each other; you do this by using the Extended-STREAMS software included with the UNIX processor.

Extended-STREAMS allows you to switch easily from the ZetaLISP-Plus environment to the UNIX environment, and even use LISP from UNIX and call UNIX tools from LISP.

The LISP Processor and the mc68000 Processor running UNIX can run concurrently, sharing various system resources. Both processors may access the disk through a common interface. Memory gets divided among the host processors based on certain system configuration information.

There are two ways to use the Extended-STREAMS software: directly using the three dedicated stream links or through a virtual CHAOSNET connection.

Some of the material below will be easier to understand if you have had prior experience with both UNIX and CHAOSNET.

Direct Connection

The easiest way to use a dedicated link is by typing ⟨SYSTEM⟩ U to get a UNIX window. This behaves just like any other Lambda window (such as a ZMACS or LISP window) except that it contains a UNIX process. As such it does not interact with other processes. ???????

There are three direct stream links between the LMI Lambda and the UNIX system. From UNIX these are called:

```
/dev/tty10
/dev/tty11
/dev/tty12
```

These are character special devices that act just like terminal lines. By default, they are in raw, no echo mode, but you can modify this. The UNIX initializer only accepts logins on `/dev/tty10`, so this is the line that the routine called by ⟨SYSTEM⟩ U uses. `/dev/tty11` and `/dev/tty12` are not enabled in the UNIX file `/etc/ttys`, so the initializer doesn't set up login listeners for them. If you want to use them with login capability you must change the initial 0s (zeros) to 1s (ones) for the appropriate entries in `/etc/ttys`.

From LISP, the three relevant symbols are:

```
*unix-port-0*
*unix-port-1*
*unix-port-2*
```

These are bound to objects of the type `si:unix-stream`. You must send the `:setup` message to these streams before use and after a warm boot. I/O operations to these streams cause data to be sent to or received from the corresponding UNIX devices.

CHAOSNET Link

The CHAOSNET link provides a general communication mechanism allowing each host processor to behave like any other CHAOSNET host including the use of file transfer (CFTP) and remote terminal (SUPDUP) protocols. (Use of CHAOSNET on the LISP Machine is described in Chapter 23 of the *LISP Machine Manual*.) The following assumes a familiarity with the use of CHAOSNET with both UNIX systems and LISP Machines. **It's Chapter 26 of Orange** *Chinual*.

You can use the new STREAMS software even if your Lambda doesn't have an associated UNIX processor, to link with UNIX on another machine. Just specify the host address of a UNIX host on your local area network. It is immaterial that the associated UNIX host is not a part of the local system, except that the overhead required to use the physical network may degrade performance slightly.

Both the UNIX system and the LISP processor have CHAOSNET servers which respond to the contact name "EVAL". The UNIX system's EVAL server passes input from the connection to the UNIX shell and the standard input of any command and passes output back through the connection. The Lambda's EVAL server reads LISP forms from the connection, evaluates them and prints the results back. Using these servers, either system can take advantage of the resources and software of the other.

10. Editing in UNIX

UNIX comes with two editors: they are Ed, a line editor, and VI, a full screen editor. You can write programs or edit files using either one, but VI is easier to learn and use so it is the one that this Introduction discusses.

Unlike ZMACS, VI has modes. There is an insert mode and an editing mode. When you are in insert mode you can type to the editor just as if it were a typewriter. In edit mode you can move freely around the text and edit it. Because VI distinguishes between characters and commands by the mode it is in, most VI commands are simple single or double character keystrokes without any CTRL or other modifiers.

To use VI type:

`VI` *filename*

If this is a new file, VI will tell you so, otherwise it will get the appropriate file and put it in the buffer for you to edit.

To exit VI, from command mode type:

`:wq`

If you don't want to save your work you can exit by using `:q!`. To save and continue editing use `:w` *filename*. The *filename* argument is optional; if you don't give one VI will overwrite the file with the same name as your VI buffer with the contents of the buffer.

Inserting Text, Moving Around

To go into insert mode press `i`. To leave insert mode and go to command mode press (ESC). (Note: If you are using VI from the high-resolution monitor, rather than something like a Z29, use (ALT MODE) as your "escape" key.) When you use `i` to insert text, text is inserted to the left of the cursor. There are other insertion commands that begin inserting text differently. Some of the more common are:

`I`	inserts text at the beginning of the current line.
`a`	append, inserts text to the right of the cursor.
`o`	opens a new line for text below the current line.
`O`	opens a new line for text above the current line.

All commands that leave you in insert mode must be ended with (ESC) when you want to return to command mode.

VI offers a wealth of cursor commands; some are presented in the chart below:

In Line Movement

	⟨SPACE⟩	moves cursor one character to the right.
	l	moves cursor one character to the right.
	⟨BACK SPACE⟩	moves cursor one character to the left.
	h	moves cursor one character to the left.
*	w	moves cursor to the beginning of the next word.
	e	moves the cursor to the end of the current word.
*	b	moves cursor back a word.
*	0	moves cursor to the beginning of the current line.
*	$	moves cursor to the end of the current line.

Between Line Movement

	-	moves cursor to the beginning of the previous line.
	CTRL-P	moves cursor up one line, but in the same column.
	⟨RETURN⟩	moves cursor to the beginning of the next line.
	CTRL-N	moves cursor down one line, but in the same column.
*	H	moves cursor to the top line of the screen.
	M	moves cursor to the middle line of the screen.
*	L	moves cursor to the bottom line on the screen.

The commands with asterisks next to them can take numeric arguments. For instance, 3w will move you 3 lines ahead, 4L moves the cursor four lines from the bottom of the screen, etc.

There are also commands that enable you to move through your file more quickly by scrolling or paging.

CTRL-D	scrolls down through a file
CTRL-U	scrolls up through a file
CTRL-F	pages forward through a file
CTRL-B	pages back ward through a file

Scrolling gives more of a feeling of continuity than paging because it leaves a larger overlap between screens.

You can also use the search commands to move longer distances.

Deleting and Moving Text

VI offers several ways to change already written text. The simplest way to correct minor mistakes is to use the ⟨RUB OUT⟩ or ⟨DELETE⟩ key. This works even while you are in insert mode. Another way is to switch to writeover mode. The "R" command replaces text character for character until you go back to edit mode with ⟨ESC⟩.

Many of the cursor commands take editing prefixes, so that instead of MOVING that distance, they will EDIT on that amount of text.

d	deletes text
c	changes text and leaves you in INSERT MODE
y	yanks, or copies, text

For instance, **3dw** deletes the next three words; **cw** will change the current word and then leave you in insert mode to add more text if you want. It is equivalent to typing **dw** followed by **i**. The line commands are a slight exception to this general rule: **cc** changes a whole line and **dd** deletes it.

The **d** commands do not actually delete the text, they place it in a buffer from which you can "pull" it back. So, if you want to COPY text use "y", and if you want to MOVE text to a new location use "d".

P	pulls the text back and places it after the cursor
p	pulls the text back and places it before the cursor

Searching

VI's search facility allows you to look for a particular word or phrase anywhere in your text. This is a good way to move large distances or find phrases that you want to change without going through the text line by line. The only place that VI will not be able to find something is on the current line. This should not be a problem, because you will always be able to see that line on your screen.

/pattern ⟨ESC⟩ is VI's basic search command. VI will move the cursor to the beginning of the first occurance of the pattern you are searching for. It will search downward in your text from the current cursor position, loop to the top and continue searching until it comes back to your original place. If the pattern does not occur in your text the cursor will remain where it was, and VI will display a message on the bottom of your screen.

n	asks VI for the next occurance of the pattern downward in the text.
N	asks VI for the next occurance of the pattern upward in the text.

You can also use the search facility to define arbitrary regions of text. For instance, **d/***foofraw* ⟨ESC⟩ will delete everything from the current cursor position to the next occurance of *foofraw*.

Appendices

A: A Sample INIT File

B: A Guide to the ⟨TERMINAL⟩ and ⟨SYSTEM⟩ Keys

C: Further Reading

Here are some suggestions for more reading on topics covered in this manual. Most of these are manuals written or distributed by LMI; some are high quality books that are readily available.

Chapter One

All of the hardware manuals *Hardware I* through *Hardware III* are relevant to some degree. Also, *The Field Service Manual* in the *Basics* volume.

Chapter Two

The Field Service Manual provides further information on booting.

Chapter Three

The LISP Machine Manual is, of course the final authority on ZETALISP-PLUS. For a conceptually oriented introduction to LISP read *LISP: A Gentle Introduction to Symbolic Programming* by David S. Touretzky. Dr. Robert Ingria at LMI has prepared a pamphlet enumerating the differences between the dialect of LISP used in this book and ZETALISP. LMI will send this free upon request.

Chapter Four

The ZMACS Introductory Manual and the *ZMACS Reference Manual* both provide more detailed discussion of the capabilities of ZMACS as a programming tool.

Chapter Five

The Window System Manual discusses aspects of LISP and the LISP Machine that you need to know about in order to program using windows.

Chapter Six

The LISP Machine Manual and the forthcoming *Applications Programming on the LMI Lambda* both discuss debugging tools and techniques in greater depth.

D: Glossary

Index

PROGRAMMING ON THE LAMBDA

Programming on the Lambda, a hands-on tutorial guide to the Lambda programming environment, will be available to all LMI customers in first quarter 1983.