MANUAL REVISION HISTORY

Explorer System Software Design Notes (2243208-0001 *A)

Original Issue ..... June 1985

Revision A ..... June 1987

© 1987, Texas Instruments Incorporated. All Rights Reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Texas Instruments Incorporated.

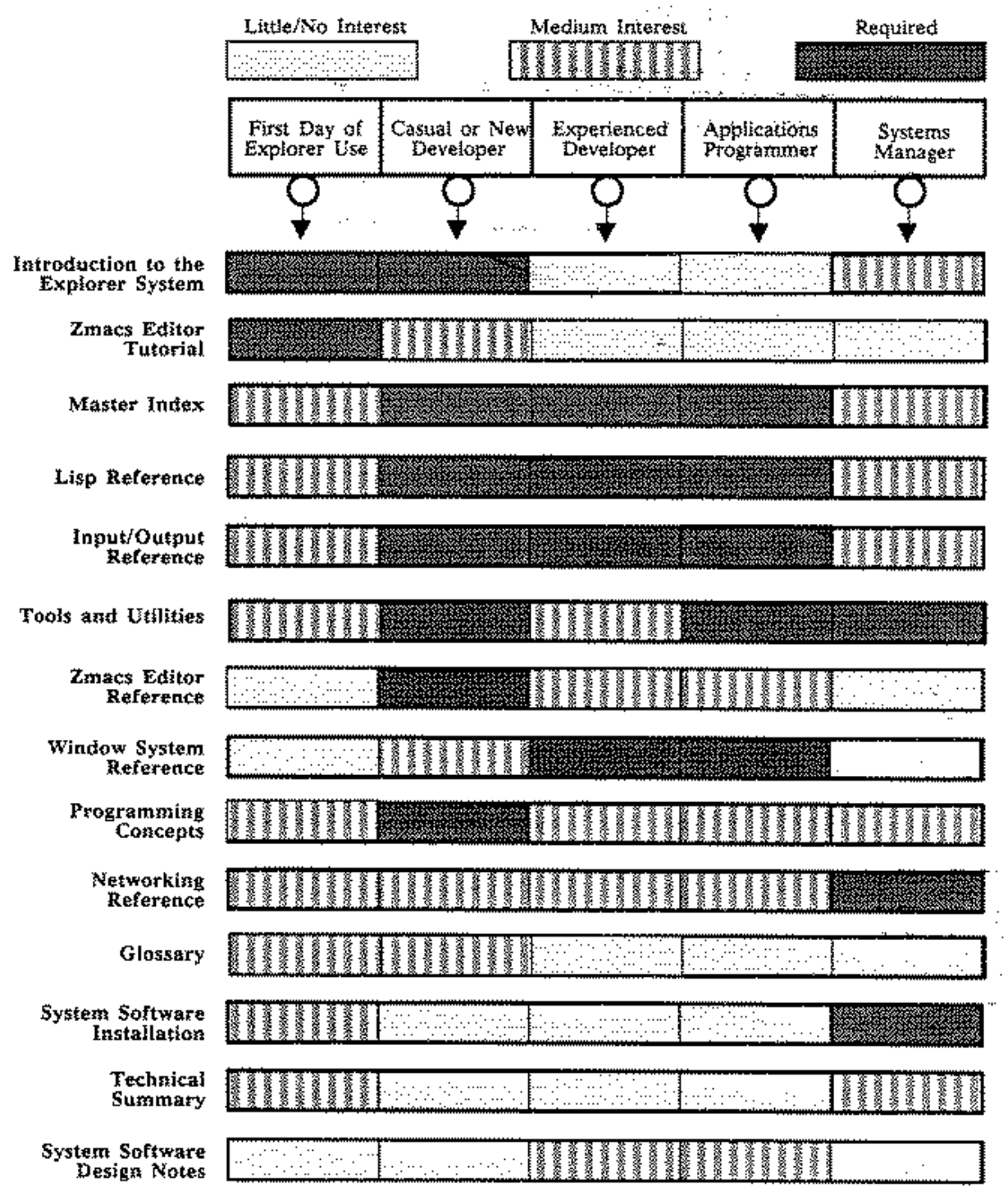
The system-defined windows shown in this manual are examples of the software as this manual goes into production. Later changes in the software may cause the windows on your system to be different from those in the manual.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

Texas Instruments Incorporated
ATTN: Data Systems Group, M/S 2151
P.O. Box 2909
Austin, Texas 78769-2909

THE EXPLORER™ SYSTEM SOFTWARE MANUALS



THE EXPLORER™ SYSTEM SOFTWARE MANUALS

Mastering the Explorer Environment	Explorer Technical Summary	2243189-0001
	Introduction to the Explorer System	2243190-0001
	Explorer Zmacs Editor Tutorial	2243191-0001
	Explorer Glossary	2243134-0001
	Explorer Networking Reference	2243206-0001
	Explorer Diagnostics	2533554-0001
	Explorer Master Index to Software Manuals	2243198-0001
	Explorer System Software Installation Guide	2243205-0001
Programming With the Explorer	Explorer Programming Concepts	2549830-0001
	Explorer Lisp Reference	2243201-0001
	Explorer Input/Output Reference	2549281-0001
	Explorer Zmacs Editor Reference	2243192-0001
	Explorer Tools and Utilities	2549831-0001
	Explorer Window System Reference	2243200-0001
Explorer Options	Explorer Natural Language Menu System User's Guide	2243202-0001
	Explorer Relational Table Management System User's Guide	2243203-0001
	Explorer Grasper User's Guide	2243135-0001
	Explorer Prolog User's Guide	2537248-0001
	Programming in Prolog, by Clocksin and Mellish	2537157-0001
	Explorer Color Graphics User's Guide	2537157-0001
	Explorer TCP/IP User's Guide	2537150-0001
	Explorer LX™ User's Guide	2537225-0001
	Explorer LX System Installation	2537227-0001
	Explorer NFS™ User's Guide	2546890-0001
	Explorer DECnet™ User's Guide	2537223-0001
	Personal Consultant™ Plus Explorer	2537259-0001
System Software Internals	Explorer System Software Design Notes	2243208-0001
	Release Information, Explorer System Software	2549844-0001

Explorer and NuBus are trademarks of Texas Instruments Incorporated.
Explorer LX is a trademark of Texas Instruments Incorporated.
NFS is a trademark of Sun Microsystems, Inc.
DECnet is a trademark of Digital Equipment Corporation.
Personal Consultant is a trademark of Texas Instruments Incorporated.

THE EXPLORER™ SYSTEM HARDWARE MANUALS

System Level Publications	Explorer 7-Slot System Installation	2243140-0001
	Explorer System Field Maintenance	2243141-0001
	Explorer System Field Maintenance Documentation Kit	2243222-0001
	Explorer System Field Maintenance Supplement	2537183-0001
	Explorer System Field Maintenance Supplement Documentation Kit	2549278-0001
	Explorer NuBus™ System Architecture General Description	2537171-0001
System Enclosure Equipment Publications	Explorer 7-Slot System Enclosure General Description	2243143-0001
	Explorer Memory General Description (8-megabytes)	2533592-0001
	Explorer 32-Megabyte Memory General Description	2537185-0001
	Explorer Processor General Description	2243144-0001
	68020-Based Processor General Description	2537240-0001
	Explorer II Processor and Auxiliary Processor Options General Description	2537187-0001
	Explorer System Interface General Description	2243145-0001
	Explorer NuBus Peripheral Interface General Description (NUPI board)	2243146-0001
Display Terminal Publications	Explorer Display Unit General Description	2243151-0001
	CRT Data Display Service Manual, Panasonic code number FTD85055057C	2537139-0001
	Model 924 Video Display Terminal User's Guide	2544365-0001
143-Megabyte Disk/Tape Enclosure Publications	Explorer Mass Storage Enclosure General Description	2243148-0001
	Explorer Winchester Disk Formatter (ADAPTEC) Supplement to Explorer Mass Storage Enclosure General Description	2243149-0001
	Explorer Winchester Disk Drive (Maxtor) Supplement to Explorer Mass Storage Enclosure General Description	2243150-0001
	Explorer Cartridge Tape Drive (Cipher) Supplement to Explorer Mass Storage Enclosure General Description	2243166-0001
	Explorer Cable Interconnect Board (2236120-0001) Supplement to Explorer Mass Storage Enclosure General Description	2243177-0001
143-Megabyte Disk Drive Vendor Publications	XT-1000 Service Manual, 5 1/4-inch Fixed Disk Drive, Maxtor Corporation, part number 20005 (5 1/4-inch Winchester disk drive, 112 megabytes)	2249999-0001
	ACB-5500 Winchester Disk Controller User's Manual, Adaptec, Inc., (formatter for the 5 1/4-inch Winchester disk drive)	2249933-0001

1/4-Inch Tape Drive Vendor Publications	Series 540 Cartridge Tape Drive Product Description, Cipher Data Products, Inc., Bulletin Number 01-311-0284-1K (1/4-inch tape drive)	2249997-0001
	MT01 Tape Controller Technical Manual, Emulex Corporation, part number MT0151001 (formatter for the 1/4-inch tape drive)	2243182-0001
182-Megabyte Disk/Tape Enclosure MSU II Publications	Mass Storage Unit (MSU II) General Description	2537197-0001
182-Megabyte Disk Drive Vendor Publications	Control Data® WREN™ III Disk Drive OEM Manual, part number 77738216, Magnetic Peripherals, Inc., a Control Data Company	2546867-0001
515-Megabyte Mass Storage Subsystem Publications	SMD/515-Megabyte Mass Storage Subsystem General Description (includes SMD/SCSI controller and 515-megabyte disk drive enclosure)	2537244-0001
515-Megabyte Disk Drive Vendor Publications	515-Megabyte Disk Drive Documentation Master Kit (Volumes 1, 2, and 3), Control Data Corporation	2246129-0002
	Volume 1, General Description, Operation, Installation and Checkout, and Part Data	2246125-0004
	Volume 2, Theory, General Maintenance, Trouble Analysis, Electrical Checks, and Repair Information	2246125-0005
	Volume 3, Diagrams	2246125-0006
1/2-Inch Tape Drive Publications	MT3201 1/2-Inch Tape Drive General Description	2537246-0001
1/2-Inch Tape Drive Vendor Publications	Cipher CacheTape® Documentation Manual Kit (Volumes 1 and 2 With SCSI Addendum and, Logic Diagram), Cipher Data products	2246130-0001
	1/2-Inch Tape Drive Operation and Maintenance (Volume 1), Cipher Data Products	2246126-0001
	1/2-Inch Tape Drive Theory of Operation (Volume 2), Cipher Data Products	2246126-0002
	SCSI Addendum With Logic Diagram, Cipher Data Products	2246126-0003

Control Data is a registered trademark of Control Data Corporation.
WREN is a trademark of Control Data Corporation.
CacheTape is a registered trademark of Cipher Data Products, Inc.

Printer Publications	Model 810 Printer Installation and Operation Manual	2311356-9701
	Omni 800™ Electronic Data Terminals Maintenance Manual for Model 810 Printers	0994386-9701
	Model 850 RO Printer User's Manual	2219890-0001
	Model 850 RO Printer Maintenance Manual	2219896-0001
	Model 850 XL Printer User's Manual	2243250-0001
	Model 850 XL Printer Quick Reference Guide	2243249-0001
	Model 855 Printer Operator's Manual	2225911-0001
	Model 855 Printer Technical Reference Manual	2232822-0001
	Model 855 Printer Maintenance Manual	2225914-0001
	Model 860 XL Printer User's Manual	2239401-0001
	Model 860 XL Printer Maintenance Manual	2239427-0001
	Model 860 XL Printer Quick Reference Guide	2239402-0001
	Model 860/859 Printer Technical Reference Manual	2239407-0001
	Model 865 Printer Operator's Manual	2239405-0001
	Model 865 Printer Maintenance Manual	2239428-0001
	Model 880 Printer User's Manual	2222627-0001
	Model 880 Printer Maintenance Manual	2222628-0001
	OmniLaser™ 2015 Page Printer Operator's Manual	2539178-0001
	OmniLaser 2015 Page Printer Technical Reference	2539179-0001
	OmniLaser 2015 Page Printer Maintenance Manual	2539180-0001
	OmniLaser 2108 Page Printer Operator's Manual	2539348-0001
	OmniLaser 2108 Page Printer Technical Reference	2539349-0001
	OmniLaser 2108 Page Printer Maintenance Manual	2539350-0001
	OmniLaser 2115 Page Printer Operator's Manual	2539344-0001
	OmniLaser 2115 Page Printer Technical Reference	2539345-0001
	OmniLaser 2115 Page Printer Maintenance Manual	2539356-0001
Communications Publications	990 Family Communications Systems Field Reference	2276579-9701
	EI990 Ethernet® Interface Installation and Operation	2234392-9701
	Explorer NuBus Ethernet Controller General Description	2243161-0001
	Communications Carrier Board and Options General Description	2537242-0001

Omni 800 is a trademark of Texas Instruments Incorporated.
OmniLaser is a trademark of Texas Instruments Incorporated.
Ethernet is a registered trademark of Xerox Corporation.

TABLE of CONTENTS

Paragraph Title

Preface

SECTION 1 Hardware Overview

1.1 INTRODUCTION
1.1.1 Explorer Processor
1.1.2 Explorer Buses and Components
1.1.2.1 The A-Bus
1.1.2.2 The M-Bus
M-Memory
The PDL, PDL Pointer, and PDL Index
The MD Register
The VMA Register
The O-Bus
1.1.3 Microcode
1.1.3.1 I-Memory
1.1.3.2 Instructions
1.1.3.3 Labeling Conventions
1.2 MAJOR COMPONENTS
1.2.1 System Enclosure
1.2.2 Brief Descriptions of Each Board
1.2.3 Monitor
1.2.4 Mass Storage Subsystem
1.3 INTRASYSTEM COMMUNICATION
1.3.1 NuBus
1.3.1.1 Block Moves
1.3.1.2 Time-Out Period
1.3.1.3 Parity
1.3.1.4 Reset Operation
1.3.1.5 Power Fail
1.3.1.6 System Addressing
1.3.1.7 Configuration (ID) ROM
1.3.1.8 Bus Locking
1.3.1.9 NuBus Clock
1.3.1.10 Unimplemented Memory
1.3.1.11 Byte, Halfword, and Word Transfer Protocols
1.3.2 Explorer Bus (Local Bus)

1.3.3 I/O Connectors
1.4 INTERSYSTEM COMMUNICATION
1.5 EXPLORER SYSTEM DEFINED ADDRESSES

SECTION 2 Bootstrap Loading

2.1 INTRODUCTION
2.2 TYPES OF LOADS
2.3 POWER ON / SYSTEM RESET
2.4 CONFIGURATION ROM
2.5 THE STBM AND DEVICE INDEPENDENCE
2.6 NVRAM FORMAT
2.7 ERROR CODES AND MESSAGES
2.7.1 STBM ROMS - Error Codes
2.7.2 Menu Boot and Primitive - Error Codes and Error Messages
2.7.2.1 Menuboot and Primitive Error Messages
2.8 LIGHT CODE TABLE
2.8.1 Explorer I STBM LED Codes
2.8.2 Explorer II STBM LED Codes
2.9 MICROLOADS
2.10 INTERFACE BETWEEN BOOT PROM AND MICROLOADS
2.11 CONFIGURATION PARTITION, PRIM ALGORITHM, AND DOWNLOADING
2.11.1 The Algorithm
2.11.2 Downloading

SECTION 3 Interrupts

3.1 INTRODUCTION
3.2 INTERRUPTS ON THE EXPLORER PROCESSOR

SECTION 4 Device Handling

4.1 INTRODUCTION
4.2 DEVICE DECODING
4.3 DEVICE-HANDLER DATA STRUCTURE ALLOCATION
4.4 DEVICE DETAILS
4.4.1 The NuBus Peripheral Interface Board
4.4.2 The Keyboard
4.4.3 CRT Vertical Retrace (Interrupt)
4.5 SERIAL AND PARALLEL IO PORTS
4.5.1 RS232 Serial Port
4.5.2 Parallel Port

SECTION 5 The Virtual Memory System

5.1 VIRTUAL MEMORY
5.2 PHYSICAL ADDRESSES
5.3 VIRTUAL ADDRESSES
5.4 PHYSICAL MEMORY USE
5.5 VIRTUAL MEMORY PARTITIONING
5.6 EXPLORER I MAPPING HARDWARE
5.7 EXPLORER I REVERSE FIRST LEVEL MAP
5.8 EXPLORER I MEMORY MAP ALLOCATION
5.9 EXPLORER II MEMORY MAP HARDWARE
5.9.1 Explorer II Map Usage Table (MUT)
5.10 EXPLORER II MEMORY MAP ALLOCATION
5.11 PAGE EXCEPTION HANDLING
5.11.1 Memory Access Codes
5.11.2 Memory Map Status Codes
5.11.2.1 Status Code 0: Map Miss
5.11.2.2 Status Code 1: Meta Bits Only
5.11.2.3 Status Code 2: Read Only
5.11.2.4 Status Code 3: Read/Write First
5.11.2.5 Status Code 4: Read/Write
5.11.2.6 Status Code 5: Page might be in PDL Buffer
5.11.2.7 Status Code 6: Possible MAR Trap
5.11.3 GCV and Oldspace Exceptions
5.11.3.1 Oldspace
5.11.3.2 Garbage Collection Volatility Faults
5.12 VIRTUAL MEMORY SYSTEM TABLES
5.12.1 Page Hash Table
5.12.2 Physical Page Data Table
5.13 A-MEMORY PAGING INFORMATION
5.13.1 PDL Buffer Handling
5.13.2 Physical Memory Map

SECTION 6 Paging and Disk Management

6.1 VIRTUAL PAGE MANAGEMENT
6.2 PAGE MANAGEMENT OVERVIEW
6.3 PAGE HASH TABLE
6.3.1 Page Replacement Process
6.3.2 Page Table Sizes
6.3.3 PHT Hashing Algorithm
6.4 DISK PAGE MAPPING SCHEME
6.5 LOGICAL PAGING DEVICES
6.6 VIRTUAL MEMORY SYSTEM SUBPRIMITIVES

SECTION 7 Internal Storage Formats

7.1 INTRODUCTION
7.2 THE TAGGED ARCHITECTURE
7.3 BOXED VERSUS UNBOXED MEMORY
7.4 DATA TYPE SUMMARY
7.5 IMMEDIATE DATA TYPES
7.5.1 DTP-Fix
7.5.2 DTP-Character
7.5.3 DTP-Short-Float
7.6 POINTER OBJECT TYPES
7.7 LIST POINTER OBJECTS
7.7.1 DTP-List
7.7.2 CONS Cells
7.7.3 Cdr-Coded Lists
7.7.4 Destructive Operations: RPLACA, RPLACD
7.7.5 DTP-Stack-List
7.7.6 Closures
7.8 STRUCTURE POINTER OBJECTS
7.8.1 Symbols
7.8.2 Instances
7.8.3 Arrays
7.8.3.1 Stack Groups
7.8.3.2 Regular PDL Arrays
7.8.3.3 Special PDL Arrays
7.8.4 Compiled Functions (FEFs)
7.8.5 DTP-Header
7.8.6 DTP-Extended-Number
7.8.6.1 BIGNUM
7.8.6.2 RATIONAL
7.8.6.3 COMPLEX
7.8.7 Floating Point Numbers
7.8.7.1 Short Float
7.8.7.2 Single Float
7.8.7.3 Double Float
7.8.8 The Cdr Code Field in Structures
7.9 OTHER POINTER OBJECTS
7.9.1 DTP-Locative
7.9.2 DTP-U-Entry
7.10 HOUSEKEEPING TYPES
7.10.1 Forwarding Pointer Types
7.10.1.1 DTP-One-Q-Forward
7.10.1.2 DTP-External-Value-Cell-Pointer
7.10.1.3 GC Forwarding Pointers
7.10.2 Structure Forwarding
7.10.3 DTP-Self-Ref-Pointer
7.10.4 Trap Types
7.10.4.1 DTP-Null
7.10.4.2 DTP-Trap and DTP-Ones-Trap
7.10.4.3 Unused Type DTP-Free

SECTION 8 Arrays

8.1 ARRAYS
8.1.1 Array Type
8.1.2 Array Leaders
8.1.3 Simple Bit
8.1.4 Named Structure Flag
8.1.5 Index Length Field
8.1.6 Number of Dimensions Field
8.1.7 Displaced Bit
8.1.8 Physical Bit
8.1.9 Option Words
8.1.9.1 Long Length Word
8.1.9.2 Dimension Words
8.1.9.3 Displaced Array Option Words

SECTION 9 Storage Management

9.1 INTRODUCTION
9.1.1 Areas and Regions
9.1.2 Address Space Allocation and Use
9.2 AREAS
9.3 REGIONS
9.3.1 Region Bits
9.4 STANDARD AREAS
9.5 SYSTEM COMMUNICATION AREA

SECTION 10 Garbage Collection

10.1 INTRODUCTION
10.2 BASICS
10.2.1 Garbage Collection Spaces
10.2.2 Scavenging
10.2.3 Incremental Collection
10.2.4 Generational Garbage Collection
10.2.5 Scavenge Space
10.3 EXPLORER TGC IMPLEMENTATION
10.3.1 Generations
10.3.2 Areas and Regions
10.3.2.1 Volatility
10.3.2.2 Default Cons Generation
10.3.3 Automatic Collection Mode
10.3.4 Batch Collections
10.3.5 Scavenging for TGC
10.3.6 Indirection Cells
10.3.7 Following QCYP Forwarding

10.3.8 Promotion
10.3.9 Locked Volatility 0 Areas
10.4 GC SUBPRIMITIVES AND VARIABLES
10.4.1 Flipping, Scavenging and Reclaiming Oldspace
10.4.2 GC Predicates
10.4.3 Starting and Stopping Automatic GC
10.4.4 TGC Control
10.4.5 Number Consing
10.4.6 Miscellaneous

SECTION 11 Function-Calling

11.1 INTRODUCTION
11.2 FUNCTIONAL OBJECTS
11.2.1 Compiled Functions
11.2.2 Interpreted Functions
11.2.3 Indirect Functions
11.2.3.1 Symbols
11.2.3.2 Closures
11.2.3.3 Instances, Named-Structures, and Funcallable Hash-Arrays
11.2.4 Non-Functions
11.2.4.1 Arrays
11.2.4.2 Stack-Groups
11.2.4.3 Microcode-Entry Functions
11.2.5 Obsolete Functional Objects
11.3 HISTORY AND MOTIVATIONS
11.4 STACK LAYOUT
11.5 THE CALL-INFO WORD
11.5.1 Call Fields
11.5.2 Return Fields
11.5.3 State Fields
11.6 FEF LAYOUT
11.7 CALLING A FEF
11.7.1 The Instructions
11.7.2 The Actions
11.8 CALLING ANYTHING ELSE
11.8.1 Calling the Interpreter
11.8.2 Calling an Instance
11.8.3 Calling a Microcode-Entry Function
11.8.4 Calling "Out"
11.8.5 The Support Vector
11.9 RETURNING
11.9.1 Basics
11.9.2 Details
11.10 CATCH AND THROW
11.11 CATCH
11.12 THROW

11.13 UNWIND-STACK

SECTION 12 Closures

12.1 INTRODUCTION
12.2 DYNAMIC CLOSURES
12.3 LEXICAL CLOSURES
12.3.1 Lexical Closure Implementation
12.3.2 Dedicated Locals
12.3.3 Calling a Lexical Closure
12.3.4 Free Lexical References
12.3.5 Constructing Lexical Closures
12.3.6 Lexical Environments
12.3.7 Environment Descriptor List
12.3.8 Managing Environments

SECTION 13 Flavor Internals

13.1 INTRODUCTION
13.2 FLAVOR NAMES
13.3 FLAVOR INSTANCES
13.4 FLAVOR STRUCTURE
13.5 FLAVOR METHOD-TABLE (FLAVOR-METHOD-TABLE slot of Flavor structure)
13.6 INSTANCE VARIABLES ACCESS
13.7 FLAVOR MAPPING TABLES
13.8 FLAVOR METHOD HASH TABLE (FLAVOR-METHOD-HASH-TABLE slot of Flavor structure
13.9 FLAVOR COMPOSITION (COMPOSE-FLAVOR-COMBINATION)
13.10 METHOD COMPOSITION (COMPOSE-METHOD-COMBINATION)
13.11 METHOD COMBINATION
13.12 HAVE-COMBINED-METHODS
13.13 INSTANCE VARIABLE ADDRESSING
13.14 MAPPING TABLE REFERENCE
13.15 METHOD INTEGRATION
13.16 COMPILE FLAVOR METHODS
13.17 DEFFLAVOR
13.18 DEFMETHOD
13.19 METHOD LOADING
13.20 MAKE-INSTANCE (INSTANTIATE-FLAVOR)
13.21 INSTANCE CALLING (SEND)

SECTION 14 Stack Groups

14.1 INTRODUCTION
14.2 THE STACK GROUP DATA STRUCTURE
14.2.1 Static Section
14.2.2 Debugging Section
14.2.3 High Level Section
14.2.4 Dynamic Section
14.3 SPECIAL PDL

SECTION 15 Storage Subprimitives

15.1 STORAGE SUBPRIMITIVES
15.2 DANGERS OF SUBPRIMITIVES
15.3 STORAGE LAYOUT DEFINITIONS
15.4 DATA TYPES
15.5 POINTER MANIPULATION
15.6 POINTER ARITHMETIC
15.7 FORWARDING
15.8 ANALYZING STRUCTURES
15.9 CREATING OBJECTS
15.10 RETURNING STORAGE
15.11 COPYING DATA
15.12 SPECIAL MEMORY REFERENCING
15.12.1 Subprimitives for Boxed Memory Referencing
15.12.2 Subprimitives for Unboxed Memory Referencing
15.13 TQC SUBPRIMITIVE SEMANTICS
15.13.1 QC Young Pointer Usage
15.13.2 Dividing the Subprimitives
15.13.3 Structure Forwarding Considerations
15.13.4 Use of Group 2 Subprimitives on "SAFE" Boxed Storage
15.13.5 Special Cases of %BLT-TYPED and %BLT

SECTION 16 Other Subprimitives, Variables, and Counters

16.1 INTRODUCTION
16.2 ARRAY SUBPRIMITIVES
16.3 STACK LIST SUBPRIMITIVES
16.4 FUNCTION-CALLING SUBPRIMITIVES
16.5 SPECIAL-BINDING SUBPRIMITIVE
16.6 CLOSURE SUBPRIMITIVES
16.7 LOCKING SUBPRIMITIVE
16.8 EXPLORER I/O DEVICE SUBPRIMITIVES
16.9 SUBPRIMITIVES TO SHUT DOWN THE LISP ENVIRONMENT
16.10 VIRTUAL MEMORY SYSTEM SUBPRIMITIVES
16.11 GARBAGE COLLECTION SUBPRIMITIVES

16.12 MICROCODE VARIABLES
16.12.1 M-Memory Variables
16.12.2 A-Memory Variables
16.13 MICROCODE COUNTERS
16.13.1 Accessing Counters
16.13.2 Page Exception Handling Counters
16.13.3 Page Fault Handling Counters
16.13.4 Page Replacement Algorithm Counters
16.13.5 PHT and PPD Information Counters
16.13.6 GC Counters
16.13.7 Hardware Fault Detection Counters
16.13.8 Miscellaneous Counters

SECTION 17 Error Handling

17.1 INTRODUCTION
17.2 MICROCODE ERROR CONDITIONS
17.2.1 Microcode Error Table
17.2.1.1 Special Error Table Entries
17.2.1.2 Normal Error Table Entries

SECTION 18 Crash Handling

18.1 ILLEGAL OPERATION
18.2 CRASH RECORDING
18.2.1 Crash Record Allocation
18.2.2 Crash Record Contents
18.2.3 The Crash Table

SECTION 19 Compiler Notes

19.1 COMPILE-TIME PROPERTIES OF SYMBOLS
19.2 DECLARATIONS
19.3 XLD FILES
19.4 OPTIMIZATION
19.5 COLD-LOAD ATTRIBUTE

SECTION 20 Macro-Instructions

20.1 INTRODUCTION
20.2 MAIN OPS
20.3 SHORT BRANCHES

20.4 IMMEDIATE OPERATIONS
20.5 CALL INSTRUCTIONS
20.6 MISC-OPS
20.7 AUX-OPS
20.7.1 AUX-OP Complex Call
20.7.2 AUX-OP Long Branch
20.7.3 AUX-OPS With Count Field
20.8 AREFI INSTRUCTIONS
20.9 MODULE GROUP
20.10 UCODE ENTRIES
20.11 MACROCODE INSTRUCTION SET

APPENDIX A Data Structures

A.1 NONVOLATILE RAM (NVRAM)
A.1.1 NVRAM Data Structure Definitions
A.1.1.1 Test and Boot Resources
A.1.1.2 Last Shutdown Information
A.1.1.3 Crash Record Registers
A.1.1.4 Typed Blocks
A.1.1.5 NVRAM Format
A.2 DISK PARTITION STRUCTURES
A.2.1 Volume Label Partition
A.2.2 Partition Table Partition
A.2.3 Test Zone Partition
A.2.4 Format Information Partition
A.2.5 Partition Descriptions
A.3 MICRO-LOAD PARTITION FORMAT SPECIFICATION FOR THE EXPLORER II PROCESSOR
A.3.1 Introduction
A.3.1.1 Scope
A.3.1.2 Reference Documents
A.3.2 Overview
A.3.3 Detail Section Definitions
A.3.3.1 Microload Header Section
A.3.3.2 Instruction Memory Section
A.3.3.3 Dispatch Section
A.3.3.4 A&M Memory Section
A.3.3.5 Tag Classifier Section
A.3.3.6 I/O Space Initialization Section
A.3.3.7 I/O Space Data Section
A.3.3.8 Main Memory Section
A.3.3.9 Auxiliary Data Section
A.3.3.10 Entry Data Section

APPENDIX B Acronyms

B.1 ACRONYMS USED

LIST of TABLES

Table	Title	Paragraph
1-1	System-Defined Addresses	1.5
3-1	Explorer Control Space	3.2
3-2	Interrupt Vectors	3.2
5-1	Address Space Map Bits Assignment	5.9
5-2	Virtual Memory Maps Status	5.9
5-3	Virtual Memory Map	5.9
5-4	Map Usage Table (MUT)	5.9.1
5-5	MAR Status Codes	5.11.2.7
5-6	Physical Page Data Area Word Format	5.12.2
6-1	Page Hash Table: Swap Status Codes	6.3.1
7-1	Lisp Object Data Type Attributes	7.4
7-2	Housekeeping Data Types	7.4
7-3	Pointer Types	7.6
7-4	CDR Codes	7.7.1
7-5	Qs of Symbol	7.8.1
7-6	Header Types	7.8.5
7-7	SELF-REF-POINTER Format	7.10.3
8-1	Array Types	8.1.1
9-1	Area Attributes	9.2
9-2	Region Characteristics	9.3
9-3	Space Type Codes	9.3.1
10-4	Space Type Terminology	10.2.1
10-5	TGC Generations	10.3.1
10-6	Volatility Level Meanings	10.3.2.1
11-1	Contents of the Support Vector	11.8.5
12-1	Some Lexical Closure Examples	12.3
13-1	Mapping Table	13.7
13-2	Flavor Structure Definition	13.21
13-3	Flavor-Plist Properties	13.21
14-1	Stack Group States	14.2.4
14-2	Processor Flags	14.2.4
14-3	Special PDL Block Type	14.3
15-1	GROUP 1. Use on BOXED cells ONLY	15.13.2
15-2	GROUP 2. Use on UNBOXED or "SAFE" BOXED locations	15.13.2
18-1	Crash Record Allocation Registers	18.2.1
18-2	Crash Record Format	18.2.2
B-1	Description of Acronyms	B.1

LIST of FIGURES

Figure	Title	Paragraph
1-1	Backplane Configuration	1.3
1-2	Layout of Words, Halfwords, and Bytes	1.3.1
3-1	Device Descriptor Block	3.2
4-1	NUPI Device Descriptor Block	4.4.1
4-2	NUPI Device Descriptor Information Word	4.4.1
5-1	Explorer I Level-1 Map (LVL1)	5.6
5-2	Explorer I Level-2 Maps (LVL2)	5.6
5-3	Memory Map Access Codes	5.11.1
5-4	Memory Map Status Codes	5.11.2
5-5	PDL Buffer Wrap-Around	5.13.1
5-6	Physical Memory Map Entry	5.13.2
6-1	Page Hash Table Entry Format	6.3
6-2	PHT and PPD Sizes	6.3.2
6-3	Disk Page Mapping Table	6.4
6-4	Device Status Codes	6.4
6-5	Logical Page Device Information Block	6.5
7-1	G Format	7.2
7-2	Character Format	7.5.2
7-3	BIGNUM Structure	7.8.6.1
7-4	BIGNUM Header Format	7.8.6.1
7-5	BIGNUM Data Format	7.8.6.1
7-6	Short Float Format	7.8.7.1
7-7	Single Precision Float Structure Format	7.8.7.2
7-8	Double Precision Float Structure Format	7.8.7.3
8-1	Array Header Word	8.1
8-2	Array with Leader	8.1.2
9-1	Region Bits Area Entry Description	9.3.1
9-2	Map of Systems Communications Area	9.5
10-3	Indirection Cell Forwarding	10.3.6
11-1	Call-Frame Layout	11.4
11-2	Call-Info Word Layout	11.5
11-3	The Layout of a FEF	11.6
12-1	Lexical Closure Structure	12.3.1
12-2	Lexical Variable Slot Descriptor	12.3.7
13-1	Flavor Instance	13.3
20-1	MAIN OP Instruction Format	20.2
20-2	Short Branch Instruction Format	20.3
20-3	Immediate Operation Instruction Format	20.4
20-4	Call Instruction Format	20.5
20-5	MISC-OP Instruction Format	20.6
20-6	AUX-OP Instruction Format	20.7
20-7	AREFI Instruction Format	20.8
20-8	Module Group Instruction Format	20.9

Preface

This set of system design notes ships with each Explorer System, but is not expected to be of use to every user. It is provided for the benefit of system programmers who need more detailed understanding of the system structure and low-level interfaces than is documented in the user manuals, for the purpose of modifying, extending, or interfacing with the system at a lower level than the normal user interfaces, for low level debugging, or for performance analysis.

These notes include a simple description of the Explorer hardware, a description of elements of the Explorer Lisp machine virtual architecture, and several sets of material useful to systems programmers.

The following sections are included in this document:

Section 1 -- Hardware Overview - Covers high level information about the system hardware. More detail can be found in the hardware documents that ship with each system.

Section 2 -- Bootstrap Loading - Describes the various types of system loads and the related structures.

Section 3 -- Interrupts - Discusses the handling of interrupts.

Section 4 -- Device Handling - Explains the handling of I/O requests for various types of devices.

Section 5 -- Virtual Memory - Describes how the virtual memory mapping scheme works on the Explorer.

Section 6 -- Paging and Disk Management - Describes the demand paging scheme on the Explorer.

Section 7 -- Internal Storage Formats - Details the formats used for representing Lisp data items.

Section 8 -- Arrays - Details the formats used for representing arrays.

Section 9 -- Storage Management - Explains how memory is organized into areas and regions, and describes several specific areas used by the system software.

Section 10 -- Garbage Collection - Describes the techniques and structures used by the Explorer for handling garbage collection and for implementing Temporal Garbage Collection (TGC).

Section 11 -- Function Calling - Details the mechanisms used for function calling and the internal formats of compiled code.

Section 12 -- Closures - Describes how dynamic and lexical closures are implemented on the Explorer.

Section 13 -- Flavors - Discusses the data structures used for handling flavors.

Section 14 -- Stack Groups - Describes the stack group data structure and the special push down list.

Section 15 -- Storage Subprimitives - Explains the functionality and lists the arguments for low-level storage manipulating subprimitives and explains new semantics imposed on some of them by TQC.

Section 16 -- Other Subprimitives, Variables, and Counters - Explains the functionality and lists the arguments for non-storage-related low-level system functions, and documents all special microcode variables and counters.

Section 17 -- Error Handling - Describes microcode error handling and explains the various error conditions that can arise.

Section 18 -- Crash Handling - Explains how crash records are organized and how you can examine them from Lisp.

Section 19 -- Compiler Notes - Supplements section 21. Compiler Operations, of the Explorer Lisp Reference manual.

Section 20 -- Macro-Instructions - This section describes the macroinstruction set.

Appendix A -- Data Structures - Details the content of the non-volatile RAM and the disk partition structures, and describes the micro-load partition format specification for the Explorer II processor.

Appendix B -- Acronyms - List of acronyms used in this manual.

SECTION 1

Hardware Overview

1.1 INTRODUCTION

1.1.1 Explorer Processor.

The purpose of this section is to supply the reader with sufficient knowledge of the processor so that the terminology in the following sections can be understood. This section does not attempt to explain all there is about the processor. If the reader wants to know in detail about the processors, he or she should read the manuals SPECIFICATION: EXPLORER PROCESSOR, part number 2236414-0001 and/or SPECIFICATION: EXPLORER II PROCESSOR, part number 2540834.

There are two types of processor: The Explorer I and the Explorer II. The main difference in the two is that the Explorer I (the original processor) is implemented on a board and the Explorer II is implemented on a chip. There are other differences, but this section only deals with the common features.

The function of the processor is to execute code called microcode. An overview of microcode will be discussed later.

1.1.2 Explorer Buses and Components.

The internals of the Explorer processor appears to be an almost general purpose computer. Inside the Explorer there is memory and special registers which are manipulated by the microcode instructions. The memory and special registers are located on one of several buses which exist in the processor. The three major buses found in the processor are the A-Bus, the M-Bus, and the O-Bus.

1.1.2.1 The A-Bus.

The A-Bus has only one component which is a block of memory. The memory size is 1K by 32-bit words - this notation means 1K block of 32 bit words. The first 64 words of this memory block are special and will be discussed in the section on the M-Bus. (There is another component on the A-Bus, but it is transparent

to the microcoder and its purpose is to speed up accesses to and from the memory on the A-Bus.) The memory on the A-Bus is referred to as the A-Memory.

1.1.2.2 The M-Bus.

The M-Bus has many components on it. Only a few will be discussed.

M-Memory.

On the M-Bus there is a block of memory whose size is 64 words by 32-bits. This memory is referred to as M-Memory. The first 64 words of A-Memory are an exact image of M-Memory. That is, if microcode writes to location 5 of M-Memory, then location 5 of A-Memory also gets changed to image location 5 of M-Memory.

The PDL, PDL Pointer, and PDL Index.

PDL means Push Down List. The PDL is a cache stack of 1K by 32-Bit words. The PDL Pointer and PDL Index are both components on the M-Bus. The PDL Pointer points to the top of the stack (TOS). The PDL Index can be modified to point anywhere in the stack. Like any other stack, data can be pushed onto it or popped off it. It should be noted that the Lisp microcode uses the cache PDL as a cache portion of a larger PDL which resides in NuBus memory.

The MD Register.

The MD register is a 32 bit register. It can be thought of as the processor's window. When data is read from outside the processor it comes into the MD register. When data is to be written outside the processor, it is first placed into the MD register.

The VMA Register.

The VMA register works in conjunction with the MD. MD holds what is read or to be written and the VMA holds where the external data is to be read from or to be written to.

There are other components on the M-Bus, but these are not discussed in this document. See one of the specifications listed earlier for more details.

The O-Bus.

There is a component in the processor called the Arithmetic Logic Unit (ALU). Its function is to take data from the M-Bus and the A-Bus and perform arithmetic or logical operations on that data (like addition). When the ALU performs that operation, it puts the result on the O-Bus. The O-Bus in turn selects which

component to store the result in. A- and M-Memory are both on the output side of the O-Bus along with M-Bus registers like MD or VMA. This means the O-Bus can select these components to store its data in.

1.1.3 Microcode.

1.1.3.1 I-Memory.

I-Memory is another memory block in the Explorer processor. I-Memory is used to store the microcode instructions. The size of I-Memory on the Explorer I is 16K by 56-Bits. The size of I-Memory on the Explorer II is 32K by 64-bits. I-Memory is sometimes referred to as Writable Control Store or WCS.

1.1.3.2 Instructions.

Microcode has a lot of similarity to assembly language in that it is very close to the actual architecture of the machine. There are four basic types of microcode instructions, but there are many ways to augment each instruction type. The four types are

- The ALU instruction. This instruction type allows the microcoder to perform arithmetic or logical instructions on an A-Memory value and an M-Memory value or an M-Bus register and store the results into a specified destination.

- The Byte Instruction. This instruction type allows the microcoder to take a specified portion of an M-Memory Word or M-Bus Register and deposit it into an A-Memory word and then place the result into a specified destination.

- The Jump Instruction. The Jump instruction can take on many forms with augmentation. For example, the Jump instruction can do a conditional or unconditional Jump; it can conditionally or unconditionally return from a subroutine call; and, it can conditionally or unconditionally perform a subroutine call.

- The Dispatch Instruction. The Dispatch instruction is something like a Jump instruction; however, its destination is not located in the instruction, but rather in a table called the dispatch table. This table is located in special memory on the M-Bus called Dispatch Memory. The Dispatch instruction uses an M-Bus source as an index into the table. Control then jumps or calls the instruction whose address is in that

particular location in Dispatch Memory.

1.1.3.3 Labeling Conventions.

There are labeling conventions for A- and M-Memory. If a label is assigned to an A-Memory location, it begins with the characters 'A-'. So a temporary location in A-Memory might be labeled A-TEMP. The corresponding convention is true for M-Memory labeling. An M-Memory location might be called M-SAVE. The reason this is important to the reader is that certain M-Memory and A-Memory locations have special meaning to the Lisp microcode and are referenced in this document. An example is the location A-BOOT-COMMAND-BLOCK which is a flag to the Lisp microcode just after booting occurs.

1.2 MAJOR COMPONENTS

The Explorer system consists of three major hardware components: the system enclosure, the monitor, and the mass storage subsystem. It is expected that over time additional components and peripherals will be added to form a broad family of compatible systems.

1.2.1 System Enclosure.

The system enclosure consists of a seven-slot card chassis housed in a low profile cabinet that fits under a desk. The backplane for the chassis provides the vehicle for all I/O connections and contains a NuBus for board-to-board interconnection. The cabinet also contains a power supply and provides two switched outlets for the monitor and mass storage unit.

1.2.2 Brief Descriptions of Each Board.

Explorer processor

- Explorer I - 500-IC microcoded Lisp engine
- Explorer II - Lisp engine containing Lisp microprocessor chip
- Two-level, demand-paged virtual memory map
- Self-test ability
- Private bus (local bus) to Explorer memory board and graphics display (Explorer I only)

Explorer memory board

- 2mb, 4mb, 8mb, 16mb, 32mb versions with byte parity
- Two ports - NuBus and the Explorer private bus (on 2, 4 and 8mb boards)
- Interfaces Explorer processor to NuBus
- Read-only memory (ROM) contains board identification information and extensive board diagnostics

System Interface board (SIB)

- Time of day and interval timers
- Non-volatile memory (NVRAM) - the NVRAM is a 2kbyte stable storage RAM on the system interface board. It is used during system testing and booting to locate resources and later for storing and retrieving a variety of other information.
- Monitor, keyboard and mouse controls
- RS-232C port and an eight-bit parallel port
- ROM containing board identification, board diagnostics, and generic device drivers for the monitor and keyboard

NuBus Peripheral Interface (NUPI) board

- NuBus to SCSI intelligent adaptor

- Extensive self-test

- ROM contains board identification information and system load device driver

Local Area Network (LAN) board

- Ethernet 10 MHZ interface

- Extensive on-board packet buffers directly accessible via NuBus

- ROM containing board identification information, diagnostics, and system load device driver.

1.2.3 Monitor.

The landscape black and white monitor is connected to the system enclosure via a dual fiber-optic cable, allowing placement of the monitor remote from the system enclosure. The keyboard and mouse connect to the front of the monitor. The mouse provides rapid, smooth cursor movement and positioning using a 200 dots per inch optical pad.

1.2.4 Mass Storage Subsystem.

The mass storage subsystem provides two 5 1/4" envelopes, power, and cooling for the mass storage system. Winchesters or cartridge tape components may be inserted in either of these two envelopes.

Connection to the system enclosure is via an SCSI interface cable.

1.3 INTRASYSTEM COMMUNICATION

The system backplane provides three connectors (P1, P2, and P3) per slot. These three connectors are used for all connections to the board. There are no front edge connectors. The backplane dedicates connector P1 as the NuBus, the primary system bus. P2 is used for the local bus on certain slots, and is available for general I/O on other slots. P3 is a general I/O connector on all slots. The configuration of the backplane is shown in Figure 1-1.

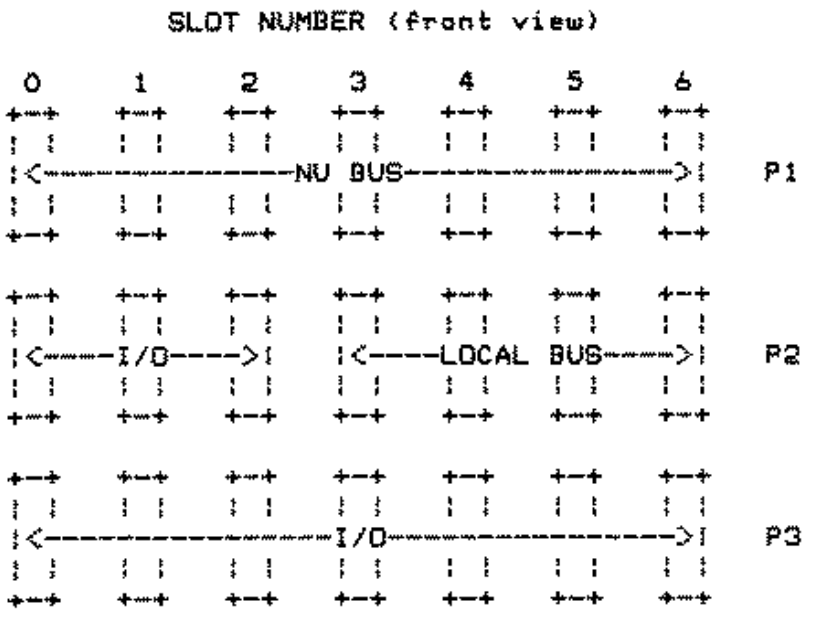


Figure 1-1 Backplane Configuration

1.3.1 NuBus.

The NuBus is the primary mechanism for board-to-board communication. The NuBus is a 32-bit system bus on which the address and data lines are multiplexed onto the same lines. All control, data and power lines are defined so as to fit through one 96 pin DIN connector. On the Explorer backplane, connector P1 is reserved in all slots for the NuBus.

Data may consist of 8 bits (one byte), 16 bits (one halfword), or 32 bits (one word). The bit and byte numbering scheme is referred to as little endian in which both the lowest numbered bit and byte is the least significant. Bytes, halfwords, and words are organized as shown in Figure 1-2.

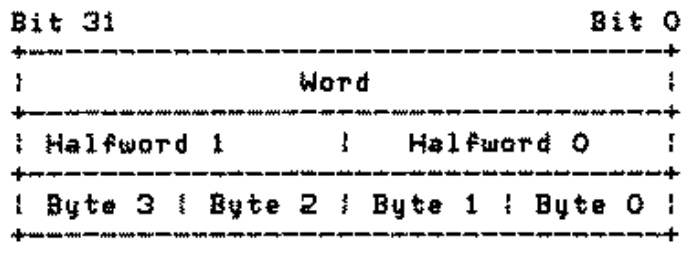


Figure 1-2 Layout of Words, Halfwords, and Bytes

The NuBus spec leaves some areas open or subject to interpretation for a given system. The following paragraphs define these areas as they apply to the Explorer system.

1.3.1.1 Block Moves.

The ability to perform block moves as defined by the NuBus spec is required on any large block of memory that may be considered a system resource. Small blocks of memory used as a buffer on a peripheral controller or a block of memory used only as a private memory and located in the control address space need not support block moves. All boards indicate via the ID ROM flags the presence or absence of block support on that board. On the Explorer only the memory board supports slave block moves.

1.3.1.2 Time-Out Period.

The bus time-out period for the Explorer is fixed at 25 microseconds (ms).

1.3.1.3 Parity.

Bus parity is not generated or checked by any Explorer component board.

1.3.1.4 Reset Operation.

On power-up, the source (the System Interface Board) that drives the RESET/ signal will hold it active (low) for at least 100 ms to guarantee the correct startup of certain MOS devices. After RESET/ goes inactive (high), all boards that execute self-tests will proceed with their tests.

RESET/ may be driven low at any time by any device to cause a system restart. This would typically be done by a "deadman" timer option or a remote diagnostic option. In this case, all boards will execute self-test and the system will reboot. RESET/ must be low for two or more clock cycles to force the system restart.

1.3.1.5 Power Fail.

The System Interface Board posts a power fail event to those boards that need a power fail warning. Software must program the SIB with the specific addresses for power fail events.

1.3.1.6 System Addressing.

Starting addresses for the boards are determined by slot location. In hexadecimal form, this address is >FS000000 (sometimes appears as F(ID)000000). The most significant hexadecimal digit, F, is the largest hexadecimal number, corresponding to binary 1111. The next digit, S, is the slot number given in hexadecimal. The remaining six zeros are hexadecimal digits, each corresponding to binary 0000.

1.3.1.7 Configuration (ID) ROM.

Each board has an ID ROM that describes the basic characteristics of that board. This ROM may be accessed at addresses F(S)FFFFFF - F(S)FFFF00 for any and all slot ID's. The ID ROM contains 64 or more bytes of information about each board, such as: serial number, part number, and board type. This ROM resides at the highest address of control space on each board and supplies one byte of data (byte 0) for each word address.

1.3.1.8 Bus Locking.

No bus master will lock the bus for more than five contiguous single-word transfers or one 16-word block transfer. However, during power fail, the bus may be locked to quickly post power-fail events.

1.3.1.9 NuBus Clock.

The System Interface Board that drives the NuBus clock line makes provision to disable that clock output so that two such cards may be installed.

1.3.1.10 Unimplemented Memory.

When a board recognizes an address that is within its control space but it has no memory at that address, it may either:

* Return ACK with timeout status

* Do nothing and allow the system watchdog to generate a timeout

1.3.1.11 Byte, Halfword, and Word Transfer Protocols.

The following bus protocols prevent unnecessary problems on bus transfers:

1. Slaves that provide only a byte interface position this byte in the least significant byte position (byte 0) of the NuBus word. Consecutive bytes appear in consecutive NuBus words, always at the byte 0 position. A master may read or write the slave using byte, halfword, or word operations. Data in any positions other than byte 0 must be ignored.

2. Slaves that provide only a 16-bit halfword interface position this in the least significant halfword position (halfword 0) of the NuBus word. A master may read or write the slave using byte, halfword, or word operations. Data in positions other than halfword 0 (bytes 0, 1) must be ignored.

3. Slaves that provide a full 32-bit word interface perform byte, halfword or word transfers.

With these guidelines, a master that can do only word transfers can communicate with any slave device. A master that is limited to byte or halfword transfers cannot talk to word-only slaves. All NuBus masters should have word transfer capability. Design of masters without 32-bit word capability is strongly discouraged.

1.3.2 Explorer Bus (Local Bus).

The Explorer Bus is a high-speed bus for private communication between one master and one or more slaves, such as a processor/memory board set. One Explorer bus is provided on the backplane using connector P2 of slots 3, 4, 5, and 6. An Explorer must have a memory board in slot 4 in order for the processor to boot. The memory interface logic is hardwired to slot 4. An additional Explorer bus or buses may be added by connecting two or three connectors together with a cable or

adapter board on the back side of the backplane. The Local Bus is used only by the Explorer I.

1.3.3 I/O Connectors.

Certain pins of P1, P2, and P3 are dedicated to power and ground functions.

1.4 INTERSYSTEM COMMUNICATION

All intersystem or external I/O connections are made from the back side of the backplane. Fiber optic links are the preferred method of interconnecting the system boxes to minimize radio frequency interference and electrostatic discharge effects and to reduce cable crowding.

The monitor to system chassis link carries video and audio data to the monitor, and receives audio, keyboard, and mouse information from the monitor.

The mass storage to system chassis link carries data and commands to and from the mass storage box.

At least one RS-232C interface and one eight-bit printer interface are provided on each system for low cost peripheral connection.

1.5 EXPLORER SYSTEM DEFINED ADDRESSES

While the Explorer system is based on the notion of dynamic configuration, several items must be fixed in order for the system to establish initial operation. Table 1-1 defines these fixed items.

Table 1-1 System-Defined Addresses

Address	Description
F6E00000	Explorer power fail event.
F6E00004 !! F6E0003C	All hardware-decoded interrupt events, except power fail, will fall within these addresses. The byte written to any interrupt even address must contain a 1 in bit 0. The remaining bits are reserved for future use and should also be set to 1.
FS000000	Starting address of memory that may be used by the system for general purpose, as indicated in the ROM flag register. This is the least significant byte of the memory block.
FSFFFF00 !! FSFFFFFF	Configuration ROM, first 64 bytes. Larger configuration ROMs will occupy the contiguous space below these addresses.
F(SIB)Fxxxxx	Bus timeout address. Nothing shall respond to this address, so a read or write at this address tests timeout generation.

SECTION 2

Bootstrap Loading

2.1 INTRODUCTION

Bootstrap loading the Explorer system involves several stages and several types of loads. While the remainder of this document is involved with system operation, bootstrap loading occurs before the system is operational and is, therefore, quite different from what follows in later sections.

2.2 TYPES OF LOADS

There are four type of loads:

* Power on / System reset. These are exactly the same except for the way they are initiated. A power on boot is performed when power is cycled on the main chassis. A system reset is performed by issuing the boot key chord META-CTRL-META-CTRL-ABORT. Much goes on during this type of load, but when the LISP system is being loaded, the primary software entity that gets loaded is the "primitive" microcode (from now on this will be referred to as the "primitive"). Microcode is loaded from a microload band (also known as a microcode partition) on secondary storage into the writable control store (WCS) of the processor. The system microcode in turn loads the system (the LISP system software) from a load band on secondary storage.

* Cold boot. This is performed by issuing the boot key chord META-CTRL-META-CTRL-RUBOUT. This causes the Explorer to reload the current system microcode and the current system load band from secondary storage. The previous environment is lost.

* Warm boot. This is performed by issuing the boot key chord META-CTRL-META-CTRL-RETURN. This causes the Explorer to restart the system code in such a way as to preserve the previous environment. Neither the system microcode nor the system load are re-read from secondary storage.

* Menu boot. This is performed by issuing the boot key chord META-CTRL-META-CTRL-M. This causes the Menuboot microload (see description of Menuboot in the Power on / Cold boot description below) to be loaded from the same unit that the current system load was loaded from. Menuboot is then executed as described below.

Note that each of the four boot key chords described above will cause the Explorer processor to take a true hardware trap (not just a polled event) to the appropriate point in microcode. So even if the processor is stuck in a loop, it should respond to any of the four chords. An additional key chord, META-CTRL-META-CTRL-C, can be used if the system appears to be in an infinite loop. This causes the system to crash and to record the currently running Lisp function.

If the system microcode in Writable Control Store (WCS) has been violated, then the Cold boot, Warm boot, and Menu boot may not function correctly. In this case you have no choice but to cycle power or do a System reset.

2.3 POWER ON / SYSTEM RESET

Both of these actions cause a NuBus reset which forces all boards that have a self-test to perform it. A NuBus reset causes the Explorer processor to enter its boot ROM based Microcode at a fixed location.

The first action it takes is to perform the processor self-tests.

Then all internal memories and registers are cleared.

If the Explorer I processor fails self-test, it will crash with a light code #x89. The Explorer II processor reports individual selftest numbers as they are run, with the failing test number left on the lights in case of failure. (See the light codes table.)

System Test - if the processor passes its self-test, it proceeds to the next phase, System Test and Boot.

1. First the processor determines whether it is to be the System Test & Boot Master (STBM) or if some other processor will be. In a single processor system, the Explorer will of course be the STBM. In a multi-processor environment, the processor in the lowest numbered slot that is an STBM candidate and has passed

self-test (within 10 seconds) is the STBM.

If the Explorer processor is not the STBM, it enters Secondary mode and waits for the STBM to post an event to awaken it. It then performs secondary booting operations.

2. Assuming the Explorer is the STBM, it performs the following actions:

NOTE

All searches for resources are performed by starting with slot 0 and looking at boards in successively higher numbered slots until slot 15 has been checked. For each slot, the STBM first checks for a value of #xC3 in the ID character of the boards Configuration ROM (see paragraph on the Configuration ROM). This indicates the board has a valid Configuration ROM. Next, check that the CRC in the configuration ROM is correct. Then the Resource Type field in the Configuration ROM of each board is examined to see if the board contains the desired resource.

a. Search for NVRAM. If the NVRAM bit of the Resource Type field is set, then the board contains NVRAM. The Explorer checks the CRC on the contents of the NVRAM (Explorer I does not) and the format generation number to verify it contains a value of #x01. Upon finding a valid NVRAM, the STBM extracts pointers to a Monitor, Keyboard, and Load Source for later use. See the paragraph on NVRAM for format details.

b. Find a monitor. If a valid NVRAM was found, then the monitor slot and unit numbers from NVRAM are used and the monitor is validated by first checking that the monitor bit in the Resource Type field is set, and then by issuing the Initialize- Monitor device driver call. If this completes successfully the monitor has been found. If there was no valid NVRAM, or if the Initialize-monitor call failed, then the processor searches for a board with the monitor bit on in the Resource Type field of the configuration ROM. If it finds one it calls the Initialize-monitor function of the device driver

on that board. If the call completes successfully, the monitor has been found. If no monitor can be found, the processor attempts to perform a default boot.

c. Display the message "Slot S TESTING SYSTEM", where S is the slot number of the processor board, on the monitor.

d. Find a memory board. The processor searches for a board with the memory bit set in the Resource Type field. When it finds one, it validates it by running the Interface Diagnostic, located in the ROM on that board. The diagnostic is run with messages disabled. If it passes, that memory board is used. Otherwise, the processor searches for another memory board. It is assumed that every memory board will have at least two megabyte of memory. If no valid memory board can be found, the message "ERROR: NO GOOD MEMORY FOUND" (Explorer I = "ERROR: 00000004") is displayed on the monitor, and the processor crashes with a light code (Explorer I = #x8A; Explorer II = #x74)

e. If a memory board was found, the processor then performs Chassis Testing where it performs the following actions on each successive board in the system, starting with slot 0 and ending with slot 15. If any test fails, the rest of the tests for that board are skipped.

- Tests whether a board is present in the slot (if no NuBus timeout is received when reading the configuration ROM, then a board is present). If the slot is empty, go to next slot.

- Displays "Slot S" on the monitor, where S is the slot number of the board under test.

- Tests whether the Configuration ROM contents are valid: ID character = #xC3 and CRC verifies. If not, display "ROM" ("TESTS FAILED" displayed later).

- If the ROM flags in the Configuration ROM indicate that the board performs a self-test, then wait up to 20 seconds for the self-test to complete. The board will reset a bit in the onboard Flag Register (see paragraph on Configuration ROM) when its self-test is complete. Check the Flag Register for a self-

test failure, and if one occurred, display "SELF" ("TESTS FAILED" displayed later).

- If the ROM flags in the Configuration ROM indicate that the board participates in NuBus tests, command it to do so and check the results. If a failure is detected, display "NUBUS" ("TESTS FAILED" displayed later).

- If the Configuration ROM Diagnostic Offset field is not = #xFFFFFF, execute the boards Interface Diagnostic.

- If all tests for a slot have passed, then turn off the slot Test LED in the Configuration Register (see paragraph on the Configuration ROM), and display "passed", otherwise display "TESTS FAILED".

f. Find a keyboard. If a valid NVRAM was found, then the keyboard slot and unit number from NVRAM is used and the keyboard is validated by first checking that the keyboard bit in the Resource Type field is set, and then by issuing the Initialize- keyboard device driver call. If this completes successfully the keyboard has been found. If there was no valid NVRAM, or if the Initialize-keyboard call failed, then the processor first attempts to initialize a keyboard at the same slot and unit as the system monitor currently in use (not supported by Explorer I). If that also fails then it searches for a board with the keyboard bit on in the Resource Type field of the configuration ROM. If it finds one it calls the Initialize-keyboard function of the device driver on that board. If the call completes successfully, the keyboard has been found.

If no keyboard can be found, the processor performs a default load using the boot device slot and unit from NVRAM. If there was no valid NVRAM, the processor searches for a boot device by first searching for a slot that has the boot source bit (Explorer I: and not the LAN bit) in the Resource Type field set, and then using the lowest numbered unit at that slot. The microcode specified as default in the partition table are loaded.

Initial Menu - if a keyboard was found, the processor sounds a tone and displays the Initial Menu: "D=Default load, M=Menu load, R=Retest, E=Extended tests :" If no key is pressed within approximately 15 seconds and no boards have failed Chassis Test, the processor attempts a default load of the MCR partition marked as default on whichever load source is determined to be the default load source. Pressing "D" or "RETURN" also results in a default load.

1. Pressing "R" causes the Chassis Testing phase, described above, to be repeated.

2. Pressing "E" causes the Chassis Testing phase to be repeated, but each Interface Diagnostic is run in "extended mode". In extended mode, an interface diagnostic will display the board identifier, the part number, the name of each test it runs on the screen, and an indication of whether each test passed or failed. Additional testing may also be performed. For example, the memory board diagnostic tests all of memory in extended mode, but not in normal mode.

3. Pressing "M" causes the processor to go into the menu boot sequence, described below.

4. At this point there are a number of "hidden options" in addition to those already mentioned. They are intended for use by expert users such as system managers and maintenance personnel.

a. Pressing "S" tells the processor that you want to do a default boot, but you want to use a boot unit other than the default unit. The processor will then present the device selection menu (described below), but will perform the boot as soon as you have selected the device.

b. Pressing "G" tells the processor that you want to load GDOS, the diagnostic operating system. The processor will then present the device selection menu (described below) but will boot GDOS as soon as you have selected the device.

c. Pressing "N" means that you want to type in the names of microcode and load bands to be booted. The processor will then prompt you for each of these names. You must type these names exactly as they are displayed by print-disk-label; the processor will not convert lower case to upper case automatically. The shift key or caps lock key will give you upper case. The rubout key will let you correct mistakes. Once you have typed in a name, the return key indicates you are

ready to proceed. When you have entered both names, the processor presents the device selection menu (described below). As soon as you select a device, the processor will boot the microcode and load band you typed in.

d. Pressing "F" (not supported on Explorer I) tells the processor that you want to load FDOS, the Factory version of the diagnostic operating system. The processor will then present the device selection menu (described below) but will boot FDOS as soon as you have selected the device.

e. Pressing "!" (not supported on Explorer I) causes the processor to enter a Debug utility menu.

Default Boot. After D is selected for Default Boot, the processor attempts to load from either the default load source specified in NVRAM or the first unit found by searching all slots. A "waiting" message is displayed until the default load source is ready, then the STBM interprets the Device Driver in the load source interface board configuration ROM to load the first MCR partition with the default bit set. Under the Release 3.0 the MCR named PRIM is used for an Explorer I, BOOT for an Explorer II.

Both BOOT and PRIM are primitives and the term "primitive" refers to both. The "primitive" has three primary functions which are:

1. The "primitive" co-ordinates the booting of a machine with multiple processors. Since the Explorer does not currently have multiple processors, this is not done in the "primitive" even though it has a skeleton design for such a function.

2. The "primitive" performs slave device downloading. Downloading can be thought of as patching the ROM on a controller board such as a disk controller board. If there is a software bug in the controller board ROM, it can be fixed by loading new software into a RAM area on the controller board and executing out of the RAM area rather than the ROM area.

3. The "primitive" loads the Lisp MCR code into the processor.

To do all of the above, the "primitive" uses a special type of partition called a Configuration partition. There may be several configuration partitions on the default disk. The default configuration partition is used by the "primitive".

A configuration partition has several entries in it, which are the name of the Lisp MCR, the name of the Lisp Load band, the name of the download software partition, and other information reflecting the machines configuration. If the default configuration reflects an erroneous machine configuration an error will occur. Later in this section there will be a description of the configuration partition and the algorithm used by the "primitive".

It should be noted that on an Explorer I PRIM and BOOT are identical but separate software partitions. On the Explorer II the single partition BOOT takes on both roles as "primitive" and menuboot. Both Menuboot and the "primitive" assume 2MB or greater memory boards.

Menu Boot - If you entered an "M" at the Initial Menu the processor then presents the device selection menu (the header is "AVAILABLE LOAD DEVICES"). This menu lists the available load devices and asks you to select one. After you select a device, the processor then loads a microcode band called "BOOT" from the device you selected. This is Menuboot, which then takes over the remainder of the boot process. It should be noted that this is the point where the processor leaves ROM and enters code (Menuboot) that has been downloaded into WCS. This is important because if for some reason Menuboot does not exist on the device you selected (or if it has been wiped out), this step will not work. If Menuboot is non-existent, the message "MICROLOAD NOT FOUND" will be displayed. If Menuboot has been wiped out, the message "BAD MICROLOAD FORMAT" will be displayed. These messages can also occur on any other microload attempt.

The first thing Menuboot does is to present the menu "L=LISP load, M=Multi-unit load, D=Diagnostic load, P=Print device label,C=Configuration Boot"

1. To do a Configuration Boot you enter an C or simply press RETURN since Configuration Boot is the default. After a C is pressed a menu of load devices appear. The operator is to select the desired device. After the device is selected, a list of configuration partitions appear on the screen. The operator then selects which configuration partition is to be used in the booting process.

2. To do a Lisp load enter "L" . The processor presents a

menu of available load bands on the device previously selected. Asterisks may appear by some of the load bands. The asterisks should be ignored and the user should select the desired load band.

Once you have selected a load band, you will be presented with a menu of available microcode bands. Again, asterisks may appear by some microcode bands. The asterisk may be ignored as above. The microcode that is preferred by the load band you previously selected will be named in a header message above the microcode menu. The preferred microcode is simply the one that the load band was saved with. You can generally use microcodes other than the preferred one, but the system will have to load the error table over the network. Once you have selected a microcode partition, the processor will load that microcode and pass it the name of the load band to be used.

3. If you select "M" the processor will present you with the the device selection menu before each partition menu. This allows you to select the system microcode and and system load band from different devices.

4. If you select "D" the processor will present you with the device selection menu (described above). Once you select a device the processor will display a menu of available diagnostic microloads for that device. Selecting one will cause that microcode to be loaded and executed.

5. If you select "P" the processor will present you with the device selection menu again (see above). When you select a device the processor displays the list of partitions on that device, similer to what is displayed by a (print-disk-label) form in LISP.

2.4 CONFIGURATION ROM

A machine based on the NuBus architecture has up to 16 NuBus slots. The Explorer has seven. Each slot may contain one board. Each board contains a configuration ROM. The configuration ROM is located at the highest NuBus physical addresses allocated for each slot (i.e. the ROM ends at address FsFFFFFC, where s is the slot number of the board) with data stored one byte per NuBus word.

The Configuration ROM fields that are of concern to the booting

process are as follows:

Resource Type field - This is a one byte field, located at address FSFFFF00. Each bit that is set to one indicates a resource that the board contains.

- Bit 0 Memory - board contains memory that may be used during booting. This memory will start at FS000000 and be at least one contiguous megabyte long.
- Bit 1 Boot source - board is a controller (e.g. disk, tape, LAN) that has a unit(s) that may be used as a load device. This board contains a load device driver.
- Bit 2 LAN - board contains a Local Area Network (LAN) controller that may provide a load device via the network This board may contain a load device driver.
- Bit 3 Monitor - board has a unit that can be used as the system monitor during the boot process. This board contains a monitor device driver.
- Bit 4 Bootable processor - board is capable of performing the standardized functions of a "secondary" processor to an STBM during system booting operations.
- Bit 5 Keyboard - board has a unit that can be used as the system keyboard during the boot process. This board contains a keyboard device driver.
- Bit 6 NVRAM - board contains non-volatile RAM that may contain system test and boot default parameters. If this bit is set the three byte board relative offset to the NVRAM is stored in locations FSFFFEF4 - FSFFFEFC (one byte per word) and the log 2 of the NVRAM size is stored in location FSFFFEF0.

Identification character - This is a one byte field, located at address FSFFFF04. If it contains the value #xC3, then the configuration ROM contains valid data, otherwise it does not. This provides a way for "foreign" boards to exist in the system without confusing the STBM.

ROM Flags - a one byte field at location FSFFFF10 which contains the following flags (and several others which do not concern us here):

- Bit 0 A one indicates the board does self-test.
- Bit 1 A one indicates the board will participate in NuBus tests.
- Bit 2 A one indicates the board is capable of being an STBM.

Flag Register Offset - a three byte field at locations FSFFFF14 - FSFFFF1C (one byte per word) containing the board relative offset to the Flag Register. The Flag Register is a one byte field containing the following flags:

Bit 0	A one indicates self-test is still in progress.
Bit 1	A one indicates the board failed self-test.
Bit 2	A one indicates that a peripheral or subsystem controlled by this board failed test.

Diagnostic Offset - a three byte field at locations FSFFFF20 - FSFFFF28 containing the board relative offset to the Diagnostic Engine code that makes up the interface diagnostic. A value of >FFFFFF indicates there is no interface diagnostic.

Device Driver Offset - a three byte field at locations FSFFFF2C - FSFFFF34 containing the board relative offset to the Diagnostic Engine code making up the boards device driver(s). There must be one device driver for each of the following bits that are set in the Resource Type field: boot source, monitor, keyboard.

Configuration Register Offset - a three byte field at locations FSFFFF38 - FSFFFF40 containing the board relative offset to the Configuration Register, which is a 8 bit field which contains (among other information) the following:

Bit 0	Reset - writing a one resets the board
Bit 1	NuBus Master Enable - a one enables the board to read and write via the NuBus (0 disables the boards Nubus interface)
Bit 2	Test LED - A one turns on the red LED on the board. A zero turns it off.
Bit 3	Test - when set to one during system testing requests the board to perform its NuBus Test.

CRC Signature - A 2 byte field at locations FSFFFFB8 - FSFFFFBC that contains the CRC value computed over the boards ROM. The ROM size is stored in location FSFFFFB4.

2.5 THE STBM AND DEVICE INDEPENDENCE

One of the goals of the STBM is to allow the ROM code on each board to be independent of the other boards in the system. For example, it must be possible to replace the disk controller board with one that provides a different interface to the processor, but not to have to change the processors boot ROMs. This is accomplished by having a boot source device driver (DDR) in the ROM on the disk controller. The DDR provides the processor with a generic interface to the device. The DDR is written in a

processor independent, interpreted language, called Diagnostic Engine code, and is executed by the processor. Since it is interpreted, the processor must have a Diagnostic Engine Interpreter in its ROM. Being processor independent means the disk controllers ROM does not have to change if the processor is replaced with one of a different type. It is called Diagnostic Engine code because the scheme was originally developed to allow processor independent diagnostics to be written. Most boards in the system have an Interface Diagnostic written in Diagnostic Engine code in their Configuration ROM.

There are three type of DDRs: load source, monitor, and keyboard. They provide the processor with a generic interface to the three peripheral resources it requires to perform system testing and load. The monitor and keyboard allow the processor to communicate with the operator, and the load source provides a source from which to load code. Load source interfaces currently exist for disk, tape, and LAN, and the processor-to-boot-source interface is generic enough to support virtually any kind of device that is capable of loading code. DDRs are intended to be used for bootstrap loading, not for normal system use.

2.6 NVRAM FORMAT

The information in NVRAM is accessed as though it were stored one byte per word. The first part of NVRAM contains system default configuration information, which is accessed during the boot process:

```
base addr + #x00 = STBM Monitor unit number LSB byte      Binary
base addr + #x04 = STBM Monitor unit number MID byte      Binary
base addr + #x08 = STBM Monitor unit number MSB byte      Binary
base addr + #x0C = STBM Monitor slot number (FF = none)   Binary

base addr + #x10 = STBM Keyboard unit number LSB byte     Binary
base addr + #x14 = STBM Keyboard unit number MID byte     Binary
base addr + #x18 = STBM Keyboard unit number MSB byte     Binary
base addr + #x1C = STBM Keyboard slot number (FF = none)  Binary

base addr + #x20 = Boot source unit number LSB byte       Binary
base addr + #x24 = Boot source unit number MID byte       Binary
base addr + #x28 = Boot source unit number MSB byte       Binary
base addr + #x2C = Boot source slot number (FF = none)    Binary

base addr + #x30 = NVRAM format generation number.
                   Equal 01 for all NuGeneration devices  Binary

base addr + #x34 = NVRAM format superset revision
                   number.                                Binary

base addr + #x38 = NVRAM CRC LSB byte                     Binary
```

base addr + #x3C = NVRAM CRC.MSB byte Binary

The NVRAM contains additional information in a specific generic format so that it can be accessed by any type of processor. The format is described in detail in the NuBus System Architecture Specification.

2.7 ERROR CODES AND MESSAGES

For the most common error conditions, textual error messages are displayed. However, in some cases, the STBM ROM code and Menuboot will display hexadecimal error codes.

2.7.1 STBM ROMS - Error Codes.

ERROR: 00000002 - Load device offline or not responding.
The device is powered down or is not connected.

ERROR: 00000003 - Load device error.
The load device experienced an unrecoverable error.

ERROR: 00000004 - Processor could not find a memory board that passed tests. The processor checks the following when looking for a memory board: Look for the value #xC3 in Configuration ROM ID character.
Look for memory bit set in Configuration ROM Resource Type field.
Make sure CRC in configuration ROM is correct. Run the Interface Diagnostic in the board's ROM and check that it had ROM and check that it had no failures.

DEVICE ERROR: 00000005 - Unexpected NUBUS error.
The processor was executing diagnostic engine code in a device driver in the NuBus Peripheral Interface (NUPI) or SIB ROM when an unexpected NUBUS error occurred.

DEVICE ERROR: 0000006 - Command timeout.
The NUPI device driver issued a NUPI command block and the NUPI did not set the complete bit in the status field. The minimum timeout value is 10 seconds. If the disk label is messed up, it could possibly cause this problem. The NUPI could also be faulty. If the NUPI considers a command block to be invalid, it will exhibit this failure mode. The last NUPI

command block that the NUPI device driver issued is located at location FS00C000, where S is the slot number of the memory board in the lowest numbered slot.

DEVICE ERROR: 00000009 - Network down.
The Ethernet is disconnected, shorted, or open.

DEVICE ERROR: 0000000A - Invalid unit number for the load device.

DEVICE ERROR: 0000000B - Ethernet board failed to initialize.

DEVICE ERROR: 0000001D - Bad DEI instruction header.
A board was found with a valid configuration ROM, but the Diagnostic Engine code that the Diagnostic offset or Device Driver offset in the configuration ROM points to has an invalid header. This possibly means that the ROM is bad.

DEVICE ERROR: 00000011 - Invalid DEI request.
The ROM on the board is good, but a request was made that could not be handled by that board. (e.g. a boot request was given to the monitor). This probably means that the contents of NVRAM are invalid. Try doing a menu boot, specifying the boot unit. Once the system is booted type in (si:setup-nvram) to the LISP Listener.

DEVICE ERROR: 00000012 - Diagnostic Engine code instruction space (ispace) problems.
The processor found an invalid instruction when trying to execute Diagnostic Engine code out of the ROM on one of the boards. This could happen when executing a diagnostic or a device driver. This possibly means the ROM is bad.

DEVICE ERROR: 00000013 - Diagnostic Engine code data space (dspace) problems.
The processor found one of the following problems when trying to execute Diagnostic Engine code out of the ROM on one of the boards: stack overflow, stack underflow, or dspace variable out of range. This could happen when executing a diagnostic or a device driver. This could be due to a bug in the code being executed, or the ROM could be bad.

DEVICE ERROR: 60000000 and above - NUPI command status.
These are errors that the NUPI device driver

passes back from the status field of the NUPI command block. See the NUPI Hardware Specification.

2.7.2 Menu Boot and Primitive - Error Codes and Error Messages.

ERROR: 00000014 - Device access error. The NUPI returned bad status.

ERROR: 00000015 - Invalid label. The first word of block 0 did not contain "LABL".

ERROR: 00000016 - Invalid partition table. The first word of the partition table did not contain "PRTN".

ERROR: 00000017 - No available microloads. There were no Explorer microcode partitions in the partition table.

2.7.2.1 Menuboot and Primitive Error Messages.

Message	Meaning
Warning: No Microcode Partitions on Device	Device selected in a Lisp load or device specified in a configuration partition had no microcode.
Warning: No Configuration Partition on Device	Device selected in a Configuration Boot had no configuration partition on it.
No Default Configuration Partition	On default boot, the "primitive" could not locate a default configuration partition.
Unable to Read Device	An error occurred when trying to read the load device.
Invalid Slot or Unit Number in the Configuration Partition.	The portion of the configuration partition containing the disk slot and unit number for the Lisp MCR is invalid.
Bad Load Partition or Load Device	The entry portion of the configuration partition containing the information about the load band has invalid information in it.
Currently Executing CPU is not in Configuration	A configuration partition was used which did not have a valid entry for the Explorer processor board.
Cannot Download Device	When trying to download a device, one of four problems occurred. First, the part number in the entry field may not match; second, there is a problem with the disk slot or unit number specified; third, there is a problem in matching the CPU type to the board type in the configuration partition; and last, there is no match on the download partition name.
Unable to Read the	This means the data in the

Partition Table	partition table is not set up correctly. To be set up correctly the following must be true: There must be a default entry named LABEL with partition type of Volume Label and CPU type of generic. There also must be a default entry named PTBL with partition type of Partition Table and CPU type of generic.

2.8 LIGHT CODE TABLE

In some cases the processor cannot proceed and cannot display a message. In these cases the processor will crash and display a code in the amber colored light-emitting diodes (LEDs) located on the processor board. These can be viewed by opening the front door on the system unit and looking through the slot on the interlock door. The lights are read as an eight bit binary number (seven bits for Explorer II) with the lowest amber LED as the least significant bit. The codes shown below are hexadecimal.

2.8.1 Explorer I STBM LED Codes.

Physical locations of LEDs:

(H)(6)(5)(4)!(3)(2)(1)(0)! (Fault)

H	= Halt	amber
6:0	= status indicators	amber
Fault	= Fault indicator	red

81 = Power failure. The processor took the power failure hardware trap.

82 = The processor took the control store parity error trap. This probably means that the processor's WCS is faulty.

83-87 = Should never occur. If they do the processor is probably faulty.

88 = The processor received an unexpected NuBus error.

89 = Processor failed self test.

8A = No memory. If the processor can find a monitor it will also display ERROR: 00000004 as described above.

8B = No boot device. This crash will only occur if the

processor cannot find a boot device and can not find a monitor on which to display a message.

8C = Microload problems. This will only occur if the processor cannot find a monitor on which to display the message "BAD MICROLOAD FORMAT"

8D = DEI PROBLEMS. This will only occur if the processor cannot find a monitor on which to display the device errors 10 - 13 described above.

8E = Monitor device driver problems. The processor got a non-zero completion code on a call to the monitor device driver.

2.8.2 Explorer II STBM LED Codes.
Physical locations of LEDs:

(H)(B)(L)(R)|(F)(6)(5)(4)|(3)(2)(1)(0)| (Fault)

H	= Halt	amber
B	= Busy	amber
L	= cache hit left	amber
R	= cache hit right	amber
F	= cache filling	amber
6:0	= status indicators	amber
Fault	= Fault indicator	red

with Fault LED on:

selftests:

7F = Processor unable to load code from EPROM

01-05 = Part 1 self-tests (Kernel tests)

06 = Passed kernel selftests, attempting to load remainder of tests and STBM (Part 2)

07-39 = Part 2 Lisp Chip and processor board selftests

3A-3C = Floating Point Board tests

crash codes:

71 = No online devices from which to download

72 = Bad microcode format found during attempted download

73 = Device error during attempted download

74 = No good system memory found

75 = NuBus error during download from Nubus memory to internal memories

76 = MCR partition requires floating point board which is not present

Fault LED off:

On = STBM arbitration phase, looking at slot n

1n = NVRAM search phase, looking at slot n

2n = Monitor search phase, looking at slot n

3n = Memory search phase, looking at slot n

4x = STBM testing chassis slot

41 = ROM test (C3, format version, CRC)
42 = Selftest
44 = NuBus test
48 = Interface diagnostic

5n = Keyboard search phase, looking at slot n

60 = At top level STBM menu

61 = Attempting default boot

62 = Building device menu

63 = Waiting for load device to come ready

64 = Reading partition from load source

65 = Processing MCR sections (except last)

66 = Loading WCS, PDL to A/M, and enter new code

70 = Waiting for first Secondary event

71 = Processing first Secondary event

72 = Waiting for second Secondary event

73 = Booting quietly

78 = Waiting RAM download (P3 mode 5)

2.9 MICROLOADS

The writable control store and other internal memories of the Explorer processor are loaded from a microload. A microload is read by the boot PROM from a mass storage device, interpreted, and the internal memories are loaded.

The Explorer microassembler produces an output file in the "MCR" format. The file name will be "xxx.mcr" where "xxx.lisp" is name of the source file. The "load-mcr-file" function converts the "mcr" file to the microload format and installs it as an "MCR" partition on a disk. This compact representation can be loaded by the Device Driver on the disk controller board when directed to do so by the processors bootstrap PROM. This format provides for the loading of I-mem, A/M memories, D-mem, and main memory. On the Explorer II it also provides load data and initialization instructions for additional onboard IO space structures. The Explorer II MCR format is specified in Appendix A.

2.10 INTERFACE BETWEEN BOOT PROM AND MICROLOADS

When a microload is loaded by the boot ROMs, certain information is passed to the microload in dedicated A memory locations. The following A memory locations are used for the purpose of passing parameters between the boot ROM and microloads:

variable name	A memory location
A-BOOT-COMMAND-BLOCK	#x3F8
A-BOOT-LOD-DEVICE	#x3F9
A-BOOT-MEMORY	#X3FA
A-BOOT-MONITOR	#x3FB
A-BOOT-KEYBOARD	#x3FC
A-BOOT-DEVICE	#x3FD
A-BOOT-MCR-NAME	#x3FE
A-BOOT-LOD-NAME	#x3FF

The boot ROM sets up the following locations as parameters passed to a microload that is loaded:

A-BOOT-MEMORY is set to FsOOOOOO where s is the slot number of either the first memory board found or (if in secondary mode) the memory specified for this processor in the Command Block received from the STBM Primitive.

A-BOOT-MONITOR is set to designate the system monitor. The slot number is in the most significant byte and the unit number is in the three least significant bytes.

A-BOOT-KEYBOARD is set to designate the system keyboard. The format is the same as for a-boot-monitor.

A-BOOT-DEVICE is set to designate the boot device. The format is the same as for a-boot-monitor.

A-BOOT-MCR-NAME contains the name of the microload in ASCII little endian format.

A-BOOT-LOD-NAME contains the name of the load band. The format is ASCII little endian. If a load band was not selected, this word will contain a value of binary zero.

The boot ROM will only set up A-BOOT-COMMAND-BLOCK if the Explorer processor is being booted as a secondary processor, in which case it shall have the NuBus address of the Command Block.

If Menuboot is run, it will store the system load name in A-BOOT-LOD-NAME in ASCII little endian format. If the system load is on a different unit than the microload, then A-BOOT-LOD-DEVICE will be set to that unit. Otherwise, it will contain the same value as A-BOOT-DEVICE. If Menuboot is not run, A-BOOT-LOD-DEVICE will contain a value of binary zero. The system microcode must check this variable to determine where to get the system load.

In order for Menuboot (or other WCS microcode programs) to request the load of another microload into WCS, there are special processor dependent mechanisms that must be used to reenter ROM. The following A memory locations must be set up for all processors: A-BOOT-MEMORY, A-BOOT-MONITOR, A-BOOT-KEYBOARD, A-BOOT-DEVICE, A-BOOT-MCR-NAME. For the Explorer I the requesting code must turn off the PROM-disable and Bus-Error-Trap-Enable bits in the Machine Control Register (MCR) and then jump to PROM location #x1E. For the Explorer II the requesting code must enable refresh, IROM, and memory cycles in the Machine Control Register (MCR), set the highest M-memory register to #xC3, and then jump to IROM address at #xFF.

The Explorer II ROM code also supplies information to the downloaded code about the type of load which has just occured.

PDL location 0 is loaded with a value indicating the boot type:

0 = STBM, default mode
1 = STBM, non-default
2 = Secondary, default
3 = Secondary, non-default
4 = Special RAM load mode

2.11 CONFIGURATION PARTITION, PRIM ALGORITHM, AND DOWNLOADING

A Configuration Partition is a 17 block disk partition which supplies boot time parameters for a system. These include information such as which software shall be loaded by each downloadable processor and controller and values which allocate ownership of system resources amongst processors. The partition is divided into two parts: the partition header and the configuration modules.

1. The partition header is one block long and is divided into two parts which are both a half block long. The first half of the headers contains the following:

   a. At offset 0 the four characters "CNFG" are found. These characters are used to validate the fact that this is a configuration partition.

   b. The next two characters are used as a CRC value. The "primitive" does not use this.

   c. The next four characters are used as the generation and revision values. The "primitive" does not use these values.

   d. The remaining portion of the first half of the configuration partition is available for comment space.

2. The entries in the second half of the header are called pointers. Pointers are 16 bytes long. The following byte values are relative to offset X200 of the header block. The pointer values are:

   a. Bytes 0,1: A pointer to a configuration module. The pointer is relative to the start of the configuration partition. If the pointer value is 0, this implies that this entry is empty.

   b. Bytes 2,3: Length in blocks of the configuration partition.

   c. Bytes 4 - 7: A boot timeout. This is primarily for multiprocessing systems and is not used by

the Explorer.

d. Bytes 8 - B: The count of configuration entries. This value will be discussed in the configuration module portion of this section.

e. Bytes C,D: A CRC value for the configuration module associated with this entry.

f. Bytes E,F: The board type. If the board associated with this entry is a processor, then the value in this position is equal to the processor value found in the processor configuration ROM at location FSFFFF9C. For a controller, this value is equal to the disk partition type assigned to the download software for a particular controller.

g. Bytes 10,11: Implies via a 1 in a bit position, what slot the board may be in. For instance, an Explorer I board can only be in slot six so bit six would be set to a one for the Explorer I board. Some boards can be in any slot so this value would be all ones for those boards. Notice a configuration partition assumes a sixteen slot chassis.

h. Bytes 12,13: Indicates board type. The value 1 indicates that this entry expected a processor board. A 2 indicates that this entry expected a downloadable controller. No other values have meaning.

i. Bytes 14 - 1F: Reserved.

3. The next entries in the configuration partition are called configuration modules. They can be accessed by another processor when they are in main memory. This feature is not used by the Explorer processor. However, much of the information in the configuration module is used by the Explorer "primitive". The following byte values are relative to the beginning of the configuration module. The information in the configuration module is as follows:

   a. Bytes 0 - 3: The busy status flag. This flag is used when the configuration is in memory. The Explorer "primitive" does not use this field.

   b. Bytes 4 - 7: NuBus memory base address. This is not used by the Explorer "primitive".

   c. Bytes 8 - B: Monitor slot and unit. This value

is set up by the Explorer Primitive.

d. Bytes C - F: Keyboard slot and unit number. This value is set up by the Explorer "primitive".

e. Bytes 10 - 13: Disk slot and unit number where the Lisp MCR partition is to be found. The Explorer "primitive" expects this value to already be set up. If a value of all ones(-1) is found, a default disk is assumed. The assumed default disk is the disk from which the Explorer "primitive" was loaded. If bytes 12,13 of the pointer entry is a 2, then this entry is the disk address of a download software partition.

f. Bytes 14 - 17: A four-character ASCII name of either the Lisp MCR partition or the download software partition. Bytes 12,13 of the pointer entry pointing to this configuration module indicates which one of the two this is. Note, all four characters must be exactly as found on disk. A name with less than four characters must be blanked filled.

g. Bytes 18 - 1B: The disk slot and unit number of the configuration partition currently being used by the Explorer "primitive". This value is set up by the Explorer "primitive".

h. Bytes 1C - 1F: The four character ASCII name of the configuration partition being used by the Explorer "primitive". This is set up by the Explorer "primitive".

i. Bytes 20 - 23: The synchronization flag. This is for multiple processors and not used by the Explorer "primitive".

j. Bytes 24 - 43: The 32 character hardware identification value. This value is used only when downloading is required (bytes 12,13 of the pointer field = 2). This is a part number followed by a three character ID. This value must match exactly the information found in the configuration ROM on the board to be downloaded or no download occurs.

k. Bytes 44 - 63: A list of 32 byte ASCII entry. The number of entries in the list is in bytes 8 - B of the pointer entry for this configuration module. Note, these entries are predefined and are strictly formatted. A removal or addition of a blank within these ASCII entries will cause the

"primitive" to fail. For an Explorer the entries are as follows:

- Entry 0: The part number and id value as described above.
- Entry 1: ASCII value "Explorer Processor" or "Explorer II Processor". The Explorer primitive keys off the string "Explorer".
- Entry 2: "Slots owned: (ASCII slot numbers each separated by a space). This entry is not used by the Explorer primitive.
- Entry 3: "Load Slot :(Single digit ASCII slot number of the disk unit containing the Lisp load band)".
- Entry 4: "Load Unit :(Six digit ASCII unit number of the disk containing the Lisp load band)". Note: if either entry 3 or 4 is an asterisk, then a default disk is used to load the Lisp load band. The disk defaulted to is the same disk from which the primitive was loaded.
- Entry 5: "Load Name :(name of the Lisp load band)". If an asterisk is used, then the first Lisp load band with the default bit set in the disk partition table is used. The primitive searches for a partition of type Explorer or of type TI Lisp.

2.11.1 The Algorithm.

The following describes the algorithm in determining if a board in the system matches and entry in the configuration partition. If a match is made then either a download occurs or a CPU entry is pushed on the stack.

First, the primitive scans each NuBus slot searching for board in a NuBus slot begining with slot 0. If a board is found a search is made through the pointer entries in the configuration

partition to find a 1 bit in the corresponding bit position in bytes 10,11 of the pointer entry. If a match is not found, the search for a board in higher slot numbers continues. If a match is found, the primitive fetches bytes 12,13 of the matching pointer. If the board found is not the same type as specified in bytes 12,13 of the pointer the above search continues. The "Same Type" check fetches byte 12,13 of the pointer to determine if the pointer entry expected a controller board or a CPU in this slot. If the board in the slot was a controller and the pointer entry of the configuration partition expected a a controller board, the the controller type is checked. If the types compare then a download is performed assuming there is a download software partition for this board. If the board type was a CPU and the pointer expected a CPU, the the configuration module address for this CPU is pushed onto a stack for later use; this assumes the CPU types matched. If no match occurs, then the slot search continues. The algorithm is complete when all slots have been examined.

2.11.2 Downloading.

Downloading is a method to patch ROM code on a controller board by inserting new code into the RAM portion of the board and using the RAM resident code rather than the ROM resident code. Previous discussions indicated that the "primitive" performed the downloading. This is only partially correct. Once the primitive has determined that the a board is to be downloaded, the disk is searched for the software partition associated with that board. The software partition contains Diagnostic Engine Code which has been previously discussed in this section. The Diagnostic Engine code performs the actual downloading. The software partition found by the primitive contains the Diagnostic Engine code in the first portion and the ROM replacement code in the second portion of the partition.

Once the Diagnostic Engine Code partition has been loaded by the primitive, it is then interpreted by the primitive's DE code interpretor. At the completion of the DE code's execution, an error indicator is checked by the primitive. If an error occurred during downloading, an error message is generated. Otherwise the algorithm continues.

SECTION 3

Interrupts

3.1 INTRODUCTION

Hardware event signals (interrupts) are the lowest level of mechanisms in the Explorer virtual architecture. An interrupt, therefore, cannot rely on any higher level to perform its work. Hence, interrupts are serviced by the lowest level of the microcode implementation without reference to virtual memory, garbage collected memory, or macro instructions. (Virtual memory service for memory map handling is considered very low level and is permitted in some interrupt contexts.)

An interrupt is caused by an I/O device requesting service, another processor signalling an event, or by system bus and internal processor errors. In order to simplify interaction between interrupts and higher levels, interrupts are not automatically processed, but are polled at convenient times by higher levels of the implementation. When an interrupt is noticed by polling, control transfers to an appropriate handler for the interrupt.

Interrupts communicate with higher levels of the system via shared memory in the form of flags in internal processor memories and shared memory. The shared memory must not be garbage collected and must be wired (so no page fault can occur when accessing it) and fixed (so it cannot be moved by the garbage collector). In other words, neither the garbage collector nor the page fault handler can be invoked by the interrupt handler.

A check for pending interrupts is made at most virtual memory operations, especially instruction fetch. Interrupts are also checked at other times during internal processing (e.g. during disk paging waits). As a result, interrupt response time, while usually within a few microseconds, has no guaranteed maximum. Interrupt handlers must, therefore, handle the case that the response was too slow if it could cause problems.

Interrupts are the primary means for external events to signal the Lisp system. In most cases, interrupts set flags for higher level processing to notice or move data between the I/O device and an I/O buffer in wired memory. However, certain time-critical processing may be best performed as part of the interrupt handler.

3.2 INTERRUPTS ON THE EXPLORER PROCESSOR

The Explorer processor has hardware to ease the detection and processing of interrupts. Since the Explorer processor is NuBus based, most interrupt are events signaled over the system bus by writing a word or byte with the low order bit set into special locations in the control space of the Explorer processor. A map of the interrupt locations and the priority of each is shown in Table 3-1.

Table 3-1 Explorer Control Space

NuBus Address (Hex)	Interrupt Priority Level (Decimal)
FsE0003C	15 (Lowest)
FsE00038	14
FsE00034	13
FsE00030	12
FsE0002C	11
FsE00028	10
FsE00024	9
FsE00020	8
FsE0001C	7
FsE00018	6
FsE00014	5
FsE00010	4
FsE0000C	3
FsE00008	2 (Highest)
FsE00004	1 (Preemptive) Boot request
FsE00000	0 (Preemptive) Powerfail

Interrupt pending is a condition testable individually and in combination with the page fault and sequence break conditions for jump and abbreviated jump microinstructions. Microcode tests whether there is an interrupt pending by performing a conditional call to the interrupt service routine if the interrupt pending condition is true.

The interrupt service routine will process all interrupts before returning to the caller. Interrupts are, of course, processed from highest priority to lowest. Interrupt priority is linked to the location in the control space of the processor as shown above in Table 3-1. The highest priority level with an interrupt pending is indicated by a special field in the machine control register (MCR) of the Explorer, or the Pending Event register at

IO space address #x3A000000 on the Explorer II. On any level with several devices, all devices that could interrupt to that level must be polled. For each interrupt level, there is a list of device descriptor blocks. The device descriptor block is shown in Figure 3-1.

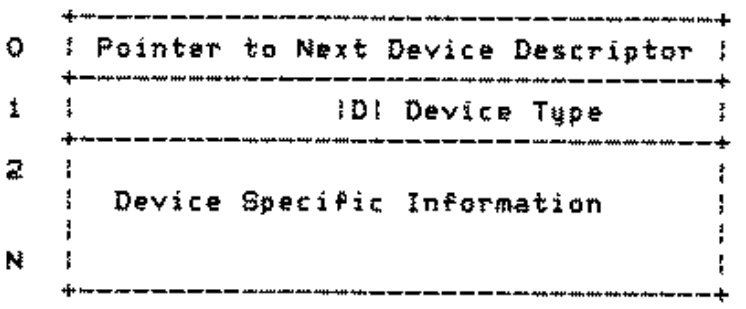


Figure 3-1 Device Descriptor Block

The lists are anchored by the interrupt vector table. The interrupt vector table is indexed by interrupt priority and contains a pointer to the interrupt descriptor block list for all the devices on that interrupt vector or priority. The value zero indicates the null link or empty list. The interrupt vector table is shown in Table 3-2.

Table 3-2 Interrupt Vectors

Interrupt Priority Level	
0	Powerfail (empty vector)
1	Boot request (empty vector)
2	Device descriptor block address
.	
.	
.	
15	Device descriptor block address

Once the highest priority interrupt level has been determined, the event request is cleared by writing a word with the low order bit set to zero into the word of the processor control space that corresponds to the interrupt level of the device. Interrupt levels 0 and 1 are dedicated to Powerfail and Boot request

respectively. These events are handled as aborts, i.e. the processor traps directly to processing of the event. If either of these events are detected as polled interrupt events then the processor failed to trap. The interrupt processor will cause the machine to halt.

For levels 2 to 15 the interrupt service routine uses the highest priority level with an interrupt pending to index into the interrupt vector table. This yields the list of device descriptor blocks for this interrupt level. The interrupt processor then traverses this list. For each block in this list the interrupt type is extracted from word 1 and the specific handler for this interrupt type is called. This interrupt type specific handler is responsible for determining if the device pertaining to this descriptor block has requested interrupt processing, and if so, performing that processing. When this interrupt handler returns, the next block in the list is examined, etc., until the end of the list is reached.

When the end of the list has been reached the interrupt pending condition is tested. If no interrupts are pending, the interrupt processing is complete and the interrupt service routine returns to its caller. If an interrupt is pending, the entire process is repeated.

SECTION 4

Device Handling

4.1 INTRODUCTION

The device handling features and conventions have been designed for simple but flexible operation, with few restrictions relating only to cooperation with system device operations, i.e. virtual memory on the disk. Each device is assigned a device type which is a small positive integer. Device type numbers are assigned at system build time. This section is aimed at describing the specific operation and the internal structures used by many of the devices in the system. First, the general scheme for handling I/O requests is presented, later sections detail specific devices.

4.2 DEVICE DECODING

Each device in the system that requires interrupt processing maintains a Device Descriptor Block (see section on interrupts). This descriptor block contains all the information that the system needs for processing requests and interrupts for this device. The Device Descriptor Block maintains information about the device state. A separate structure, the Request Block (RQB) is used to transfer request information. The first word in the Device Descriptor Block is used as a link word for the interrupt decoder and points to the next device descriptor on the same interrupt level. The second word is called the device information word. This word contains information needed to determine what type of processing is needed when an initiate I/O request is sent to the device and also what type of interrupt processing is required. The device type implicitly specifies the number of device specific words required. The details of some specific devices follow. The initiation and interrupt processors use the device specific portion of the block to maintain information pertaining to the device and the outstanding requests. Therefore, every device in the system that requires microcode handling must provide an entry point in the initiate dispatch table and an entry point in the interrupt handler dispatch table. The entries are placed in the tables according to device type.

To initiate an I/O request the %IO miscop (miscellaneous operation) is used. The parameters are the device descriptor and

a request descriptor. There are two basic forms for the request descriptor. The first form is a fixnum. In this case, the single word specifies the operation being requested. The second form is a request block. The request block is an array. The specifics of the request block are defined by the device handler. To initiate a request the device type is fetched from the Device Descriptor Block and the device initiation handler is called for this device type. The device initiation handler is responsible for servicing this request by either starting the device operation or placing the request in a queue for later processing.

To make the best use of the peripheral resources on the machine there is a mechanism to queue requests for I/O and continue processing. In any case, we would not want the processor to just sit idle during I/O requests, but rather run other processes that might be ready to run. To be able to do this some devices maintain a list of outstanding requests, and when one request has completed automatically start the next request. This requires that the interrupt handlers for a device must maintain the queue as part of its duties.

4.3 DEVICE-HANDLER DATA STRUCTURE ALLOCATION

Memory references made by interrupt handlers must be to static, wired virtual memory or to the physical address space. Device-specific control registers are generally accessed as NuBus physical memory which has no mapping in the virtual address space. Other interrupt-handling data structures, however, generally are mapped into the Lisp virtual address space so that both the microcode and Lisp can access them as normal virtual memory. Allocation of these data structures follows the following conventions:

- Device descriptor data structures as well as the interrupt servicing queue itself are allocated in the SYS:DEVICE-DESCRIPTOR-AREA. This is a special system-wired no-scavenge area in low virtual memory. The structures are allocated simply as segments of un-typed memory, hence are not viewed as Lisp objects; therefore, this area is not garbage collected or scavenged. Addresses within the data structures can only point to other data structures in the SYS:DEVICE-DESCRIPTOR-AREA or to objects in other static areas.

- Actual I/O buffers themselves are generally allocated as Lisp arrays in static areas, and are wired down before the I/O starts using paging primitives such as SYS:WIRE-ARRAY or SYS:WIRE-PAGE. Examples of current static areas are:

SYS: DISK-BUFFER-AREA	disk and tape RQBs and data buffers
SYS: SERIAL	serial and parallel port I/O Buffers
CHAOS: CHAOS-BUFFER-AREA	Ethernet packet buffers

The remainder of this section describes specific devices and their I/O Interrupt handlers.

4.4 DEVICE DETAILS

4.4.1 The NuBus Peripheral Interface Board.

The NUPI is used for interfacing to disk and tape devices. The NUPI receives operation requests by writing the address of a NUPI Command Block to a special address in the NUPI address space designated as the Command Register. The NUPI then processes this command asynchronously from the Explorer processor. When the request has been completed, the NUPI stores the status of the operation back into the status word of the request block and, if specified in the request block, posts an event to the Explorer processor to signal completion of a command. (The Explorer convention is that all requests to the NUPI post the completion event.) This event posting is fielded by the processor as an interrupt. The device interrupt handler is called to process the completion of the current request and initiate the next one if required.

Figure 4-1 shows the format of the Device Descriptor Block for the NUPI. The Link Word and Device Information Word are standard for all devices. The Control Space Word holds the NuBus address of the control space of the NUPI board. This is needed to correspond with the board. A NUPI may have several disk or tapes units under its control. Each disk or tape unit is connected to a formatter, which is a local controller. A formatter may have one or two devices connected to it. Each device, formatter and the NUPI itself is considered a unit, and all may simultaneously have a request in progress. The device handler maintains a request queue for each device. The request at the front of the queue is the one currently being processed. When a request comes to the front of the queue the Busy Bit in the Information Word (see Figure 4-2) is set to signify that the request is in progress. When the request is completed the Busy Bit is reset and the Done Bit is set. The request block is removed from the queue. If there are any other requests in the queue at this time, processing is started for them.

The Unit Busy Bitmap indicates which units have requests in process. This is used because when a request completes and an interrupt is signalled all devices with requests in process must be checked to find which ones have completed. The bit map allows polling of only those units for which processing is possible.

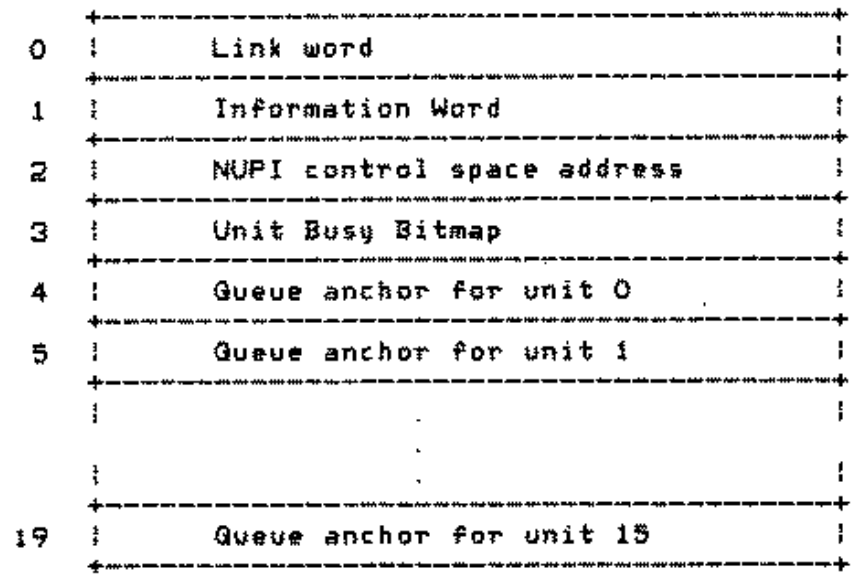


Figure 4-1 NUPI Device Descriptor Block

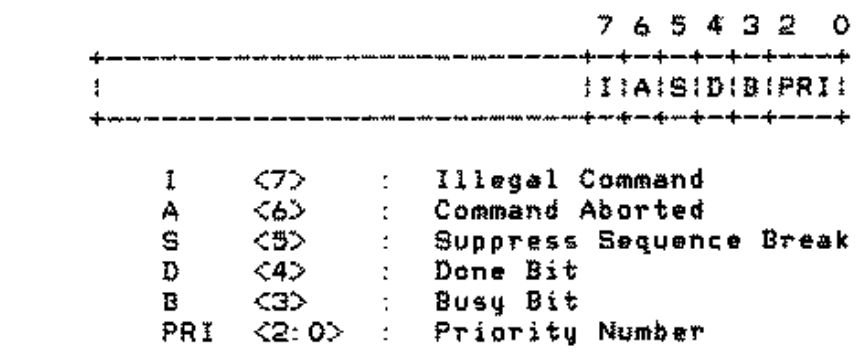


Figure 4-2 NUPI Device Descriptor Information Word

4.4.2 The Keyboard.

When a keyboard interrupt occurs, the keyboard character is copied from the hardware register into the keyboard buffer. The Lisp keyboard process notices that a character is available and

handles it.

4.4.3 CRT Vertical Retrace (Interrupt).

An interrupt occurs when the CRT begins the vertical retrace interval. At this time, the mouse is checked to see if it has moved or if any of the buttons have changed state. Mouse button changes go into the mouse buttons buffer. If the mouse has moved, it is undrawn in its old position and redrawn in its new position. The mouse process then notices either mouse motion or mouse buttons.

This interrupt also handles the timer on the Explorer I (the Explorer II has a microsecond timer on the processor). The short interval timer is read to determine the exact number of microseconds that have elapsed since the last TV vertical retrace interrupt. This value is then added to an internal microsecond timer and the short interval timer is reset.

4.5 SERIAL AND PARALLEL IO PORTS

The Explorer has one serial port for bidirectional asynchronous communication, and one Centronix (tm) standard unidirectional parallel port for high speed datatransmission. (Centronix is a registered trademark of Centronics Data Computer Corporation.) Each has microcode support and can be used with a standard stream interface.

4.5.1 RS232 Serial Port.

The Explorer serial port is based on a Zilog Z8530 Serial Communications Controller found on the System Interface Board (SIB). It can be configured to handle all asynchronous formats regardless of data size, number of stop bits, or parity requirements. It can also accomodate all synchronous formats including character, byte, and bit-oriented protocols.

There is both microcode and Lisp support for the serial port. The microcode handles receive and transmit interrupts for the chip. A stream interface to the serial port is provided to make it convenient to use. Although the chip supports synchronous modes, the software currently provides support for asynchronous modes only.

Microcode handling of serial port interrupts enables high speed transmission of characters. Characters to be sent out through the port are queued in a buffer. When the serial port signals a transmit ready interrupt, the microcode removes the next character from the buffer, if any, and sends it out. Similarly,

when the chip signals a receive character ready, the microcode will remove the character from the chip's internal register and store it into another buffer where it can be accessed from Lisp when convenient.

The microcode also has support for an XON/XOFF protocol. When enabled, this will allow external devices to control the transmission of data by the Explorer and will allow the Explorer to stop a device from sending data. When running in this mode, the receipt of an XOFF character will disable further transmission until an XON character is received. If the Explorer's serial input buffer comes within 10 characters of being full, it will automatically transmit an XOFF character to temporarily stop data transmission. Naturally, the external device must also support XON/XOFF protocol for this scheme to work correctly.

The serial port is programmed by writing data into the 14 write registers of the chip. Received characters and chip status are available in seven read registers. For details on the function of these registers see the Explorer System Interface manual. These registers are found within the NuBus address space of the SIB. It is therefore possible to program and use the chip directly from Lisp in polled mode, although this is not recommended.

The port is intended to be used in interrupt driven mode provided by the microcode and serial streams. Serial streams provide a convenient way to initialize the operating parameters of the chip and to send and receive characters. Operating parameters are normally specified in the call to SYS:MAKE-SERIAL-STREAM. Characters are transmitted with :TYO and received with :TYI. For more information on serial streams, see the Explorer Input/Output Reference manual.

4.5.2 Parallel Port.

Like the serial port, the parallel port is also found on the System Interface Board. The programming interface consists of two registers found within the NuBus address space of the SIB. The data register is an 8-bit write-only register. The command register is also 8 bits wide. Writing to the command register allows you to initialize the port, cause data strobe, and enable auto linefeeds or interrupts. Reading from the command register returns port status including busy, paper out, online, and fault.

There is microcode support for running the parallel port in interrupt mode. Data to be transmitted is placed into an output buffer. When the external device is ready to receive a character, an event is posted by the SIB to the processor. The microcode will remove the next character from the output buffer and send it out. If a large number of characters are queued in

the output buffer, data can be transmitted at very high speeds. A stream interface is provided to make the parallel port easy to use.

SECTION 5

The Virtual Memory System

5.1 VIRTUAL MEMORY

Virtual Memory is the simulation of a large fast primary memory by the use of a fast but smaller primary memory and a large but slow secondary memory. Blocks, called pages, are moved between primary and secondary memory according to a page management strategy.

A page management strategy that moves a page into primary memory when it is referenced but not present in primary memory (a page fault) is termed demand paging. Usually, a page being moved into primary memory displaces some other page. The choice of the page to remove is made by applying the page replacement policy. If the page chosen for replacement has been altered while in primary memory, it must be written to secondary memory before it can be replaced. A page in primary store that has been altered is called a dirty page.

Some pages are exempted from paging. These are termed wired pages. Wired pages are used for interrupt handler buffers because interrupts cannot take a page fault; for pages containing paging tables on which a page fault cannot be allowed; for pages involved in DMA transfers; and other pages containing critical data which must be accessed without a page fault or for which the performance penalty for taking a page fault is too great.

In the Explorer, semiconductor memory is used as primary memory and disk is used as secondary memory. Pages are moved into primary memory when referenced and not present -- demand paging. Every attempt is made to replace a page which is not dirty so that a write to secondary memory is not needed.

A page exception is said to occur when for some reason the virtual to physical address mapping could not be completed by the mapping hardware without microcode support. There are many reasons for this; only one of those reasons requires access to secondary storage. If a page is referenced and this reference cannot be completed without operations with secondary storage, then a page fault has occurred. This distinction is made so that page exception rates and page fault rates can be distinguished.

Virtual memory references are processed below everything except interrupts in the Explorer hierarchy (they are depended on by

everything except interrupts). A virtual memory reference is considered an atomic operation on the Explorer system; that is, the time spent waiting for a virtual memory reference to complete (including time spent waiting for disk paging activity) is not available for use by any other part of the system except interrupt processing. This policy greatly simplifies the implementation of various low-level Explorer subsystems.

5.2 PHYSICAL ADDRESSES

Memory is accessed by use of a physical address, a system-wide name for some storage. The Explorer is based on the NuBus, a 32-bit, high-speed bus. All NuBus addresses are byte addresses with words aligned so that the low order 2 bits are zero.

Explorer physical memory, monitor screen memory and I/O peripheral control registers all reside within the same 32-bit address space. Not all bus addresses will be accessible directly mapped in the Lisp virtual address space. This is true because the 25-bit Lisp virtual address is smaller than the 32-bit NuBus address space. Data can be read and written into unmapped physical addresses using special physical I/O functions from Lisp.

5.3 VIRTUAL ADDRESSES

An address in the Explorer system is the size of a pointer field, which is 25 bits. The virtual address is divided into a virtual page number and a page offset. The virtual page number is the high order 16 bits of the virtual address, and page offset is the low order 9 bits of the virtual address. Thus, each virtual page contains 512 words (or 2048 bytes) of storage.

The Explorer I and Explorer II both have memory mapping hardware based on a map page size of 256 word (or 1024 bytes). Therefore, there are two hardware map pages for each virtual page. When referring to these maps, the map page number is based on the high order 17 bits of the virtual address and will be called the virtual page map number. The page offset is based on the low order 8 bits and will be called the page map offset.

The Explorer has a simple memory map. I/O is not, by default, part of virtual memory, but instead is accessed by special physical I/O operations. The A-memory (the processor memory for data accessed by the microcode) has a dedicated virtual address, which is at the very top of the virtual address space, but consumes none of the physical address space.

In order to translate from a virtual to a physical address, the virtual page map number is looked up in the hardware page maps to produces a 22-bit page frame number. The page frame number is concatenated with the 10-bit page map offset to make a 32-bit physical address. This is used to address the primary memory over the system (or other) bus.

The map also produces other outputs for use by the processor: 2 access bits, 2 status bits, 6 meta bits and 2 garbage collector volatility bits. These outputs are used by the microcode to help manage page aging, storage allocation attributes (on a per-region basis), garbage collection, and other functions.

The Explorer microprocessor has a standard NuBus interface. In addition, The Explorer I microprocessor has a special bus to "local memory". This local memory also exists in the NuBus address space but the special high-speed dedicated bus to this memory reduces the NuBus traffic.

5.4 PHYSICAL MEMORY USE

The physical memory present in the machine can be divided into two categories according to its usage. A portion of memory is set aside for use by the paging system itself and by microcode and Lisp device handlers that perform physical addressing. Data in this space is said to reside in physical memory which is not accessible in the virtual memory address space. Such physical memory is termed the permanently wired pages. The rest of the local physical memory is used as a transient page area, i.e., the virtual memory system assigns the pages of the virtual memory to physical locations as a part of its management functions.

5.5 VIRTUAL MEMORY PARTITIONING

To avoid the need for a very large mapping memory, or an associative memory, both the Explorer I and Explorer II use two-level memory maps for the virtual-to-physical address translation. Several sections following this one describe the hardware maps in detail. This section, however, is meant to provide an overview and motivation for the map functions.

The first level map (called the Level-1 map on the Explorer I and the Address Space Map on the Explorer II) effectively partitions the virtual address space into segments of 8K words (16 pages). They both have 4096 entries and are indexed by the top 12 bits of the virtual address. Note that there are two such entries per 32-page address space quantum. The 32-page address space quantum is the unit in which address space is allocated to regions; that

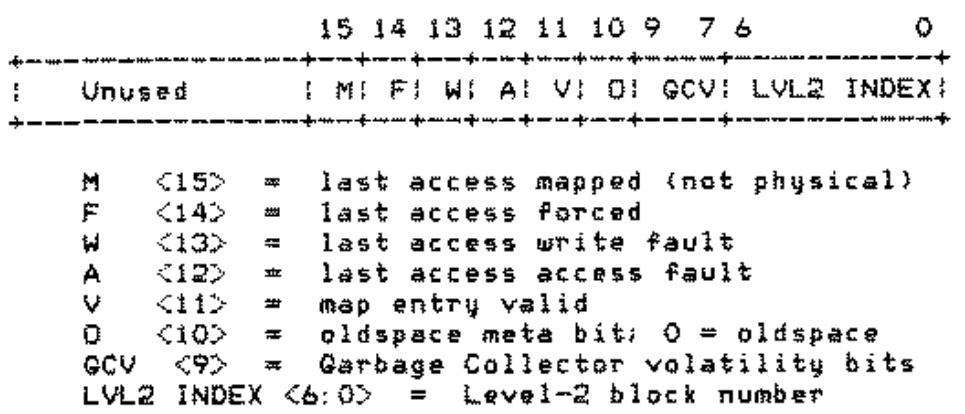
is, a region must be at least 1 address space quantum long, and must have a length evenly divisible by the address space quantum.

On both Explorer I and Explorer II, a number of second level maps are allocated to the first level maps. The number of second level maps differs; but in both cases it is much less than the amount required to map in the entire 25-bit address space. If the address is not currently being mapped by the mapping hardware, the microcode consults the Page Hash Table (PHT, described later) which describes all pages in physical memory.

In general terms, the first level maps can be thought of as a way of segmenting the address space into blocks of contiguous virtual pages with identical attributes in order to aid storage allocation and garbage collection functions. The second level maps can be thought of as a cache describing the most recently accessed pages. Except for special system pages, the storage-management and garbage-collection-specific information in the memory maps is initially set up from the REGION BITS associated with a virtual address. They may later be altered by the garbage collector.

5.6 EXPLORER I MAPPING HARDWARE

The Explorer I Level-1 Map (Figure 5-1) is 4096 words long and is indexed by the top 12 bits of the virtual address. The first level map, if valid, produces a 7-bit index into the second level map along with several status bits.



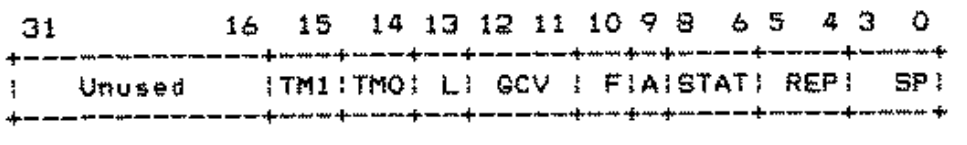
M <15> = last access mapped (not physical)
F <14> = last access forced
W <13> = last access write fault
A <12> = last access access fault
V <11> = map entry valid
O <10> = oldspace meta bit; 0 = oldspace
GCV <9> = Garbage Collector volatility bits
LVL2 INDEX <6:0> = Level-2 block number

Figure 5-1 Explorer I Level-1 Map (LVL1)

The second level map (Figure 5-2) consists of 128 blocks of 32-registers each. Each set of two registers represents one map-page entry. After using the most significant 12 bits of the

virtual page map number to index into LVL1, the remaining 5 bits of the virtual page map number select a register within the LVL2 block. If both the LVL1 and LVL2 entries are valid, and if all other status indications are positive, the selected LVL2 map register produces the 22-bit physical page frame address. Note that since the LVL2 map blocks are indexed by the least significant 5 bits of the 17-bit virtual page map address, each of the 128 blocks represents 16 contiguous virtual pages.

Memory Map Level 2 Control Bits



TM1	<15>	=	last access TM1
TMO	<14>	=	last access TMO
L	<13>	=	last access locked
GCV	<12:11>	=	Garbage Collector volatility bits
F	<10>	=	force allowed
A	<9>	=	access bit: 0 = no access
STAT	<8:6>	=	status bits (see Figure 5-4)
REP	<5:4>	=	representation type: 0 = list, 1 = structure, 2,3 unused

Memory Map Level 2 Address Bits

Figure 5-2 Explorer I Level-2 Maps (LVL2)

The second level map is read in as two separate functional sources since the data field is greater than 32 bits. The field is separated into Memory-Map-Level-2-Control-Bits and Memory-Map-Level-2-Address-Bits.

To handle a page exception a check must first be made to see if the LVL1 entry is valid. A bit in the Memory-Map-Level-1 register indicates this. If the first level map entry is not valid, a block of second level map must be allocated and initialized with "map not valid" entries. The first level map must be set up to point to it. From here the page exception is handled as a second level map miss (described later).

If the first level map entry is valid then the data in the second level memory map control register determines the action to be taken. The map status code is used to determine the processing case.

5.7 EXPLORER I REVERSE FIRST LEVEL MAP

For each block of 128 Second Level Map registers, there is an entry in the Reverse First Level Map which gives the number of the First Level Map entry which points to this block, or else indicates that this block is unused. It contains a value which, if placed in the Virtual Memory Address register (VMA), would address that first level map entry, or else it contains -1 to indicate that this block is not currently pointed to. The Reverse First Level Map is held in A-memory and is 128 words long. It is used when allocating map level-2 blocks.

5.8 EXPLORER I MEMORY MAP ALLOCATION

A simple clock scheme is used for allocation of second level memory map blocks. If a level-1 map fault occurs, the Reverse First Level Map is consulted to see if the level-2 map block is owned by this level-1 block. If so, the valid (V) bit is set and the memory reference is restarted. If not, a new level-2 map block must be allocated to this level-1 entry.

To find a level-2 map block to allocate, the Reverse First Level Map is scanned. If the value is -1 then this level-2 map block is free and can be allocated. If the value is not -1 then it contains the address of the map level-1 entry which owns this level-2 map block. If the level-1 valid (V) bit of this entry is not set then this block has not been used recently and is allocated to the new entry. If the valid (V) bit is set, then this entry is in use. The level one entry is aged by turning off the valid bit. The scan continues at the next Reverse First Level Map entry. The scan will wrap around if the end of the table is encountered. Since the aging is done during the scan, if the entire structure is scanned and no level-2 map block is found, then the scan merely continues and will choose the next entry since it was aged during the last scan.

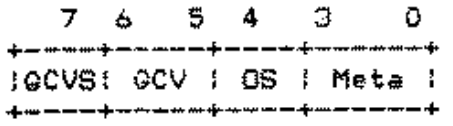
The map level-2 block is allocated by setting the index in the map level-1 entry and updating the Reverse First Level Map to reflect the new allocation.

5.9 EXPLORER II MEMORY MAP HARDWARE

The Explorer II map hardware consists of the Address Space Map (ASM), the Virtual Memory Maps Status (VMM Status), and the Virtual Memory Maps (VMM). The Address Space Map and the VMM Status replace the level-1 maps on the Explorer I and the VMM replaces both level-2 maps.

The Address Space Map (see Table 5-1) contains garbage collection oldspace and volatility information. There are 4K entries in this table, one entry for every 16 pages or 8K words, which is equivalent to two entries for every address space quantum (16K words; the smallest region size). This table is addressed by the high order 12 bits of the 25 bit virtual page address.

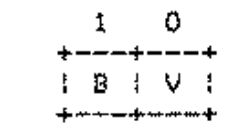
Table 5-1 Address Space Map Bits Assignment



GCVS <7> : Garbage Collection volatility valid status, 0 = true
GCV <6:5> : Volatility level for this address space half quantum
OS <4> : Old Space bit, 0 = Oldspace
Meta <3:0> : Address space Map Meta bits

The Virtual Memory Map Status contains two entries for each virtual page and therefore contains 128K entries. Each table entry consists of two bits (see Table 5-2), which indicates whether the VMM table has a valid entry. It also indicates in which VMM bank, right or left, the valid entry can be found, if one exists.

Table 5-2 Virtual Memory Maps Status



V <0> : Valid bit, 1 if map is set up
B <1> : Bank select, 1 if right bank, 0 if left bank

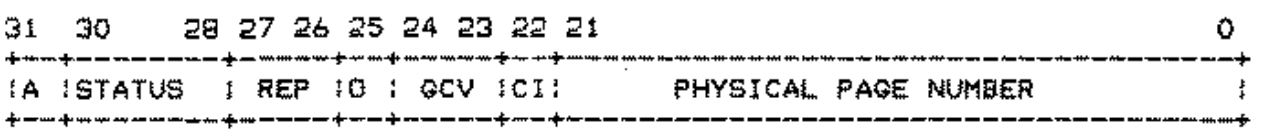
The Virtual Memory Map (VMM) consists of two banks. Each bank has 16K 32 bit maps. Each entry has the 22 bit physical frame address needed by the hardware to map from virtual memory to physical memory. The maps also contain two access bits that control the type of access to the maps. Bit 31 is the read/write access bit, and will cause a page exception if not set on any mapped cycle. Bit 30 is the write access bit, which is also the top bit of the 3 bits status field (see Table 5-3), will cause a page exception on mapped write cycles if not set.

When a mapped cycle is done, the mapping hardware will first check the the Virtual Memory Map Status Ram by indexing into it with bits 8 though 24 of the virtual address used. If the valid bit is not set, then a map miss page exception occurs. If the valid bit is set, then the hardware checks the bank select bit in the Virtual Memory Map Status to see which VMM bank it should use, the right or left. It then gets the entry from the correct bank by indexing into with bits 8 through 21 of the virtual address. If the read/write bit is set in the VMM and the cycle is a read or if both the read/write access and write access is set and the cycle is a write, then it can complete the cycle by using the physical address. If the access bits are not set correctly then a page exception occurs. The 32-bit physical address is constructed from the top 22-bit physical page number along with the 8-bit map page offset from bits 0 through 7 of the virtual page address, with byte offset, the last two digits, being 0.

Since the VMM banks are indexed by bits 8 through 21 of the virtual address, bits 22 through 24 indicate the 8 different pages that will index to the same VMM index. Since there are two banks, 2 of these 8 pages can be set up at the same time.

If the cache inhibit bit is not set, the hardware will use the cache if it can, and do cache fills as needed.

Table 5-3 Virtual Memory Map

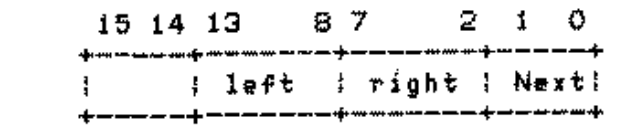


A	<31>	: Access bit, 0 is no access
Status	<30:28>	: Status (see Figure 5-4)
Rep	<27:26>	: Region representation, 0 is list space, 1 is structure space, 2 and 3 are unused
O	<25>	: Oldspace bit, 0 = oldspace
GCV	<24:23>	: Garbage Collection Volatility bits
CI	<22>	: Cache Inhibit, 1 = inhibit
Physical Page Number	<21:0>	: The top 22 bits of the 32-bit physical page address.

5.9.1 Explorer II Map Usage Table (MUT).

The Explorer II Map Usage Table (MUT) resides in physical memory and contains 8192 entries (see Table 5-4). Since there are two entries per 32 bit word, it occupies 4096 words of physical memory. Since one 2K page uses two maps, each entry has information for two left and two right VMM maps. For each pair of maps, it shows which of the 8 possible virtual memory pages is using the left bank and which is using the right bank. It also shows which bank to use next when one of the remaining 6 pages is accessed and therefore needs the maps set up.

Table 5-4 Map Usage Table (MUT)



Next <0:1> : 00 --> wired page, always in left bank
01 --> use right bank next
10 --> use left bank next
11 --> unused

Right <2:7> : The bottom 3 bits of this field indicate the top 3 bits of the page that currently has the map in the right bank. If the field is -1, then it is currently unused.

Left <8:13> : The bottom 3 bits of this field indicate the top 3 bits of the page that currently has the map in the left bank. If the field is -1, then it is currently unused.

<15:14> : Unused.

5.10 EXPLORER II MEMORY MAP ALLOCATION

Explorer II map allocation is much easier then the Explorer I. When a map exception occurs, bits 9 through 21 are used to index into the MUT (actually, bits 10 through 21 are used to get the correct word and bit 9 is used to select the halfword). The NEXT field is accessed to see if the new map should be set up in the right or left VMM bank. If the left bank has a wired page, the right bank is always used. The appropriate RIGHT or LEFT field from the MUT entry is accessed to see who currently owns the map. If it is -1, then there is no current owner. The current user virtual address is calculated from the bottom 3 bits of the LEFT or RIGHT field from the MUT along with bits 9 through 21 of the current virtual address (i.e., the one that needs the maps set up). This is used to invalidate the VMM Status Table for the two maps that currently are using the bank needed. The MUT is changed to reflect the new owner, the NEXT field is updated to use the other bank next time (unless it has wired status) and the table entry is rewritten. The VMM data in the correct bank is set up, and the VMM Status is set to valid with the correct bank select. The memory cycle that caused the page exception is then restarted.

5.11 PAGE EXCEPTION HANDLING

A memory reference which causes the maps to be altered, but can then complete produces a page exception. This includes references which must go to the PHT to reset the maps. A reference which cannot complete without access to disk (no valid information even in the PHT) is said to cause a page fault. This section describes page exception handling.

A page exception can be generated by the memory mapping hardware for a number of different reasons during the 2-level map lookup. A level-1 miss can occur if the valid bit is not set in the level-1 map. A level-2 miss occurs when a non-normal map status is found in the second level map. Examples page exceptions include inappropriate read/write access privileges; page is in the processor's PDL buffer or A-memory; page is marked as "trap on any access" (the "MAR" break feature) and others. Depending on the circumstances, some of these conditions may eventually be signalled as traps by the microcode, or the reference may continue and complete normally. The specific exception handling is based on the Memory Access and Memory Status fields and is described in detail below.

5.11.1 Memory Access Codes.

Memory Access codes are listed in Figure 5-3. Access codes are determined by combining the access bit and the most significant bit of the status bits field. These codes specify what memory operations (read, read/write or none) are to be permitted to these pages. If the desired access is permitted the memory system performs the operation. If the desired access is inappropriate, the page fault condition is set and the page exception handler is invoked. The exception handler then uses the status code to determine what steps to take next.

Access Bit	MS Status Bit	Code	Access-Type Meaning
0	x	0,1	No Access
1	0	2	Read Only
1	1	3	Read/Write

Figure 5-3 Memory Map Access Codes

5.11.2 Memory Map Status Codes.

In this section each memory map status code is examined in detail. The possible map status values and their interpretations

location information. In other words, there is first-level map information available, but no physical mapping information. This type of map entry is created when a pointer to an object is used but the object itself is not referenced. The oldspace and region representation type bits in such a map entry are needed by the garbage collector. An attempt to access the storage associated with the object later will be treated like a map miss.

5.11.2.3 Status Code 2: Read Only.

An attempt was made to write to a page that is set to read-only will cause a page exception. A special case is made for a forced write operation. In this case, the write occurs and no access fault is declared. This is needed so that the garbage collector/compactor can move data structures that ordinarily need protection.

If the operation is a regular, non-garbage collection write and the special A-memory/Lisp variable SYS:%INHIBIT-READ-ONLY is NIL, then the operation is declared illegal and an error is signaled. If SYS:%INHIBIT-READ-ONLY is non-NIL, the access is allowed.

5.11.2.4 Status Code 3: Read/Write First.

A page exception occurs on an attempt to write. The processing for this exception consists of changing the status in the map and the page hash table to read/write, indicating the contents of the page have been modified. The reference is restarted. This facility implements the dirty page status.

If the page that is being set to dirty is currently assigned to a read-only page band, then it will be reassigned to a read/write page band (given swap space) later when the page needs to be swapped out of physical memory.

5.11.2.5 Status Code 4: Read/Write.

A normal, fully mapped page. No exception should occur on this type of page. If this status occurs, the hardware is faulty, and a crash sequence will be initiated.

5.11.2.6 Status Code 5: Page might be in PDL Buffer.

Certain areas which contain regular PDLs arrange to get the map set to this status for their pages (instead of read/write). The microcode has to decide, on every reference, whether the page is in the processor's PDL buffer registers or in main memory, and simulate the appropriate operation. It may be that only part, or none, of the page is in the PDL buffer on a particular reference. Thus the page exception handler must test the virtual address to see if it falls in the range which is really in the PDL buffer right now. If not, temporarily turn on read/write access, make the reference, and turn it off again. Pages may be swapped out

without regard for whether or not they are in the PDL. This works because the normal course of swapping out invalidates the second-level map. If the page is then referenced as memory, it will be swapped in normally and its map status restored from the Region-Bits table, in the normal fashion. This will then restore the Maybe-PDL map status. Otherwise, the addressed word is in the PDL buffer. Translate the virtual address to a PDL buffer address and return the appropriate register contents.

5.11.2.7 Status Code 6: Possible MAR Trap.

The memory address register (MAR) facility allows any word or contiguous set of words to be monitored constantly, and cause a trap if the words are referenced in a specified manner. The name MAR is from the similar device on the ITS PDP-10's. The MAR trap status is set for all pages that are in the range of addresses being monitored. This range is indicated in the A-memory/Lisp variables SYS:%MAR-LOW and SYS:%MAR-HIGH. When the MAR trap occurs, the virtual address is checked to see if it falls in the range. If so, a trap may be called. It should be noted that traps, since they cause sequence breaks, are not allowed during stack group switches, so if a MAR-monitored address is referenced, a sequence break flag is set, and the break will occur at the next appropriate time.

If an address falls within the MAR'd range, then the action taken depends on the currently active MAR-mode (MAR-mode is actually a 2-bit field in each stack group's mode flags). The action taken for various flag word values is shown in Table 5-5.

Table 5-5 MAR Status Codes

Value	Memory Operation	MAR Mode	Action
0	Read	MAR disabled	No trap
1	Read	Read Trap	Trap
2	Read	Write Trap	No trap
3	Read	Read/Write Trap	Trap
4	Write	MAR disabled	No trap
5	Write	Read Trap	No trap
6	Write	Write Trap	Trap
7	Write	Read/Write Trap	Trap

5.11.3 QCV and Oldspace Exceptions.

In addition to access/status fault exception handling, exception handling is also performed for two garbage-collection- related conditions: Garbage Collection Volatility Fault and Oldspace. These two are described briefly below. More information on them

is presented in Section 9 (Garbage Collection). These exceptions are different from access/status exceptions in that the mapping hardware does not signal the page fault condition for them. Rather, the reference produces GCV and Oldspace conditions that the microcode can then selectively dispatch upon.

5.11.3.1 Oldspace.

The Oldspace condition is indicated by the Oldspace Meta Bit in the first-level map being 0. It indicates that objects in these pages are currently being garbage collected. The Oldspace Meta Bit can be ORed selectively into a microcode dispatch. The TRANSPORT dispatches use this bit. The TRANSPORTER is invoked, which decides if this is an object that needs to be moved for garbage collection. For more information on the TRANSPORTER, see Section 9.

5.11.3.2 Garbage Collection Volatility Faults.

The Garbage Collection Volatility Fault condition is also a 1-bit field. It is signaled whenever a younger object (contained in MD, memory data register) is written into older memory (represented by VMA, the memory address register). The GCV output is determined by a comparator which examines the 3-bit GCV field in the first-level map and the 2-bit GCV field in the second-level map. The details of the fault inputs and output, along with their interpretations in the Temporal Garbage Collection implementation, are presented in Section 9.

On the Explorer I, the GCV fault is stored in an inaccessible 1-bit GCV register, and is preserved only until the next memory reference is performed. Thus, it can only be sensed by a dispatch using GCV done immediately after the memory cycle. On the Explorer II, the GCV register can be read and written as a functional source and destination, hence can be stored across other memory references.

5.12 VIRTUAL MEMORY SYSTEM TABLES

There are some additional tables associated with paging that are used by the microcode: the Page Hash Table (PHT) contains an entry for every virtual page that is currently memory resident. The physical page data table (PPD) contains an entry for every physical page of memory. These tables are described in further detail below.

5.12.1 Page Hash Table.

The page hash table (PHT) resides in physical memory and is not part of the virtual address space. It describes every virtual

page that is currently resident in physical memory. When the microcode detects a map miss, the PHT is consulted to see if the virtual page is still in physical memory. If so, the Level 2 maps (Explorer I) or VMM hardware (Explorer II) can be set up from the information stored in the PHT, and the memory reference will be restarted. Since the mapping hardware now contains valid mapping information, the reference should complete normally. If there is no valid PHT entry for a virtual address, a page fault is said to have occurred, and the page must be read in from disk.

The page hash table is also used to store information used by the page aging and swap management functions. A full description of the role of the page hash table in these contexts is described in a later section.

5.12.2 Physical Page Data Table.

The Physical Page Data Table (PPD) resides in physical memory and is not part of virtual address space. When the system is booted, the system determines the size of primary memory and allocates a suitable portion of physical memory for this table.

An entry for a page in the Physical Page Data Table is shown in Table 5-6. There is one PPD entry for every 512-word page of physical memory. A PPD entry thus represents a logical page frame or page frame number (PFN) in the physical memory pool. Each entry is broken into two parts. The low order 16 bits of the word contain an index into the Page Hash Table or a page allocation status. The high order 16 bits of the word contain an index into the Physical Page Data Table linking this page to the next most recently used physical memory page.

All the physical pages that are allocated to hold the microcode management tables are marked #x+FFFFFFFF, to indicate that these pages are not to be used in the virtual memory page pool.

The Physical Page Data Table is used to determine which virtual page is contained in a given physical page. The microcode page aging and replacement algorithms are driven by a scan of the Physical Page Data Table.

Table 5-6 Physical Page Data Area Word Format

Value	Meaning
PHT INDEX FIELD:	
FFFF (hex)	System wired or permanently wired page. Page is not available in virtual memory pool.
FFFE	Free page. Page is available in the virtual memory pool but is not currently in use.
PHT Index	Normal page. Value contains the index of the page hash table entry for this page.
PPD LINK FIELD:	
FFFF (hex)	Not available. Page is not available in virtual memory pool.
FFFE	Write in progress. Page is in virtual memory pool, but is currently being written to disk.
FFFD	End of LRU list. Page is the most recently referenced page in virtual memory pool.
PPD Link	Normal page. Value is index of the physical page table entry of the next most recently referenced page in the virtual memory pool.

Given a page frame number, information about the virtual page associated with the physical page can be determined as follows. Use the PFN as a word index into the PPD. If the PPD index field is valid (greater than FFFE), then there is a virtual page in this page frame. Use the index field shifted left by 1 as a word index into the PHT.

5.13 A-MEMORY PAGING INFORMATION

In addition to the Explorer I Reverse-First-Level-Map, other virtual-memory-specific information is kept in portions of A-memory, as described below.

5.13.1 PDL Buffer Handling.

A-memory also contains the first virtual address which currently resides in the PDL buffer (in A-PDL-BUFFER-VIRTUAL-ADDRESS), and the PDL buffer index corresponding to that address (in A-PDL-BUFFER-HEAD). Note that the valid portion of the PDL buffer can wrap around as shown in Figure 5-5.

When a page exception is taken for a page that might be in the PDL Buffer, the microcode handler must check to see if the virtual memory address falls in the valid portion of the PDL Buffer. The further processing of this case is explained under the map level 2 status code for PDL Buffer (status code 5).

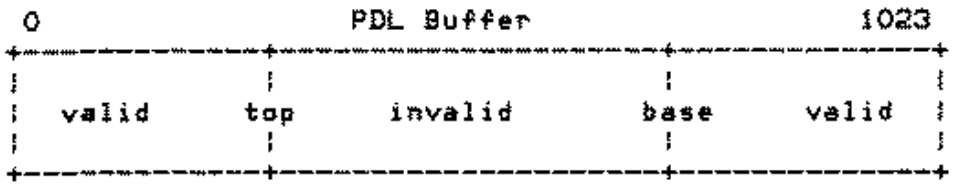
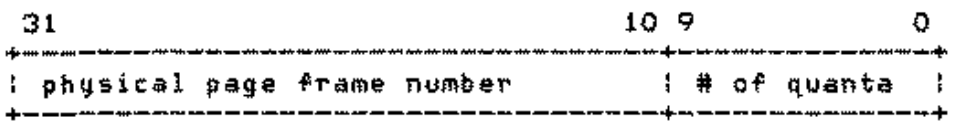


Figure 5-5 PDL Buffer Wrap-Around

5.13.2 Physical Memory Map.

The Physical Memory Map (PMM) is a set of A-memory locations which describe which logical page frame numbers (PFNs) reside at which physical memory (NuBus) board addresses. Each entry in the PMM describes one or more 2MB quantums of physical memory. The number of 2MB quantums is kept in the low 10 bits, while the top 22 bits give the (NuBus) physical page frame number where the first quantum starts (see Figure 5-6).



Physical Memory Map Entry

Figure 5-6 Physical Memory Map Entry

SECTION 6

Paging and Disk Management

6.1 VIRTUAL PAGE MANAGEMENT

This section discusses techniques for dealing with the management of virtual pages and their associated disk storage. Virtual page management functions include the page replacement strategy, updating swap status indications, and mapping the paging-related secondary memory across different disk page bands, perhaps on different physical units.

The last portion of this section describes a number of virtual memory management primitives which can be used from Lisp.

6.2 PAGE MANAGEMENT OVERVIEW

The following is an overview of the paging and disk management algorithms for processing a page fault.

1. Given a faulted virtual address, find the disk device and the device block offset where the demanded page resides.

2. Issue a disk read operation into a previously cached memory page frame. If there is no available cached memory page frame, scan the PPD looking for a (preferably clean) page to evict. Issue the disk read into the page frame found.

3. While waiting for the demand page disk I/O to complete, find a new cached memory page frame for the next demanded page by scanning the PPD. Choose a page for eviction from physical memory. If the page is not dirty, the memory maps are cleared and the PHT entry deleted. The PPD entry is coded "FFFE" indicating a free page.

4. If the page to be evicted is dirty then it must be written to a swap partition. The PPD entry is coded Disk-Write-Pending. The PHT entry and the memory maps are cleared by the disk servicing routine later after the disk operation has completed.

5. When swapping out a dirty page, determine if there are contiguous dirty virtual memory pages that can be swapped out at the same time to adjacent disk locations. If so, prepare a multi-record write request for all such pages.

6. If any pages being swapped out have never been assigned read/write page device storage, find an available page device and assign space to the pages' clusters (group of 16 contiguous virtual pages). Initiate the disk write.

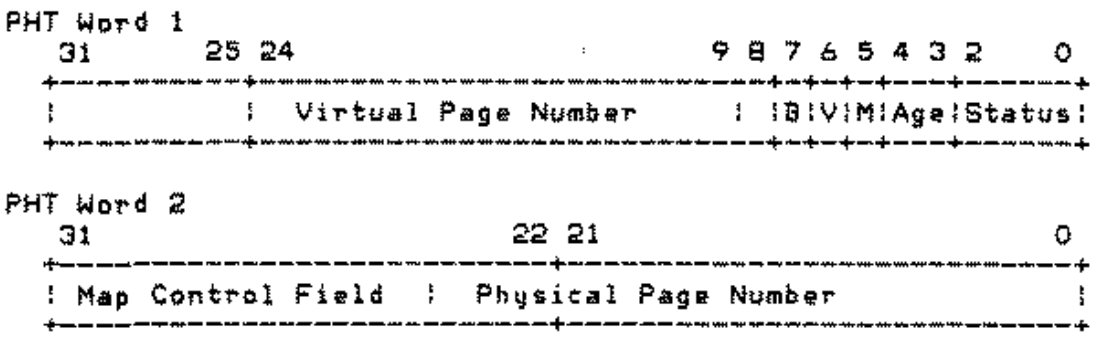
7. Create a PHT entry for the faulted page.

8. Wait for the original disk read to complete. If any disk writes were initiated they will continue in parallel with computation.

6.3 PAGE HASH TABLE

The Page Hash Table (PHT) contains physical memory mapping information for every virtual page currently in physical memory. Page exception handling may consult the PHT in order to find information to use in reloading the hardware maps. In addition, the page replacement algorithm (FINDCORE) uses information stored in the PHT. The format of a PHT entry is shown in Figure 6-1.

If there is a page exception and there is no entry in the Page Hash Table then it is a page fault, and the page needs to be read from disk. The disk address is calculated using a disk mapping scheme described later.



WORD 1:

STATUS	<0:2>	= Map status code. See table of status codes below.
AGE	<3:4>	= Number of times page has been aged.
M	<5>	= Modified bit. 1 = Modified (dirty).
V	<6>	= Valid bit. 1 = PHT entry valid.
B	<7>	= Background Write bit. 1 = Background write in progress.
VPN	<9:24>	= Virtual Page Number

WORD 2:

PHYS-PG	<0:21>	= Physical Page Number.
MAP-CONT	<22:31>	= Hardware map control data.

Figure 6-1 Page Hash Table Entry Format

6.3.1 Page Replacement Process.

The page replacement strategy is used to determine which pages should be swapped out so that a needed virtual page can use this physical memory page frame. It is also known as FINDCORE.

Page selection is accomplished on demand when a page is needed for reuse. The physical page data table is consulted starting with the top of the Least Recently Used (LRU) list formed by the entries in the PPD LINK FIELD of each entry. The Least Recently Referenced page index is held in an A-Register that defines starting entry of the list. The memory page indicated is the least recently referenced page. The status of the page is checked, and if it is available, it is designated for immediate use. If the page is not available for some reason, the scan proceeds to the page indicated by the contents of the PPD Link Field of the current entry and so forth until the oldest usable

page is found.

While scanning down the list, action is taken on individual pages as follows by consulting the page's PHT status code as described below. Table 6-1 provides a complete description of all PHT status codes.

1. If page status is normal then set status to age trap and the PPD link is removed from its current position and added to the end of the list as the new Most Recently Referenced page.

2. If page status is age trap, flushable or prepage, then remove the page from the list and designate it as the next page to be used for replacement. The scan stops at the first page found that satisfies this status.

3. If page status is anything else, or if the background writing bit is set, continue the scan with the next linked page leaving this page in the same place in the list.

Table 6-1 Page Hash Table: Swap Status Codes

Unused:	0 - Unused code.
Normal:	1 - An ordinary page is swapped in here.
Flushable:	2 - Means that there is a page here, but it has been marked as no longer needed and can be used to swap a new page into. This page may first have to be written out if the map status indicates that it has been modified since last written (map status code = 4)
Pre-page:	3 - Same as Flushable, but came in via Prepage.
Age trap set:	4 - This page was in normal status, but is now being considered for swap-out. The map may not be set up for this) page. If someone references the page, the swap status should be set back to "normal."
Wired down:	5 - The page swapping routines may not re-use the memory occupied by this page for some other page. This is used for the some other page. This is used for memory pages wired down by the SYS:WIRE-PAGE primitive and related routines.
Disk Read Pending:	6 - This page is being read in from disk. The virtual address assigned this page may not be used and the page swapping routines may not re-use the memory occupied by this page.
Disk Write Pending:	7 - This page is being written to disk in anticipation of freeing it for re-use by the page swapping routines. The virtual memory can be referenced and used, but the physical memory may not be re-used until the disk operation has completed.

6.3.2 Page Table Sizes.

The size of the page hash table is related to the size of physical memory. Since a hash technique is used to search the page hash table, two 2-word entries are usually allocated for every physical page in the system. See Figure 6-2 for sizes of both the PHT and the PPD for different memory sizes. The PHT and PPD together consume slightly less than 4% of the physical memory.

Virtual Memory Size = 128K Words

Physical Memory Size		PHT Size	PPD Size
2MB	(512K Words)	1024 pages	4096 words	1024 words
4MB	(1M Words)	2048 pages	8192 words	2048 words
8MB	(2M Words)	4096 pages	16393 words	4096 words
10MB	(2.5M Words)	5120 pages	20480 words	5120 words
12MB	(3M Words)	6144 pages	24576 words	6144 words
16MB	(4M Words)	8192 pages	32768 words	8192 words
32MB	(8M Words)	16393 pages	65536 words	16393 words
64MB	(16M Words)	32768 pages	131072 words	32768 words
128MB	(32M Words)	65536 pages	131072 words	65536 words

Figure 6-2 PHT and PPD Sizes

6.3.3 PHT Hashing Algorithm.

When the PHT is searched due to a page exception, a hash technique is used. A number of the most significant bits in the virtual page number are used as the hash key. The hash key is shifted left by one to produce a PHT-entry index into the table. The entry's PHTI Virtual Page Number is then checked against the original virtual address (if the PHTI Valid Bit is set).

Hash collisions are resolved by adding a linear rehash constant to the original hash value, and wrapping around the front of the table if necessary. The number of bits used in the hash is proportional to the PHT size. A physical memory size of 8MB, for example, uses the top 13 bits of the virtual address and has a maximum of 8 collisions per hash value. A 32MB system uses 15 bits and has a maximum of 2 collisions per hash value. At 64MB and larger, all 16 bits of the virtual address are used, and the PHT is effectively just straight-indexed.

No effort is made to keep collision chain-link information in the PHT entries because of the high overhead this would require.

Instead, a simple count of the longest collision chain encountered so far is maintained. This count, in the A-memory/Lisp counter SYS:%PHT-SEARCH-DEPTH, is used and incremented by the microcode. Each PHT hash must examine at least SYS:%PHT-SEARCH-DEPTH number of entries before declaring a hash failure. When a new virtual address is added to the hash table, a count of valid entries skipped over is kept until a free PHT entry is found. Then if that count is larger than the SYS:%PHT-SEARCH-DEPTH count the latter is incremented. The length of the longest collision chain seen can decrease when PHT entries are deleted. The microcode, however, is not authorized to decrement SYS:%PHT-SEARCH-DEPTH. Instead, a low-priority PAGE-BACKGROUND process periodically scans the table; computes the current longest collision chain; and updates SYS:%PHT-SEARCH-DEPTH accordingly.

Initially the Page Hash Table contains no entries and the Physical Page Data table entries are coded FFFE (page available for use). All cells in the Page Hash Table are zeroed. As each is allocated on demand, a PHT entry is created and entered into the table.

6.4 DISK PAGE MAPPING SCHEME

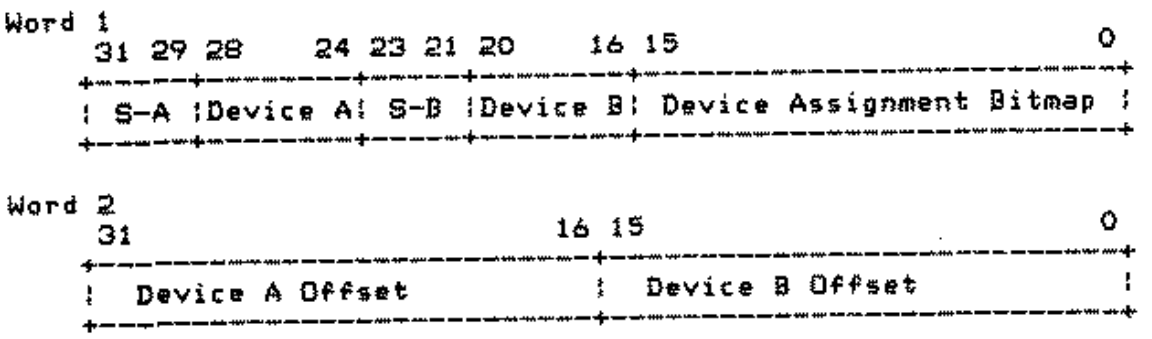
The following describes the mechanism by which a virtual page number is used to map to a disk address in one of the system's logical paging devices.

A logical page device in the Explorer system is a disk partition where disk storage is assigned to a number of virtual pages. The Lisp world LOD band is a read-only page device; that is, demand pages are read from this partition but if they are later dirtied they will be written out to a read/write page device (a PAGE band). New virtual pages created during program execution (when new Lisp objects are consed) will have read/write storage allocated to them when they are evicted from main memory.

Because of the large number of virtual pages (2**16) it is not practical use straight indexing (a one-to-one correspondence) between virtual pages and swap partition addresses. Instead, pages being swapped are be assigned the next available space in a swap partition and a disk page map is maintained for translating between virtual addresses and disk addresses. Each two-word entry in this Disk Page Map Table (DPMT) contains mapping information for a group of 16 contiguous virtual pages called a cluster; hence the DPMT is indexed by the top 12 bits of the virtual address, called the cluster number.

Through the DMPT, disk space is allocated in clusters of 32 blocks corresponding to 16 virtual memory pages (each disk block

is 1024 bytes or 256 words). The low 4 bits of the virtual page number select the page offset in the disk cluster. The format of a DPMT entry is shown in Figure 6-3.



Device Assignment Bitmap <0:15> = Indicates to which device (A or B) each of the 16 pages is assigned
0 = Device A (page band)
1 = Device B (load band)

Device B <16:20> = Logical page device number of device B

S-B <21:23> = Device B status

Device A <24:28> = Logical page device number of device A

S-A <29:31> = Device A status

Device B Offset <0:15> = 32-block offset from start of partition

Device A Offset <16:31> = 32-block offset from start of partition

Figure 6-3 Disk Page Mapping Table

*Status 0: No device assigned

*Status 1: Read only band (load band)

*Status 2: Read/Write band (page band)

*Status 3: Read/Write band assigned, however, a disk block has not yet been allocated

*Status 4: - 7: Unused

Figure 6-4 Device Status Codes

Each entry of the DPMT specifies two logical paging bands, Device A and Device B. By convention Device A describes a read/write device (swap band) while Device B describes a read-only device (the load band). A page represented by this cluster will have storage assigned on one of these two page devices; a bitmap in the entry specifies to which device each page in the cluster is mapped. The disk block corresponding to this virtual page on the page band not selected by the bit map is reserved but not used. If the entry in the bit map is switched this page would then be assigned to this other disk block.

A table is kept describing all the logical paging devices known to the virtual memory system. The Device A and Device B fields are conceptually an index into this table. Each PAGE partition on disk is represented as a separate logical page device In this way, there may be several paging bands on a single device or on several devices.

The fields S-A and S-B are the device assignment status fields for device A and device B respectively. They indicate whether the device is actually used by any pages in the cluster and if so, what kind of access is allowed. The values for these fields are described in Figure 6-4.

A virtual page operation would proceed as follows:

1. Using the most significant 12 bits of the virtual page number pick up the Disk Mapping Table Entry for the cluster.

2. Consulting the bit map decide whether this page is assigned to device A or device B of this cluster.

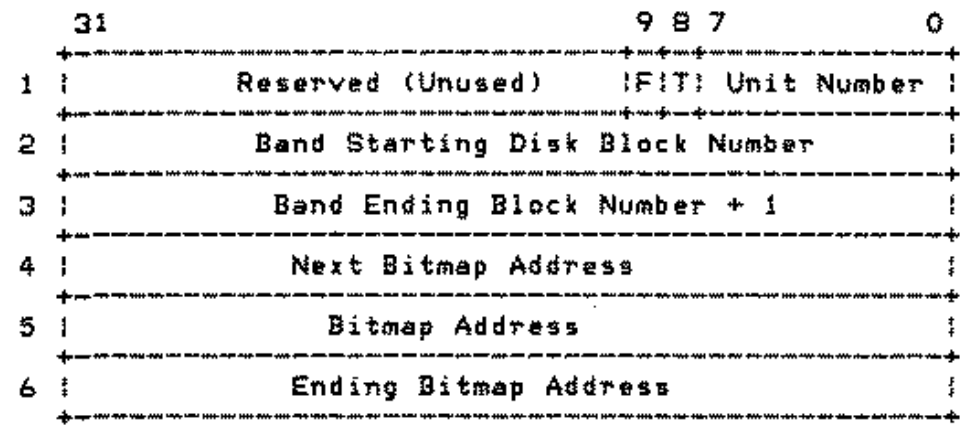
3. Using the device status field decide if a valid operation is being performed on this device. If no valid operation can be performed, call ILLOP (crash).

When the system is first booted, the DPMT indicates that all virtual pages reside in the read-only load band associated with Device B. As these pages are demanded they are read into memory from the Lisp load partition. They are not assigned swap space unless they are subsequently modified and swapped out. This avoids the necessity of copying the load band to the swap band before starting up Lisp.

Virtual pages that are as yet unused at boot time (unallocated virtual memory) are marked as unassigned in the DPMT. When a dirtied load band page or a freshly created virtual page is assigned swap space the logical paging device number of the swap partition is placed in the DPMT Device A field, the device A status is marked as read/write, and the appropriate Device A cluster offset is placed in word two of the DPMT entry. Unless this virtual page is later freed by garbage collection, it will be read from and written to this disk block from now on.

6.5 LOGICAL PAGING DEVICES

A logical paging device defines a set of contiguous blocks on a secondary storage device. The characteristics and allocation information for each paging device are maintained in a Logical Page Device Information Block structure (LPDIB). The LPDIB is described in Figure 6-5.



Word 1: Page device status information.
F <9> = Full bit
1 = all clusters allocated
T <8> = Device type
0 = read-only (load)
1 = read/write (page)
Unit <0:7> = Physical unit number of device
Word 2: Starting block number of the page band.
This is an absolute block number on the device.
Word 3: The absolute block number of the first block that is outside of the band.
Word 4: Address of the next bitmap word that contains an unallocated cluster.
Word 5: Address of the first word of the cluster-allocation bitmap. Each bitmap bit represents one cluster (32 blocks) of storage.
Word 6: Address of last bitmap word.

Figure 6-5 Logical Page Device Information Block

The LPDIBs are allocated contiguously in the SYS:DEVICE-DESCRIPTOR-AREA beginning at the address stored in SYS:ADDRESS-OF-PAGE-DEVICE-TABLE. Enough space is allocated for a constant number of paging devices (the value of SYS:NUMBER-OF-PAGE-DEVICES). During boot the microcode initializes the LPDIB for the load band. Later, during Lisp boot (in SYS:LISP-REINITIALIZE), SYS:CONFIGURE-PAGE-BANDS is called to find all the Explorer PAGE bands in this configuration and allocate LPDIBs for each of them up to the system maximum.

Disk blocks are allocated to clusters by finding a free page cluster in the LPDIB bitmap. The bitmap has one bit for each page cluster. A 1 indicates the cluster is free and a 0

indicates it is in use. Word 4 of the page device information block points to the next bitmap word to check, and is incremented as all 32 clusters indicated by that word are allocated. When it gets to the last bitmap word, it starts again at the beginning. If there are no clusters left, then the Full Flag is set and the swap band is no longer checked until a cluster is returned. Garbage collection will return clusters as the virtual memory they represent is collected and freed for re-use.

6.6 VIRTUAL MEMORY SYSTEM SUBPRIMITIVES

The subprimitives and special variables described below affect the paging algorithms of the virtual memory system. As with all Explorer system subprimitives, some of these can be extremely dangerous (cause crashes or strange behavior) if misused.

Byte specifiers and constants described can be found in SYS:UCODE;LROY-QCOM. The Lisp-coded functions can be found in the MEMORY-MANAGMENT files listed below:

MEMORY-MANAGEMENT; PAGE-DEFS
MEMORY-MANAGEMENT; PAGE
MEMORY-MANAGEMENT; PAGE-DEVICE
MEMORY-MANAGEMENT; PAGING-PROCESS
MEMORY-MANAGEMENT; PHYSICAL-MEMORY
MEMORY-MANAGEMENT; VM-BOOT

The A-Memory counters (and some A-memory variables) used by the virtual memory system are documented in the section entitled Other Subprimitives, Variables, and Counters.

%disk-switches Variable

This variable contains bits that control various disk usage features. The byte specifiers listed below are stored in the DISK-SWITCHES-FIELDS list.

Bit 0 (%%Clean-Page-Search-Enable). Page replacement algorithm will scan through physical memory looking for a clean page to flush on a FINDCORE operation. Default is on.

Bit 1 (%%Time-Page-Faults-Enable). Enables %TOTAL-PAGE-FAULT-TIME in the counter block. Value of counter is microsecond time spent in the page fault microcode plus disk wait time, but excluding code that resolves page exceptions. Default is off.

Bit 2 (%%Multi-Page-Swapout-Enable). Enables the page

replacement algorithm to clean adjacent memory page images by writing them to disk in the same disk write for a page being flushed. Default is on. Turning it off will degrade paging performance.

Bit 7 (%%SB-During-Disk-Wait-Enable). Not used.

Bits <8:15> (%%Multi-Swapout-Page-Limit). Maximum number of pages that can be updated in a multi-swapout. Values must be between 0 and 255. Default is 128.

Bits <16:23> (%%Serial-Delay-Constant). Timing constant for microcode access to the serial chip registers. This must NOT be less than 12, which yields a delay of at least 2.641 microseconds on Explorer I. Don't change this unless you know what you're doing.

set-disk-switches
(&key clean-page-search time-page-faults multi-page-swapouts sequence-breaks-during-disk-wait multi-swapout-page-count-limit serial-delay-constant) SET-DISK-SWITCHES is a user interface to safely alter the dynamic paging variables using symbolic keyword definitions to specify the fields. The defaults for each switch are "safe" values. The value returned is the new value of %DISK-SWITCHES.

set-swapin-quantum-of-area (area &optional (swapin-quantum 3))
Specifies that pages of AREA (which should be an area-number) should be swapped in in groups of 2**SWAPIN-QUANTUM pages at a time. The default is 3, which means that prepaging (if active) will swap in up to 8 pages at a time.

The swapin quantum is used only if prepaging is enabled. Currently the Explorer II paging microcode uses the prepaging feature, but the Explorer I paging system does not.

set-all-swapin-quanta (&optional (swapin-quantum 3))
Specifies the SWAPIN-QUANTUMs of all non-fixed areas at once.

wire-page (address &optional (wire-p t))
If WIRE-P is T, the page containing ADDRESS is wired down, that is it cannot be paged-out. If WIRE-P is NIL, the page ceases to be wired down.

unwire-page (address)
(unwire-page address) is the same as (wire-page address nil).

page-in-structure (object)
Makes sure that the storage that represents OBJECT is in main memory. Any pages that have been swapped out to disk are read in. If OBJECT is large, this is useful in order to get all the paging required to bring OBJECT in over with at once, rather than having it occur a bit at a time as OBJECT is referenced.

The storage occupied by OBJECT is defined by the %FIND-STRUCTURE-LEADER and %STRUCTURE-TOTAL-SIZE subprimitives.

page-in-array (array &optional from to)
This is a version of PAGE-IN-STRUCTURE that can bring in a portion of an array. FROM and TO are lists of subscripts; if they are shorter than the dimensionality of ARRAY, the remaining subscripts are assumed to be zero.

page-in-pixel-array (array &optional from to)
Like PAGE-IN-ARRAY except that the lists FROM and TO, if present, are assumed to have their subscripts in the order horizontal, vertical, regardless of which of those two is actually the first axis of the ARRAY.

page-in-words (address n-words)
Any pages that have been swapped out to disk in the range of address space starting at ADDRESS and continuing for N-WORDS are read into main memory.

page-in-area (area-number)

page-in-region (region-number)
All swapped-out pages of the specified region or area are brought into main memory.

page-out-structure (object)

page-out-array (array &optional from to)

page-out-pixel-array (array &optional from to)

page-out-words (address n-words)

page-out-area (area-number

page-out-region (region-number)
These subprimitives exist only for compatibility with old code which may reference them. They currently do nothing and simply return NIL.

%page-status (virtual-address)
If the page containing virtual-address is swapped out, or if it is part of one of the low-numbered permanently-wired system areas, this returns NIL. Otherwise, it returns the entire first word of the Page Hash Table (PHT) entry for the page.

See the section on Paging and Disk Management for the format of a PHT entry. Byte specifiers for the PHT fields can be found in the PAGE-HASH-TABLE-FIELDS list.

%change-page-status
(virtual-address swap-status access-status-and-meta-bits)
The Page Hash Table (PHT) entry for the page containing VIRTUAL-ADDRESS is found and altered as specified. T is returned if it was found, NIL if it was not (presumably the page is swapped out). SWAP-STATUS and ACCESS-STATUS-AND-META-BITS can be NIL if those fields are not to be changed.

NOTE

This subprimitive is extremely dangerous since it does no error checking. The integrity of the virtual memory system can be irreparably damaged if this function is called improperly.

%compute-page-hash (virtual-address)

%compute-page-hash-lisp (va &optional max-byte-index max-byte-size)

%rehash (old-pht-index &optional max-index)
The first two return the hash value for VIRTUAL-ADDRESS (VA). The hash value is a byte offset into the Page Hash Table (located in physical memory) where the PHT entry for VIRTUAL-ADDRESS hashes to. However, this entry may already be in use. In that case, %REHASH may be used (given the old hash value) to find the next place to look.

%COMPUTE-PAGE-HASH-LISP and %REHASH are coded in Lisp. They take optional parameters which allow testing of the hash function on different sized hash tables. The defaults for the optional arguments are the values suitable for the whatever the running configuration is.

pages-of-physical-memory
Returns the total number of physical memory pages in the current memory configuration. Any number in the range [0 ..

(1- (PAGES-OF-PHYSICAL-MEMORY))] can be used as a valid page frame number (PFN) for functions which require them.

convert-physical-address-to-pfn (physical-address)

convert-pfn-to-physical-address (pfn)

convert-slot-offset-to-pfn (nubus-slot offset-into-slot)

convert-pfn-to-slot-offset (pfn)

convert-physical-page-to-pfn (phys-pg)

convert-pfn-to-physical-page (pfn)
These routines use the A-Memory Physical Memory Map to perform conversions between a logical page frame number (PFN) and a physical NuBus address. A PFN is simply a page number between zero and the number of physical pages available in the current configuration (which can be obtained by the PAGES-OF-PHYSICAL-MEMORY function). The physical address is a NuBus byte-oriented address which corresponds to PFN.

The manner in which the physical address is expressed depends on the function you use. CONVERT-PHYSICAL-ADDRESS-TO-PFN and CONVERT-PFN-TO-PHYSICAL-ADDRESS use 32-bit NuBus addresses. The next two use the slot/byte-offset into slot form of expressing a NuBus address (two values are returned from CONVERT-PFN-TO-SLOT-OFFSET). The ones with PHYSICAL-PAGE in their names use just the top 21 bits of the NuBus address.

%delete-physical-page (pfn)
This is used to delete pfn (which is a logical page number of physical memory) from the virtual memory pool. Any virtual page that is currently in pfn will be swapped out (if necessary) and the page will be marked so that the virtual memory system will not use it in the future for holding virtual pages. This is useful for reserving physical memory for use, say, as an I/O buffer for a device which does DMA I/O. It can also be used to force a virtual page to be swapped out of physical memory.

Returns T if page was deleted successfully. Returns NIL if page was already deleted.

%create-physical-page (pfn)
This reverses the action of %delete-physical page. That is, given a deleted pfn, %create-physical-page marks it so that it can be used in the by the virtual memory system to hold a virtual page.

set-memory-size (new-size-in-pages)
Specifies the size of physical memory available to the virtual memory system to NEW-SIZE-IN-PAGES. Can be used to decrease the number of physical pages available to virtual pages, or to increase it back to the system maximum if SET-MEMORY-SIZE has previously been used to lower it. This may be useful for measuring performance based on the amount of memory.

To determine the actual number of physical pages in the current configuration, use the PAGES-OF-PHYSICAL-MEMORY function. The variable %WORKING-MEMORY-SIZE contains the NEW-SIZE-IN-PAGES that you set using SET-MEMORY-SIZE.

SET-MEMORY-SIZE uses %DELETE-PHYSICAL-PAGE and %CREATE-PHYSICAL-PAGE.

get-contiguous-physical-pages (number-of-pages)

return-contiguous-physical-pages (number-of-pages slot offset)
GET-CONTIGUOUS-PHYSICAL-PAGES may be used to obtain a block of physically contiguous memory (and always on the same memory board). NUMBER-OF-PAGES is the amount desired. A physical page is 2048 bytes (the value of the variable PAGE-SIZE-IN-BYTES). The pages thus obtained will not be available to the virtual memory system until returned by the RETURN-CONTIGUOUS-PHYSICAL-PAGES function.

If NUMBER-OF-PAGES contiguous pages cannot currently be freed for use, NILs are returned. Otherwise, two values are returned: a NuBus slot number and the byte offset in the slot which together specify the 32-bit NuBus address of the start of the physical memory block obtained.

%physical-address (virtual-address)
Returns the NuBus physical address which VIRTUAL-ADDRESS currently occupies in main memory. This value is unpredictable if the virtual page is not swapped in; therefore, this function should be used only on wired pages, or you should do

```
(WITHOUT-INTERRUPTS
   (%P-POINTER virtual-address)    ;swap it in
   (%PHYSICAL-ADDRESS virtual-address))
```

Unless you assure that the page is wired, or use the physical address returned in a section of code that is guaranteed not to change the contents of physical memory, the value returned may not be meaningful for long.

%virtual-page-number (pfn)
Given a logical page frame number (PFN), returns the virtual

page number currently in the physical page, or NIL if there is none currently. The virtual page number returned can change unpredictably unless the page is wired down, or unless you assure that no interrupts (or consing of any kind, which can cause a swapout) occurs in the section of code using the value.

%page-frame-number (va)
Given a virtual address (VA), returns the logical page frame number (PFN) it currently resides in, or NIL if none. Use of this value is subject to the same restrictions as for the two preceding functions.

%add-page-device (real-unit start-block band-size)
This function is used by the CONFIGURE-PAGE-BANDS function to add a PAGE band to the virtual memory system. REAL-UNIT is the physical unit where the PAGE band resides. START-BLOCK is the PAGE band's first block number, and BAND-SIZE is its total size in blocks.

%findcore
Frees a page of physical memory (removes it from paging) and returns its logical page frame number (PFN).

%page-in (pfn vpn)
Creates a Page Hash Table (PHT) entry that indicates that virtual page number VPN is located in logical page frame number PFN. This had better be true or you'll be in trouble.

%page-trace
No longer implemented. Signals an illegal instruction error.

deallocate-swap-space (region)
Called by the garbage collector to free up any swap space on PAGE bands that is used by the virtual memory in REGION. REGION is an Oldspace region which the garbage collector has finished collecting, so that now its virtual memory (and any associated swap space) can be freed for later re-use.

%return-page-cluster (swap-device-number cluster-offset)
This is the routine called by DEALLOCATE-SWAP-SPACE to do the real work. SWAP-DEVICE-NUMBER is the logical page device number of a PAGE band and CLUSTER-OFFSET specifies the cluster number in that paging device to be freed.

%disk-address (va)
Used to find the disk address, if any associated with virtual address VA. This address may be on the LOD band or on any of the PAGE bands in the current configuration. Returns three values, if VA is a valid virtual memory address: the absolute disk block address of the page

containing VA, the physical disk unit, and the DPMT device status code for the page. See the constants in the %DPMTE-DEVICE-OFFSETS-FIELDS list for interpretation of this third value.

SECTION 7

Internal Storage Formats

7.1 INTRODUCTION

This section details the Explorer virtual machine internal storage formats. Symbolic byte specifiers and constants for most of the formats described can be found in the system parameters file SYS:UCODE;LROY-QCOM. The convention is that byte specifiers begin with a double percent (%%) while symbolic constants representing offsets or other numeric constants begin with a single percent (%). All such symbols are in the SYSTEM package, as are the majority of functions described here.

7.2 THE TAGGED ARCHITECTURE

Lisp follows a "single sized data" convention, which states:

Any object can be represented in a fixed size storage cell.

On the Explorer, the size of this fixed-size cell is one memory word (32 bits), also called a Q for quantum. A Q is divided into three fields: a 25-bit pointer field, a 5-bit data type field, and a 2-bit cdr code field. The format of a Q is shown in Figure 7-1.

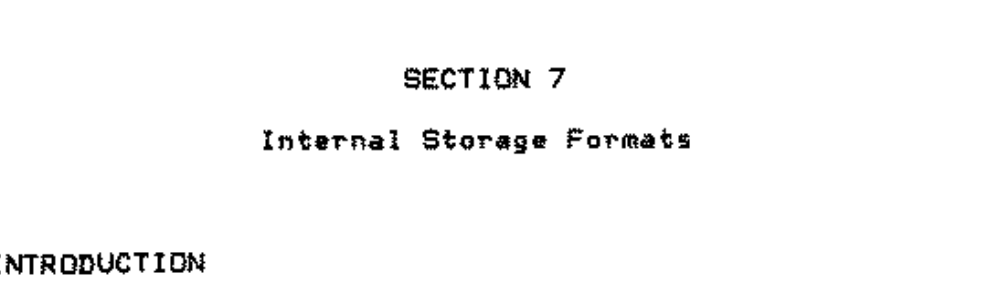


Figure 7-1 Q Format

Lisp can adhere to the single size data convention because some Lisp data items are not objects themselves but rather object references, which can be thought of as pointers to the actual

storage used by the object. The kind of object reference being made can always be determined by the data type field of the O. If it is an Immediate Type, such as a small integer, the actual data will be stored in the pointer field. No further reference need be made in this case. If the object is a Pointer Type requiring extended storage, such as an array or a list, the object's pointer field contains the first address of the extended storage.

There is an obvious analogy between object references and indirect addressing on conventional machines. However, rather than fully partitioning memory into different spaces where the different types of data objects are stored, the Explorer system uses the tagged data method of object representation. The way an object is manipulated is always dictated by the its data type tag. In turn the interpretation and use of the object's pointer field depends on the data type. There is considerable hardware-level support for this architecture in the Explorer processors themselves, allowing for extremely flexible data structuring combined with efficient access.

7.3 BOXED VERSUS UNBOXED MEMORY

Boxed or Typed cells are memory words whose data-type fields are valid; that is, they are meant to be interpreted as data types rather than as random bit patterns. Only if the data type field is valid may the pointer field and cdr code field be interpreted as pointers and cdr codes. Words whose contents are all data are termed Unboxed or Untyped. A segment of memory where all cells are boxed would be called fully boxed whereas a segment where some words are boxed and others are not is called partially boxed.

In the Explorer virtual machine, not all memory is fully boxed, although some well-defined segments are. Usually, determining if a given word is boxed or unboxed requires examining the word's context.

The first context item to consider is storage type. All virtual memory segments have a storage representation type of List or Structure. List space holds only CONS cells and cdr-coded lists, and is fully boxed. The start and end of list segments is determined by the cdr coding of the words. Structure space holds all other extended storage data structures and is not fully boxed. In structure space, the start of an object is marked by a structure header. Objects that are implemented as structures with headers are: symbols, instances, arrays, compiled functions (FEFs), and extended numbers. Of these, only symbols and instances are fully boxed. The others contain some boxed words and some unboxed words.

If a word is in structure space and if it is in a structure that is only partially boxed, further information must be sought to determine exactly which words are boxed. It is not even the case that a given structure type is always partially boxed or always fully boxed. For example, some arrays are completely typed while others have typed cells only up to the start of their actual data elements. Some extended numbers are made up of other extended numbers; hence are fully typed. Others contain bit-encodings following their headers.

The one unbreakable rule about partially boxed structures is that all boxed words must come before any unboxed storage.

For code that needs to know details about the size, boxed size, and unboxed size of data objects there are a number of functions that may be used. %STRUCTURE-TOTAL-SIZE, given a pointer to any boxed word in a structure, will return the total number of words of storage the structure uses. %STRUCTURE-BOXED-SIZE will likewise return the number of boxed words. The number of unboxed words in the structure can then be computed.

The largest drawback to these primitives is that they cannot be used on any arbitrary address. They will cause a crash if given an invalid address; they will be unpredictable (crash, trap) if given the address of an unboxed data word. For efficiency, they assume they have a good, boxed word to start with and go from there. Code that needs to find the (always boxed) start of the structure containing a given address should either begin parsing memory at the memory segment (region) beginning, or use a "safe" primitive such as %FIND-STRUCTURE-HEADER-SAFE (FSH-SAFE for short). This latter can be slow, since it parses memory in the forward direction, but makes every attempt to be safe. It will return NIL, for example, if given an invalid address.

7.4 DATA TYPE SUMMARY

The 5-bit data type field supports 32 data types. Some of these represent actual Lisp objects; that is, they can be passed as arguments to functions, returned as values, and have standard operations defined on them. These Lisp object data types comprise the primitive, architectural support out of which all more complex types are built.

The Lisp object types and some selected attributes are listed in Table 7-1.

Table 7-1 Lisp Object Data Type Attributes

Type Code	Data Type Name	Pointer Field Contents	Generic Type
5	DTP-Fix	Immediate data	Number
6	DTP-Character	"	Character
8	DTP-Short-Float	"	Number
1	DTP-List	Address	List
2	DTP-Stack-List	"	List
13	DTP-Closure	"	Function
14	DTP-Lexical-Closure	"	Function
3	DTP-Symbol	"	Symbol
9	DTP-Instance	"	Instance
4	DTP-Array	"	Array
16	DTP-Stack-Group	"	Stack Group
12	DTP-Function	"	Function
7	DTP-Single-Float	"	Number
10	DTP-Extended-Number	"	Number
11	DTP-Locative	"	Pointer
15	DTP-U-Entry	Index	Function

The remaining data types are termed housekeeping types because they are used in the virtual machine implementation rather than to represent computational objects. Examples of housekeeping uses include marking the start of extended structure storage or acting as invisible forwarding pointers.

The pointer field of a housekeeping type may contain an address or an index as with Lisp object types. If it is not an address it is usually divided up into various flag fields that describe the storage structure it is used in.

The housekeeping data types and some selected attributes are listed in Table 7-2.

Table 7-2 Housekeeping Data Types

Type Code	Data Type Name	Pointer Field Contents	Special Use
22	DTP-Symbol-Header	Address	Structure hdr
25	DTP-Instance-Header	"	"
24	DTP-Array-Header	Flags	"
26	DTP-Ref-Header	Flags	"
19	DTP-One-Q-Forward	Address	Forwarding
18	DTP-External-Value-Cell-Pointer	"	"
17	DTP-GC-Forward	"	"
28	DTP-GC-Young-Pointer	"	"
21	DTP-Body-Forward	"	"
20	DTP-Header-Forward	"	Forwarding, Structure hdr
27	DTP-Self-Ref-Pointer	Index, Flags	"
30	DTP-Null	Address	Unbound mark
0	DTP-Trap	Unused	Trap
31	DTP-Ones-Trap	Unused	Trap
29	DTP-Free	Unused	Illegal

The paragraphs below cover all the data types. First the Lisp object types are covered, divided into the immediate types and the pointer types. The pointer types are subdivided into List, Structure, and Miscellaneous types. The housekeeping data types used as structure headers are covered in the discussion of the extended storage object they implement. Finally, the remainder of the housekeeping types (the forwarding and miscellaneous housekeeping types) are detailed.

7.5 IMMEDIATE DATA TYPES

Some Lisp objects can be represented completely in one storage cell. These are called Immediate Types or INUM types, since the pointer field of such a cell is an actual value rather than a pointer to the actual value. INUM types are typically implemented for efficiency reasons. Each INUM type is discussed in the following paragraphs.

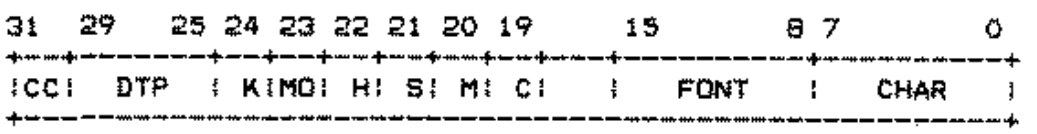
7.5.1 DTP-Fix.

This is a small integer, usually called a FIXNUM for fixed point number. The pointer field contains the actual value of the number in twos complement notation. Unlike extended numbers, FIXNUMs with the same value will always be EQ.

Integers in the range [- 2**24 ... 2**24 - 1] will be represented as FIXNUMs. Outside of this range an integer extended number (BIGNUM) will be created.

7.5.2 DTP-Character.

This is a Common Lisp character object. For Zetalisp compatability it can be used in arithmetic like a FIXNUM. Its format is shown in Figure 7-2. The byte descriptors can be found in Q-FIELDS.



CHAR	<0:7>	=	Character code or mouse click code
FONT	<8:15>	=	Font bits
C	<19>	=	Control Bit
M	<20>	=	Meta Bit
S	<21>	=	Super Bit
H	<22>	=	Hyper Bit
MO	<23>	=	Mouse Bit
KP	<24>	=	Keypad Bit

Figure 7-2 Character Format

7.5.3 DTP-Short-Float.

A short precision floating point number. The pointer field contains a 25-bit floating point number, subdivided into a 1-bit sign field, an 8-bit exponent field, and a 16-bit fraction field. The exponent field contains the excess 127 exponent, which gives a range of 10**-38 to 10***+38 approximately. The fraction field contains a 16-bit fraction, excluding the hidden bit (which is only used if the exponent field contains a non-zero value). The fraction can contain approximately 5 decimal digits. The format used is equivalent to the IEEE Std 754-1985 single precision format, with the exception that the fraction field has been

reduced from 23 bits to 16 bits.

The format of a short float word is shown later along with the other numeric formats.

7.6 POINTER OBJECT TYPES

The pointer type objects have an address (or an index) in their pointer field; the address generally points to the first word of the object's actual extended storage. Each pointer object points to a certain kind of storage. For structure-type pointers, the storage is always delineated by a structure header. For list-type pointers, the storage is a CONS or a list whose extent is defined by the local cdr coding. The pointer object types and what they point to are listed by category in Table 7-3 and are discussed individually below.

Table 7-3 Pointer Types

Structure Pointer Object:	Points to:
DTP-Symbol	DTP-Symbol-Header
DTP-Instance	DTP-Instance-Header
DTP-Array	DTP-Array-Header or DTP-Header-Forward
DTP-Stack-Group	DTP-Array-Header
DTP-Function	DTP-Fef-Header
DTP-Extended-Number	DTP-Header
DTP-Single-Float	DTP-Header

List Pointer Object:	Points to:
DTP-List	CONS cell, Cdr-Coded list, or DTP-Header-Forward
DTP-Closure	Cdr-Coded list
DTP-Lexical-Closure	CONS cell
DTP-Stack-List	Cdr-Coded list on a regular PDL

Other Pointer Object:	Points to:
DTP-Locative	Any boxed word
DTP-Self-Ref-Pointer	Indexes an instance cell

7.7 LIST POINTER OBJECTS

7.7.1 DTP-List.

The pointer field points to a CONS or a cdr-coded list segment. Whether the storage is a CONS or a list segment, and if a list segment how long a segment all depends on the cdr code field. The encoding of the 2-bit cdr code field is shown in Table 7-4.

Table 7-4 CDR Codes

0 - CDR NORMAL
1 - CDR ERROR
2 - CDR NIL
3 - CDR NEXT

7.7.2 CONS Cells.

A CONS is a 2-word structure whose first word is the CAR and second word is the CDR. This is the two-pointer form of CONS used in most versions of Lisp. The CAR cell always has a cdr code of CDR-NORMAL, and the CDR cell is marked by CDR-ERROR. The CDR-NORMAL means that the G following this one contains the CDR. CDR ERROR means that it is an error to take the CDR of this location, since this is the second half of a full (CDR-NORMAL) node.

7.7.3 Cdr-Coded Lists.

A scheme called Cdr-Coding allows a special, high-density storage scheme for regular lists. In such a storage scheme a list of N elements can be stored in N consecutive words using a series of CDR-NEXTs followed by a final CDR-NORMAL (if this list continues on with one or more CONS cells) or a CDR-NIL (if this is the end of the list). CDR-NIL means simply that the CDR of this node is the symbol NIL. In contrast, a list of N elements constructed from normal CONS cells requires 2N words of storage.

CDR-NEXT designates a list element (a CAR). It indicates that the CDR of this CAR is a (not physically present) pointer to the next word. When the CDR of a normal list is requested, a copy of the contents of the CDR cell is returned; when a CDR of a cdr-coded list is requested, a pointer to the location one word past the CAR is constructed and returned.

The functions APPEND, LIST, and COPY-LIST always form these compact cdr-coded lists while CONS and related operations always create full nodes (CDR-NORMAL, CDR-ERROR).

7.7.4 Destructive Operations: RPLACA, RPLACD.

RPLACA and RPLACD are easily defined on CONS cells. RPLACA overwrites the first cell of the CONS and RPLACD overwrites the second. Since all CARs in a cdr-coded list have cells allocated to them, to RPLACA an element of a CDR-NEXT list, you simply overwrite the specified CAR. But since the CDR node is not actually present, but rather is implied by the cdr-coding, RPLACDing is more difficult.

RPLACD first finds the CAR whose not-yet-existent cdr needs to be written. It then allocates a completely new CONS node (CDR-NORMAL, CDR-ERROR) and copies the CAR to the first word. It then writes the new CDR in the second word. Finally, the original CAR with the CDR-NEXT field is overwritten with a word with data type DTP-HEADER-FORWARD and cdr code CDR-ERROR, which points to the newly allocated CONS node. This process is called RPLACD forwarding.

7.7.5 DTP-Stack-List.

This is a cdr-coded list which has been created on a runtime stack (regular PDL), usually as a function's REST arg. The pointer field must always point to the portion of the stack that is active (i.e., before the top-of-stack pointer). Since runtime stacks are represented by special PDL arrays, a DTP-STACK-LIST paradoxically always points to structure space.

The elements of the STACK-LIST are always cdr-coded except that it may end with either a CDR-NIL or a CDR-NORMAL which points to a CONS in List space. RPLACA works normally on a STACK-LIST. It is an error to RPLACD one.

If an attempt is made to store a STACK-LIST into memory not in the active portion of the PDL, the list elements will be copied out to regular List space. All STACK-LIST pointers will then be replaced with normal LIST pointers, and the old elements on the stack will be replaced with words with data type DTP-EXTERNAL-VALUE-CELL-POINTER and pointer fields which address the corresponding element in the copied-out list.

7.7.6 Closures.

A dynamic closure is a word of type DTP-CLOSURE which points to a block of cdr-coded storage in list space. This block is 2*N+1 words long, where N is the number of cells closed over.

A lexical closure is a word of type DTP-LEXICAL-CLOSURE which points to a CONS cell. The CAR of the CONS is the closure's function, and the CDR is the a word of type DTP-LOCATIVE pointing to the closure's lexical environment.

The type DTP-LEXICAL-CLOSURE is only created by compiled code. The interpreter implements lexical closures as a DTP-CLOSURE over the special variables that hold the interpreter's environment.

Both these closure types are described in detail in a later section on Closures.

7.8 STRUCTURE POINTER OBJECTS

The are only a handful of actual structure types. These are: symbols, instances, arrays (including stack groups), functions, and extended numbers. Each of these is discussed here.

7.8.1 Symbols.

A symbol is represented by a Q of datatype DTP-SYMBOL whose pointer points to a five-word symbol block. The five words are listed in Table 7-5.

Table 7-5 Qs of Symbol

Offset	Cell Name
0	Print Name cell
1	Value Cell
2	Function Cell
3	Property Cell
4	Package Cell

The Print Name Cell holds a word of DTP-SYMBOL-HEADER pointing to a STRING array which is the print name for the symbol. The SYMBOL-HEADER acts just like an array pointer in many contexts.

The Value Cell holds the value of the symbol, and so can be of almost any data type. If the value has not been initialized, a symbol's value cell may be empty or unbound. If so, the cell contains a DTP-NULL whose pointer field points back at the symbol header.

The Function Cell holds the functional property of the symbol. If the symbol is called as a function, the contents of this cell will be analyzed to determine what function to perform. A

symbol's function cell may also be unbound, in which case it contains a DTP-NULL which points back at the symbol header.

The Property Cell contains the property list. The use of properties is not required by the basic system at all, so this might be NIL. On the other hand, many subsystems and features make heavy use of the property list, so it is likely to contain something.

The Package Cell is used to point to the package to which the symbol belongs for interned symbols; for uninterned symbols, the package cell contains NIL. The only architectural support for packages is the package cell of symbols.

When a symbol is initially created, the value and function cells contain DTP-NULL. The property cell initially contains NIL; however, the loader and other parts of the system that create symbols may place properties on them.

The function VALUE-CELL-LOCATION can be used to obtain a DTP-LOCATIVE pointer to this location and the contents can be obtained by (CAR <loc>) or more generally (CONTENTS <loc>) on the locative so generated.

7.8.2 Instances.

A flavor instance instance is a word of DTP-INSTANCE which points to a DTP-INSTANCE-HEADER. Instances are variable length, but all their elements (which are instance variable slots) are boxed. The INSTANCE-HEADER pointer field points to this instance's parent flavor. The flavor data structure is implemented as an array (so that the INSTANCE-HEADER acts as an array pointer in most contexts) and is described in greater detail in the section on Flavor Internals.

7.8.3 Arrays.

An array object is a word of DTP-ARRAY which points to a DTP-ARRAY-HEADER. The ARRAY-HEADER may optionally be preceded by a number of boxed leader words topped by a DTP-HEADER of header type ARRAY-LEADER. The ARRAY-HEADER may be followed by by some housekeeping words (if it is a long, multidimensional, displaced, or physical array) and then generally by the actual array data storage words. The ARRAY-HEADER pointer field does not contain an address, but rather several fields of data describing the array. The section on Arrays discusses array formats in greater detail. Below we briefly introduce some special arrays: Stack Groups, Regular PDL Arrays, and Binding PDL Arrays.

7.8.3.1 Stack Groups.

The state of a computation on the Explorer is maintained in a data structure called a Stack Group when the computation is not active. A stack group is represented by a word of DTP-STACK-GROUP pointing to an ARRAY-HEADER. The stack group array is a Q array of array type ART-STACK-GROUP-HEAD. It has no data elements; all the computation state is stored in array leader words. The stack group data structure itself is described fully in a later section.

In addition to saved register values and other state information, there are two arrays associated with each stack group: the Regular PDL and the Special PDL (SPDL). These are also described more fully in the sections on function calling and stack groups, but are introduced here to touch upon their particular storage management conventions.

7.8.3.2 Regular PDL Arrays.

A computation's run-time stack is kept in the stack group's Regular PDL array. A Regular PDL array has array type ART-REG-PDL. It is a Q array in that its valid elements can contain any Lisp object; but not all its elements are valid. Those which are not valid are considered as unboxed memory. A Regular PDL array always has a leader with one leader element. The leader element (element 0) contains the stack group which owns this PDL.

The top of the currently-active stack group's runtime stack is kept in the PDL Buffer of the processor. The hardware PDL Buffer acts as a cache of up to 1024 words which greatly speeds up almost all references to the stack. The PDL Buffer cache is maintained by microcode invisibly to macrocode and all higher levels.

The Regular PDL is not allowed to contain most housekeeping types. In addition, it always contains valid boxed Qs up to the current top-of-stack pointer. However, Regular PDL array elements beyond the current top-of-stack pointer are not guaranteed to be valid Lisp objects so must not be accessed by any Lisp code or be scavenged by the garbage collector. They are considered as unboxed elements in the array.

The top-of-stack pointer for a Regular PDL is computed in one of two ways. If the Regular PDL does not belong to the current computation (i.e., is not the Regular PDL of %CURRENT-STACK-GROUP), the valid portion of the PDL array is described by the saved regular PDL pointer of its Stack Group (SG-REGULAR-PDL-POINTER <sg>). Array elements numbering up to this value may be freely accessed. If, however, this is the current stack-group's Regular PDL, the highest valid element number must be computed using the hardware PDL-BUFFER-POINTER register.

In either case, the %STRUCTURE-BOXED-SIZE primitive will always return the correct number of valid, boxed words when given a

Regular PDL array. This number is the top-of-stack element index plus 2, since it counts the PDL array header word and the array long length word as valid, boxed words.

If a Regular PDL needs more elements than its initial allocation size the microcode traps out to the Error Handler. The Error Handler will allocate a larger array and structure-forward the old PDL to the new one.

7.8.3.3 Special PDL Arrays.

A computation's dynamic (special variable) binding stack is represented using a Special PDL array (SPDL) which has array type ART-SPECIAL-PDL. Like the Regular PDL array, the SPDL is a Q array in that its valid elements can contain any Lisp object; but not all its elements are valid. Those which are not valid are considered as unboxed memory. The SPDL also has one leader element which contains the stack group owning this SPDL.

The valid top-of-stack for the binding stack is stored in the stack group (SG-SPECIAL-PDL-POINTER <sg>) if this is not the current stack group's SPDL. Otherwise, as with Regular PDLs, the valid portion can only be computed using %STRUCTURE-BOXED-SIZE.

If a SPDL needs more elements than its initial allocation size the microcode traps out to the Error Handler which will allocate a larger array and structure-forward the old SPDL to the new one.

7.8.4 Compiled Functions (FEFs).

When a function is macro-compiled, the macrocompiler produces a compiled code object called a Function Entry Frame (FEF). This is represented by a word of DTP-FUNCTION pointing to storage starting with a word of type DTP-FEF-HEADER. The storage occupied by the FEF can be divided into two parts. The first part, which is at least eight words long, is called the overhead section and consists only of boxed words. This section contains encoded information about the function and pointers to variables and other functions called by the function. The second section is completely unboxed and contains the function's macrocode instructions packed two to a 32-bit memory word.

See the section on Function Calling for more details on FEFs.

7.8.5 DTP-Header.

This word is the beginning of a block of storage which is either an array leader, a single float, or and extended number. The pointer field does not contain an address; instead it has a HEADER-TYPE field which explains what purpose the header is serving. The HEADER-TYPEs are summarized in Table 7-6. Array

leaders are discussed in the Arrays section. The remaining extended number structures are described below.

Table 7-6 Header Types

Code	Header Type Name	Pointed To By
0	%HEADER-TYPE-ERROR	Unused
1	%HEADER-TYPE-UNUSED-1	Unused
2	%HEADER-TYPE-ARRAY-LEADER	DTP-LOCATIVE
3	%HEADER-TYPE-UNUSED-3	Unused
4	%HEADER-TYPE-SINGLE-FLOAT	DTP-SINGLE-FLOAT
5	%HEADER-TYPE-COMPLEX	DTP-EXTENDED-NUMBER
6	%HEADER-TYPE-BIGNUM	DTP-EXTENDED-NUMBER
7	%HEADER-TYPE-RATIONAL	DTP-EXTENDED-NUMBER
8	%HEADER-TYPE-DOUBLE-FLOAT	DTP-EXTENDED-NUMBER

7.8.6 DTP-Extended-Number.

An word of type DTP-EXTENDED-NUMBER pointing to a DTP-HEADER word signifies one of the following, depending on the HEADER-TYPE of the HEADER word: an extended integer (BIGNUM); a ratio between two integers (RATIONAL); a double-precision floating point number (DOUBLE-FLOAT); or a complex number (COMPLEX). All except DOUBLE-FLOAT are described here. The floating point formats are detailed in the next subsection.

7.8.6.1 BIGNUM.

A BIGNUM is an extended precision integer represented by an object of type DTP-EXTENDED-NUMBER pointing to a DTP-HEADER. The storage length of a BIGNUM is determined by the size of the integer it represents. It is composed by a HEADER word and a number of unboxed data words as shown in Figure 7-3. After the BIGNUM header, the integer is stored in successive words with the least significant word first.

The format of the BIGNUM header is shown in Figure 7-4. The LENGTH field gives the length of the BIGNUM in words; this is the length of the BIGNUM structure minus one. Each of the BIGNUM data words has the format shown in Figure 7-5. The high order bit is always 0. The remaining bits are a section of the bits of the positive integer that is represented.

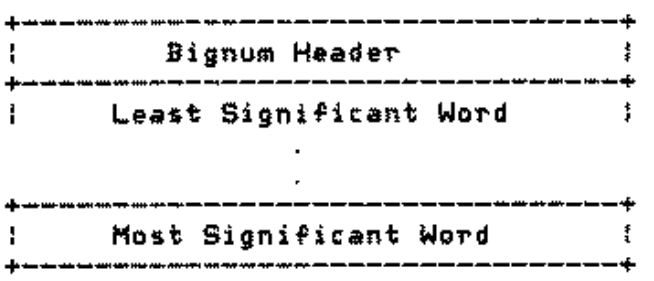


Figure 7-3 BIGNUM Structure

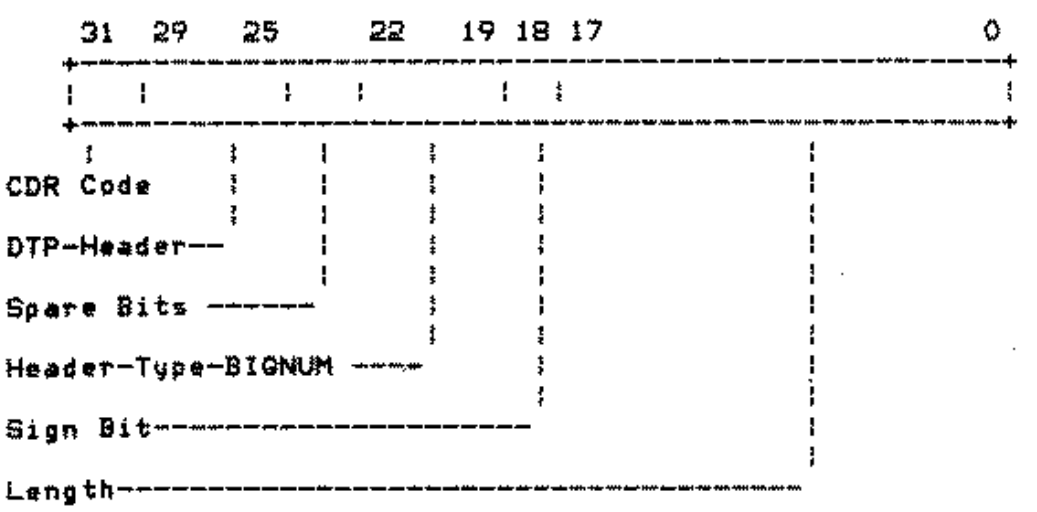


Figure 7-4 BIGNUM Header Format

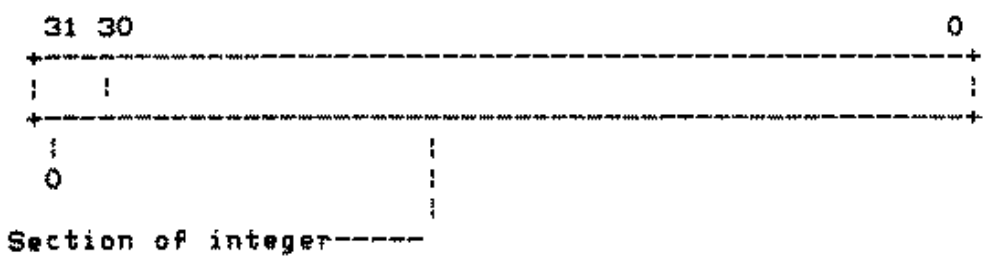


Figure 7-5 BIGNUM Data Format

7.8.6.2 RATIONAL.

A RATIONAL number is an object of type DTP-EXTENDED number pointing to a 3-word fully boxed structure. The first word is of type DTP-HEADER and the next two words are the numerator and the denominator, respectively, of the ratio. Each of these may either be a FIXNUM or a BIGNUM.

7.8.6.3 COMPLEX.

A COMPLEX number is an object of type DTP-EXTENDED number pointing to a 3-word fully boxed structure. The first word is of type DTP-HEADER and the next two words are the real part and the imaginary part, respectively, of the COMPLEX number. Each of these may be any numeric type.

7.8.7 Floating Point Numbers.

There are several floating point formats supported on the Explorer. Short precision floating point numbers are an INUM type; that is, the pointer field of the Q contains the value of the object rather than a pointer to it value. Single precision and double precision floating point numbers are represented as a pointer pointing to a DTP-HEADER with header type %HEADER-TYPE-SINGLE-FLOAT and %HEADER-TYPE-DOUBLE-FLOAT respectively. Single precision floating point numbers have a pointer type DTP-SINGLE-FLOAT; double precision floating point numbers have a pointer type DTP-EXTENDED-NUMBER.

7.8.7.1 Short Float.

The format of a short precision floating point number is shown in Figure 7-6. This format is equivalent to the single precision floating point format, with the fraction field reduced to 16 bits.

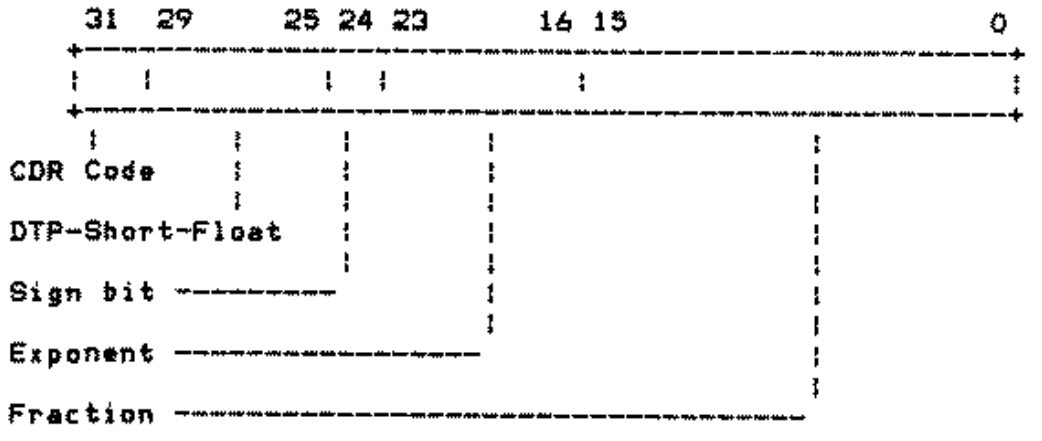


Figure 7-6 Short Float Format

7.8.7.2 Single Float.

A single precision floating point number is represented as an object of DTP-SINGLE-FLOAT, pointing to a two-word structure as shown in Figure 7-7. The second word is unboxed.

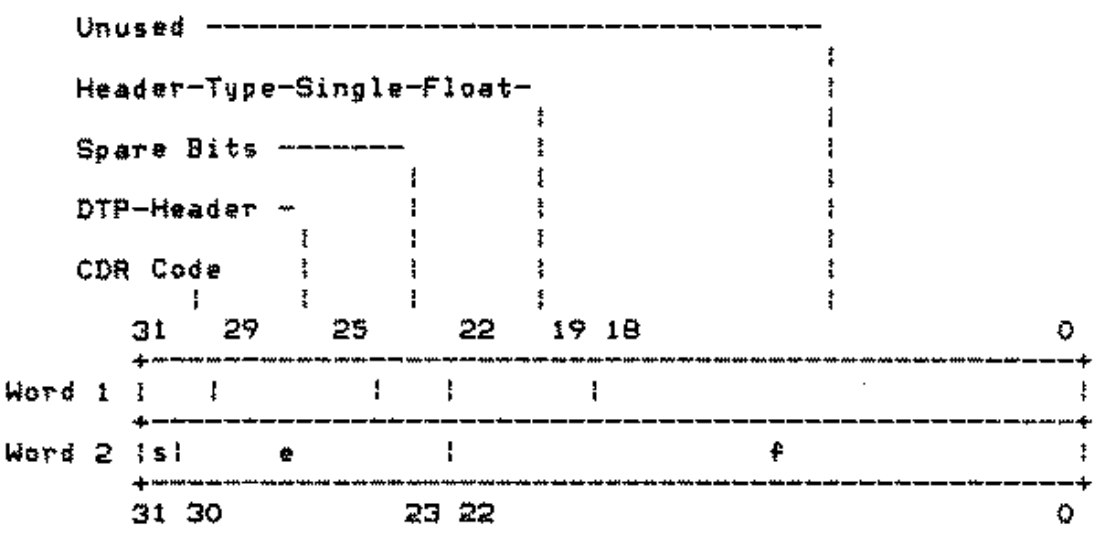


Figure 7-7 Single Precision Float Structure Format

The format used is the IEEE Std 754-1985 single format. The 8-bit exponent field uses an excess-127 (decimal) format. The single precision floating point format uses a 23-bit fraction field. The range for single precision floating point numbers is approximately 10**-38 to 10**+38, with a precision of approximately 7 decimal digits.

7.8.7.3 Double Float.

A double precision floating point number is represented as an object of DTP-EXTENDED-NUMBER, pointing to a three-word structure as shown in Figure 7-8. The two words following the HEADER are unboxed.

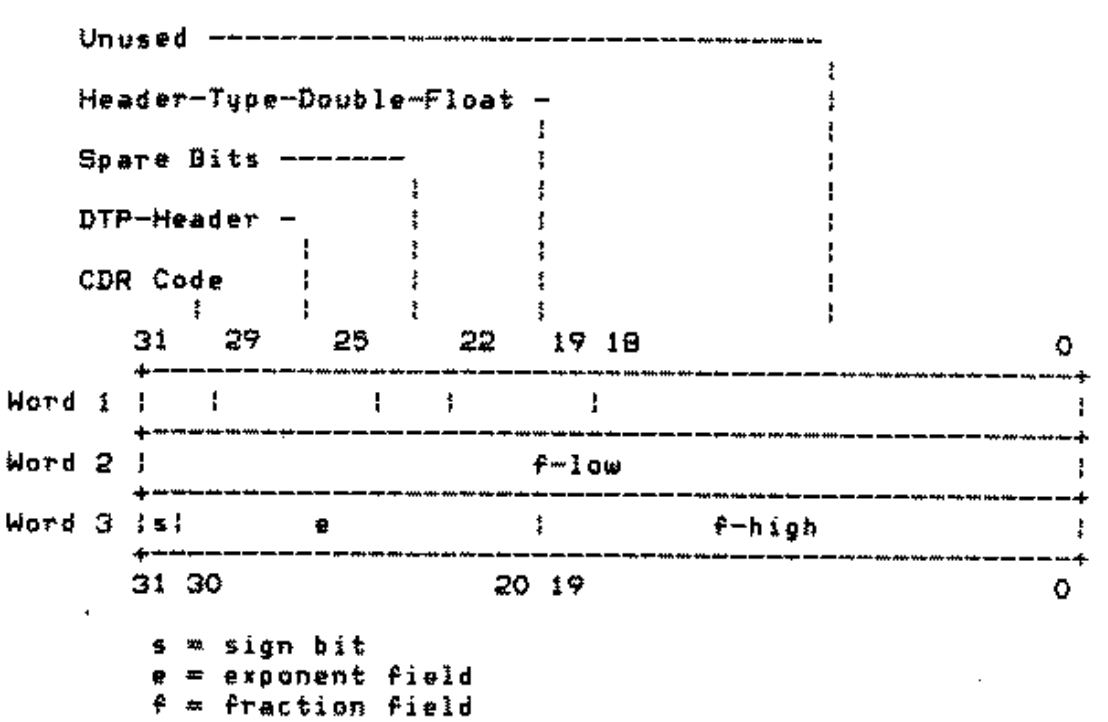


Figure 7-8 Double Precision Float Structure Format

The format used is the IEEE Std 754-1985 double format. The 11-bit exponent field uses an excess-1023 (decimal) format. The double precision floating point format uses a 52-bit fraction field, with the 20 most significant bits concatenated with the sign and exponent fields. The range for double precision floating point numbers is approximately 10**-308 to 10**+308, with a precision of approximately 16 decimal digits.

7.8.8 The Cdr Code Field in Structures.

The cdr code field not usually significant in structure space. There are, however some exceptions. The cdr code field is used to denote actual cdr coding in structures that are treated as lists (a STACK-LIST or ART-Q-LIST array). In addition, the cdr code has a specialized meaning relating to binding blocks inside

of a Special (Binding) PDL array (see the section on Stack Groups for more details). Finally, hash table entries are cdr coded so the key and value(s) may easily be returned as a list.

7.9 OTHER POINTER OBJECTS

Two miscellaneous pointer objects are implemented with the data types DTP-LOCATIVE and DTP-U-Entry. The LOCATIVE is generally used to point to a single cell. A U-ENTRY is a functional object with an index in its pointer field.

7.9.1 DTP-Locative.

A locative is a general-purpose pointer to a single boxed cell of memory. It is not an invisible pointer type. It can be used for many things. For example, it is often used to point to an array leader's header word, or to point to bound cells on the binding PDL. Both CAR and CDR of a locative return the same thing, namely the contents of the cell pointed at.

7.9.2 DTP-U-Entry.

This data type represents a microcoded function that takes a variable number of arguments or has a REST arg. It is a legitimate functional object which can be funcalled, passed as an argument, and so forth. AREF and LIST are examples of U-ENTRYs.

The term U-ENTRY is short for Ucode, or microcode, entry. The pointer field is actually an index into the MICRO-CODE-ENTRY-AREA; this contains either a FIXNUM or a function. If it is a FIXNUM, that number is an index into the MICRO-CODE-LINK-AREA. The FIXNUM then found in the MICRO-CODE-LINK-AREA consists of the 16-bit control store address of the microcode to run for this function; 6 bits describing the required arguments pattern; and one bit indicating a REST arg or not.

If the entry in the MICRO-CODE-ENTRY-AREA is not a FIXNUM, then the current definition of this function is not microcoded and the symbol names the function to run

The DEBUG-INFO structure for a U-ENTRY function can be found by using the pointer field as an index into the MICRO-CODE-ENTRY-DEBUG-INFO-AREA.

7.10 HOUSEKEEPING TYPES

The discussion of housekeeping types in the following paragraphs begins with the forwarding pointer types. Then the remainder of the data types are described: the DTP-SELF-REF-POINTER type; the Trap Types DTP-NULL, DTP-TRAP; and the type DTP-FREE.

7.10.1 Forwarding Pointer Types.

Forwarding pointer types are "invisible" data types which provide data indirection. They are termed invisible because their presence is completely transparent to most Lisp code. Whenever a forwarding pointer is read, the read is indirected along the invisible pointer. This is sometimes called the INVIZ process, or just INVZ. INVZ is similar to indirect addressing in other computers, except that instead of being specified by the reading instruction, the indirection is specified by the data read. For example, if you ask for the symbol value of a symbol with a DTP-ONE-Q-FORWARD Q in its value cell, what you will actually get back is the contents of the word at the address specified in the pointer field of the DTP-ONE-Q-FORWARD word.

There are six forwarding data types. DTP-ONE-Q-FORWARD and DTP-EXTERNAL-VALUE-CELL-POINTER are one-word forwards used for various housekeeping functions. DTP-GC-FORWARD and DTP-GC-YOUNG-POINTER are used for garbage collection support. DTP-HEADER-FORWARD and DTP-BODY-FORWARD are used to forward a whole structure that has grown or been moved. All are discussed below.

7.10.1.1 DTP-One-Q-Forward.

This is a simple kind of invisible pointer used to hide a single cell of memory. It forwards only the word that it is in, not the whole containing structure. Its most common use is to alias a symbol's value to that of another symbol (see DEFF and FORWARD-VALUE-CELL).

7.10.1.2 DTP-External-Value-Cell-Pointer.

This is a kind of one-word forwarding invisible pointer used for several different purposes. They are most commonly found in the overhead words of compiled functions (FEFs). There they are used to point to value cells of symbols referenced by the code and to function cells of other functions called.

EVCPs are also placed in symbol value cells by the dynamic closure mechanism (see the section on Closures), and are used to replace STACK-LIST elements on the PDL after the list is copied out to normal memory (see the paragraph on Stack Lists). The

Lisp interpreter also uses EVCPs to implement dynamic binding.

7.10.1.3 GC Forwarding Pointers.

Data type DTP-GC-Forward is the forwarding pointer left behind by the garbage collector in Oldspace; in fact, it is illegal anywhere else but Oldspace. The pointer field of a DTP-GC-FORWARD must point to Copyspace, and contains the address where this cell is forwarded. In this sense the GC-FORWARD is a one-word forward.

Normally, whenever a pointer-type Q is read from memory a check is done to see if the address in the pointer field falls in Oldspace. If so, the object pointed to is copied to Copyspace and the object's old location is filled completely with GC-FORWARDS. This is called transporting. Furthermore, the original memory location read is updated so that it now has the Copyspace address. When other referenced Qs are found to point to this GC-FORWARD they are also updated in memory to have the forwarded, Copyspace address. This process is called snapping out, and preserves the identity of objects copied by the garbage collector.

The type DTP-GC-Young-Pointer (GCYP) is a single-cell forwarding pointer which is used in implementing the Temporal Garbage Collection (TGC) algorithm. It acts as a special marker indicating that the object it replaces is younger than its containing structure.

The GCYP pointer field always addresses an indirection cell in the special INDIRECTION-CELL-AREA. The indirection cell contains the actual object the GCYP replaces, including the cdr code field.

7.10.2 Structure Forwarding.

When an array needs to grow beyond its original allocation due to VECTOR-PUSH-EXTEND or ADJUST-ARRAY-SIZE operations, a process known as structure forwarding takes place. This is done by the STRUCTURE-FORWARD function. New, larger contiguous storage is allocated, all the array's leader and data element words are copied into the new storage, and the old structure is filled with forwarding markers.

The old array header word is replaced by a word of type DTP-HEADER-FORWARD whose pointer field has the address of the new array header word. All other words in the old structure, both leader words and data element words, are replaced with Qs containing data type DTP-BODY-FORWARD and pointing to the HEADER-

FORWARD word. The body forwards are needed to forward array elements that have pointers to them (created, for example, by LOCF), but are used even in unboxed arrays.

To follow a BODY-FORWARD, the pointer field is used to find the HEADER-FORWARD word. The offset (either positive or negative) from the HEADER-FORWARD word is calculated. That offset is then applied to the address of the real header, found after following all intermediate HEADER-FORWARDs have been followed.

It is possible for symbols to be structure forwarded also, but this is no longer done by any system code. No other structures in structure space can be structure forwarded. DTP-HEADER-FORWARD does have a further, different use in List space to support the RPLACD operation. This is described above in the paragraphs on Destructive List Operations.

7.10.3 DTP-Self-Ref-Pointer.

A SELF-REF-POINTER (SRP) is used for mapped and unmapped instance variable references and may be used to implement monitor variables in the future. The pointer field of an SRP contains flag bits and an index field. The format of a DTP-SELF-REF-POINTER (SRP) is shown in Table 7-7. SRPs are created and manipulated within the code for flavors, mostly in the mapping-table sections. They are currently found only in the typed overhead words of FEFs.

Table 7-7 SELF-REF-POINTER Format

Bit 19:	SELF-REF-RELOCATE-FLAG
Bit 18:	SELF-REF-MAP-LEADER-FLAG
Bit 17:	SELF-REF-MONITOR-FLAG
Bits 12-0:	SELF-REF-INDEX
Bits 12-1:	SELF-REF-WORD-INDEX

If RELOCATE-FLAG is clear, an SRP is an unmapped instance variable reference. The instance variable address is derived by indexing SELF-REF-INDEX words into the instance data structure currently associated with SELF. Unmapped instance variables are created by the :ORDERED-INSTANCE-VARIABLES option to DEFFLAVOR, and when a flavor is a base flavor with no mixins, required flavors, and so forth.

If RELOCATE-FLAG is set, the SRP is a mapped instance variable reference and a SELF-MAPPING-TABLE array must be used. This is the most common case (about 75% of SRPs). When the MAP-LEADER-FLAG is clear, the INDEX field is used as an element index into the current SELF-MAPPING-TABLE array (an ART-16b array). That

location should contain a number which is the real index into SELF of the mapped instance variable.

The MAP-LEADER-FLAG, when set, means to read the contents of a slot in the array leader of the SELF-MAPPING-TABLE. This flag is used only when fetching another mapping table during the execution of a :COMBINED method built on composed flavors (about 5% of SRPs). The SELF-MAPPING-TABLE is obtained from Local Slot 1 on the stack. The INDEX field is then an index into the mapping table array leader, which should then contain a locative to a cell containing the instance variable value.

The MONITOR-FLAG bit is intended for use in implementing "monitor variables" which cause a trap when written into; however monitor variables are not currently supported hence the Monitor-Flag bit should always be a 0 in an SRP. If the MONITOR-FLAG bit were set, the SRP would be a reference to the next memory location on read and would cause a trap on write. The MAP-LEADER-FLAG would be ignored. No monitor pointers could appear within methods.

7.10.4 Trap Types.

Reading a word with a trap data type will normally cause the error handler to be invoked (via the trans trap mechanism described in the section on Error Handling).

7.10.4.1 DTP-Null.

This datatype is used for various things to mean "nothing." Its most common use is to act as an unbound cell marker. For example, an unbound symbol has DTP-NULL in its value cell. The pointer field points back at the symbol header so that the error handler can describe the symbol that is unbound. DTP-NULL can also act as an unbound marker in other structures; but the pointer field must always point to some structure's header in order support the error handler.

DTP-NULL is also used in hash tables to mark unused entries. Here, in a special case of the pointer field restriction, the pointer field contains 0, which is the address of NIL's header.

7.10.4.2 DTP-Trap and DTP-Ones-Trap.

These data types are present mainly for error checking. A newly created virtual page is filled with words of type DTP-TRAP and pointer fields of 0 by the page-fault microcode (in fact, all 32 bits are 0). The DTP-ONES-TRAP data type is used as a means of easily detecting sign extension during arithmetic operations.

7.10.4.3 Unused Type DTP-Free.

Some virtual memory that is allocated by the system build program (GENASYS) but is not yet occupied by objects has data type DTP-FREE. It is illegal in any context (will cause crash processing to occur). It should be considered reserved for future use.

SECTION 8

Arrays

8.1 ARRAYS

An array consists of a group of elements, each of which contains a data item. The individual elements are selected by numerical subscripts. The rank of an array is the number of subscripts used to refer to the elements of the array. The rank may be any integer from zero to seven, inclusive.

The lowest value for any subscript is zero; the highest value is a property of the array. Each dimension has a size, which is the lowest number which is too great to be used as a subscript. For example, for a one-dimensional array of five elements, the size of the one and only dimension is five, and the acceptable values of the subscript are the integers zero through four.

Any array may have an array leader. An array leader is like a one-dimensional ART-Q array which is prepended to the main array. Hence an array which has a leader acts like two arrays joined together except that the indexing scheme is different for array leaders. The array leader words can be stored into and examined by special accessors, different from those used for the main array. The leader is always fully boxed; leader words can always hold any kind of Lisp object, regardless of the type or rank of the main part of the array.

Many high-level Explorer data types are implemented using arrays. These include hash tables, flavor data structures, and other structures created by DEFSTRUCT.

An array object is represented as a word of DTP-ARRAY. The pointer field point to an array header word of data type DTP-ARRAY-HEADER. The array may also indirectly point to its ARRAY-HEADER through an intermediate forwarding structure. An array object pointing to a a DTP-HEADER-FORWARD indicates that the array has been structure-forwarded. The HEADER-FORWARD pointer field will then point to an ARRAY-HEADER or to another HEADER-FORWARD. The structure forwarding process, which occurs when an array needs to be grown larger than its original size, is described in the Internal Storage Formats section.

The format of an ARRAY-HEADER is shown in Figure 8-1. These fields are discussed in subsequent subsections.

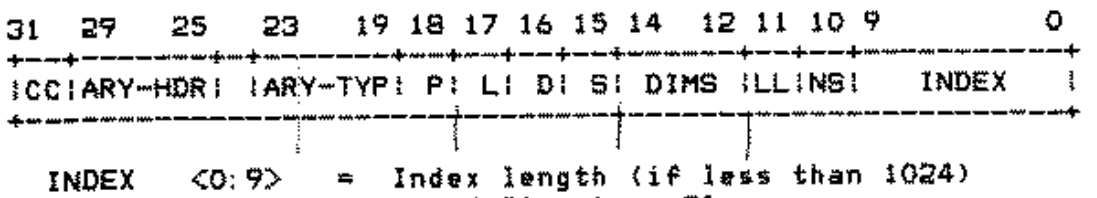


Figure 8-1 Array Header Word

8.1.1 Array Type.

The array type field indicates the type of the array. The array type determines the type of data that may be stored in the array and how this how this data is accessed.

There are many types of arrays. Some types of arrays can hold Lisp objects. These are called Q arrays. In a Q Array each memory word contains a Lisp object; hence all memory words in a Q array are boxed. Most of the Q arrays allow their cells to hold arbitrary Lisp objects, although some numeric arrays limit element contents to various numeric types.

Other types of arrays can only contain integers stored as raw bit patterns and often packed several to a 32-bit memory word. These are the bit arrays, whose data portions are unboxed. Some numeric arrays are also unboxed. Bit array elements (those which are smaller than 32 bits) are stored right-to-left within each word; the first element of an ART-4B ARRAY, for example, would be stored right-justified, beginning at bit 0 of the 32-bit word.

The array types are known by a set of symbols whose names begin with "ART-" (for ARray Type). They are summarized in Table 8-1 and are also discussed in the Explorer Lisp Reference manual. The symbol ARRAY-TYPES contains a list of all array types.

Table 8-1 Array Types

Code	Type	Boxed?	Bits/Element
0	ART-ERROR	N/A	N/A
1	ART-1B	No	1
2	ART-2B	No	2
3	ART-4B	No	4
4	ART-8B	No	8
5	ART-16B	No	16
6	ART-32B	Yes	32
7	ART-Q	Yes	32
8	ART-Q-LIST	Yes	32
9	ART-STRING	No	8
10	ART-STACK-GROUP-HEAD	Yes	32
11	ART-SPECIAL-PDL	Yes	32
12	ART-HALF-FIX	No	16
13	ART-REG-PDL	Yes	32
14	ART-DOUBLE-FLOAT	No	64 *
15	ART-SINGLE-FLOAT	No	32
16	ART-FAT-STRING	No	16
17	ART-COMPLEX-DOUBLE-FLOAT	No	128 *
18	ART-COMPLEX	No	64 *
19	ART-COMPLEX-SINGLE-FLOAT	No	64 *
20	ART-FIX	Yes	32

* These types require 2 or 4 words per element.

8.1.2 Array Leaders.

The array may optionally have an array leader which consists of a number of words BEFORE the array header. If the Has Leader Bit is set in the array header word, there is a leader present.

If there is a leader then the word immediately before the array header is a FIXNUM holding the number of array leader elements. Before that are the array leader elements, which may have any data type since any object can be stored in them. Finally, preceding the leader elements is a word of data type DTP-HEADER and header type %HEADER-TYPE-ARRAY-LEADER. The rest of the leader header word contains the total number of words in the leader (including the leader header and number-of-leader-elements words). The presence of the leader header is necessary for routines such as the garbage collecter which scan through memory in the forward direction. Note that leader elements are indexed backwards from the array header. The storage layout of an array with leader is shown in Figure 8-2.

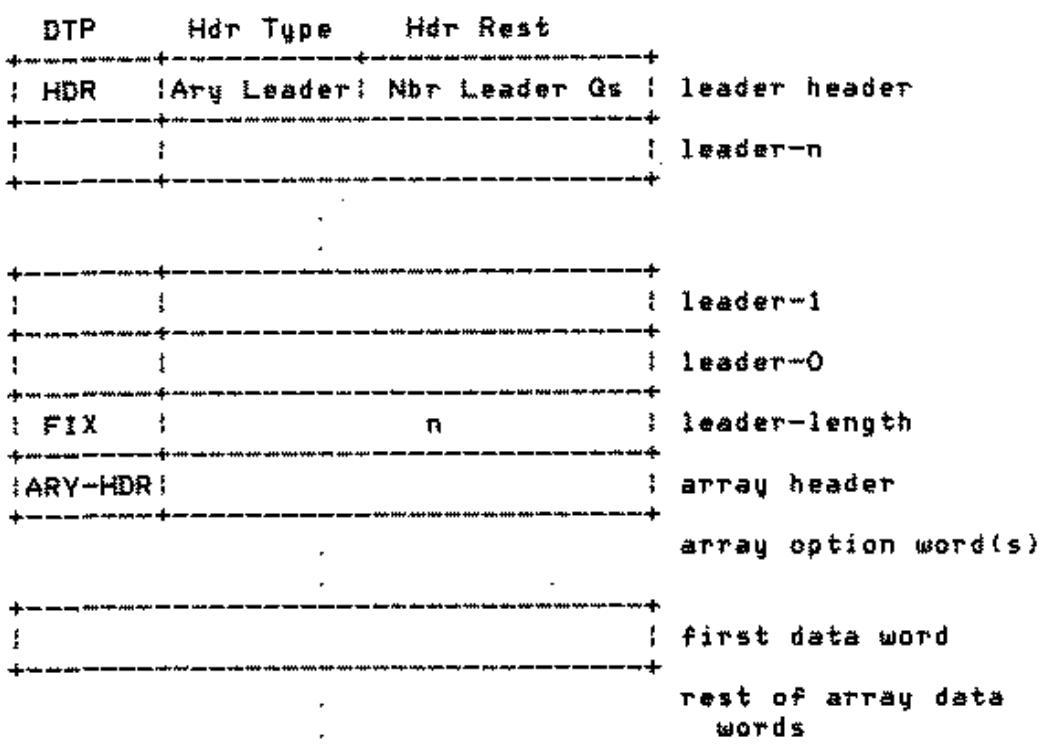


Figure 8-2 Array with Leader

8.1.3 Simple Bit.

The Simple Bit indicates that this is a simple array which can be accessed more efficiently. Simple arrays are one dimensional arrays which may or may not have leaders. They cannot be displaced, physical or long length.

Note that here the term simple is used only to distinguish a type of access that may be done by the virtual machine. It does not have same meaning as the Common Lisp data type SIMPLE-ARRAY.

8.1.4 Named Structure Flag.

The Named Structure FLAG is 1 to indicate that this array is an instance of a Named-Structure (probably defined with DEFSTRUCT with the :NAMED-STRUCTURE option). The structure name is found in array leader element 1 if the Has Leader Bit is set; otherwise it is in array element 0.

Named structures may be viewed as implementing a sort of user defined data typing facility. Certain system primitives, if handed a named structure, will obtain the name and obtain from that a function to apply, to perform the primitive.

8.1.5 Index Length Field.

The index length of an array is its number of data elements minus one. In a one dimensional array, it is the maximum value the index may take. In a multidimensional array, it is the product of the sizes of each of the dimensions.

If the index length of an array is larger than will fit in the index length field in the header, the Long Length Flag is set and the index length is stored in the next memory word.

8.1.6 Number of Dimensions Field.

The number of dimensions (rank) of the array is stored in the Number of Dimensions field. This can be a value from 0 to 7 inclusive. Zero-dimension arrays are defined to have exactly one element. Therefore, all zero dimension arrays will have a zero, but will have one element.

Elements are stored in multidimensional arrays in row-major order. An array cannot change its number of dimensions when it is grown. Each dimension after one will cause a dimension index word to be allocated.

8.1.7 Displaced Bit.

A displaced array is an array that has all its data elements in a separate, non-contiguous area of memory. A displaced array may be displaced to another array or to an arbitrary address (specified a LOCATIVE or a FIXNUM). Thus, a displaced array can be used to point at the beginning of a REGION. This is done, for example by the #'REGION-BITS array and others like it which represent the special areas/regions in low virtual memory. They are ART-Q-LIST arrays displaced to the address which is the origin of the region they represent.

Arrays displaced to a virtual or physical address are called displaced-to-address arrays. In contrast, if the array is displaced to another array, it is known as an indirect array, and the array it is displaced-to is called the indirected-to array or just an indirected array. The indirected-to array has an index length of its own. Then the index length of the indirect array appears to be MIN(x,y) where x is the index length of the indirected-to array and y is the total number of elements in the indirect array. Computing MIN prevents referencing beyond the

actual end of the indirected-to array.

A displaced index offset is used as to skip over a number of data slots in the indirected-to array or in the displaced-to-address area before element access begins. This is only tricky with an indirect array displaced to an indirected-to array. In this case, whenever the indirected-to array is referenced, it is as if that array were being referenced with an index N higher. And this index offset N is expressed in terms of the element size specified in the indirect array (not in elements of the indirected-to array). In other words, the storage of the indirected-to array is treated as if it had the same element size as the indirect array, even if the two element sizes are in fact different. Proper data alignment becomes an important consideration when using displaced index offsets.

The displaced index offset is always considered to be one dimensional; it is added after all the dimensions have been multiplied out. The resulting index is checked against the computed index length of the indirected-to array, if present.

The Displaced Bit is set in all displaced arrays. Such an array will always allocate one array option word to describe the storage displaced to, and may also allocate a displaced index offset word.

8.1.8 Physical Bit.

If this is a physical array, then both the Physical Bit and the Displaced Bit are set. Physical arrays are displaced arrays which are displaced to a NuBus physical address. Physical arrays may only be made from the bit-array types.

A physical array will always allocate one array option word to describe the physical address displaced to and may also allocate a displaced index offset word.

8.1.9 Option Words.

Between the ARRAY-HEADER word and the start of the array data elements there may be a number of array option words. If present they will occur in the following order. Each option word is discussed separately.

Non-Displaced Arrays:
- o Long Length word
- o Dimension word(s)

Displaced Arrays:
- o Dimension word(s)
- o Displaced-To word
- o Index length word
- o Displaced Index Offset word

8.1.9.1 Long Length Word.

If the total number of array elements is too big for the header index field (1024 elements or larger) the index length is stored in a long length word as a FIXNUM (limiting the maximum array size 2**24 - 1 elements). In non-displaced arrays, the long length word always follows immediately after the header word and before any dimension words.

8.1.9.2 Dimension Words.

If the array has more than one dimension, then there is a block of <number of dims minus one> words immediately after any long length word. Each dimension word holds the size of one dimension. The first dimension word contains the size of the most rapidly varying subscript (that of the last dimension in the dimension list); the second dimension word contains the size of the second most rapidly varying subscript (that of the penultimate dimension in the dimension list); and so forth until n-1 dimension words. The size of the last dimension can be computed from the N-1 dimension sizes along with the total index length in the index length field or the long length word.

8.1.9.3 Displaced Array Option Words.

In a multidimensional displaced array, the N-1 dimension words always come first, immediately after the array header. What follows is another group of 2 or 3 option words consisting of a displaced-to word, an index length word, and possibly a displaced-index-offset word. The displaced-to word is an array, locative or integer for regular displaced arrays. It and any further option words following it are considered boxed. On physical arrays, however, the displaced-to word is a 32-bit address; hence it and any words following it must be considered unboxed.

Displaced arrays must keep track of two different array sizes: their own (for computing how much actual storage this displaced array "stub" takes); and the number of elements in the displaced-to storage area. To accomodate the latter, an index length word is allocated after the displaced-to word.

The index field in the ARRAY-HEADER of the displaced array itself is used to contain the number of option words, not counting the dimension words, that this displaced array has. Thus it will always be either a FIXNUM 2 or 3.

If there is a displaced index offset, its FIXNUM word will be placed last.

SECTION 9

Storage Management

9.1 INTRODUCTION

This section discusses Explorer storage management in Areas and Regions. Garbage Collection (GC) and its special spaces are covered in the section on Garbage Collection.

Storage allocation for Lisp objects is implemented on top of a large uniform address space provided by the Virtual Memory System. The collection of all Lisp objects is known as the Lisp Object Space. Storage Allocation and Garbage Collection manage the mapping of Lisp Object Space to the virtual address space. Both storage allocation and garbage collection are logically above the virtual memory system. The virtual memory system does not understand and therefore cannot assist in a meaningful way in the allocation of address space to Lisp objects.

9.1.1 Areas and Regions.

The storage allocation system manages the address space by breaking it down into smaller segments called regions. A region is the smallest quantum of address space managed by the address space management system. Regions in turn belong to areas.

Areas are created by explicit commands (see MAKE-AREA). Area creation merely defines the abstract area entity, whose regions are intended to contain objects related in some sense (such as all local objects used by a process). One initial region is created when the area is first made.

9.1.2 Address Space Allocation and Use. Virtual address space is allocated to regions when they are created. The allocation a region is defined by the region's origin (starting virtual address) and its length. Actual occupation (use) of virtual address space by objects takes place in regions as storage requests are made by the various storage allocation primitives. The general term consing is used to refer to any such allocation of storage, whether to actual cons cells or to other objects.

Storage is assigned to objects in a linear fashion starting at the origin address of a region. The subset of region locations occupied by objects is defined by the region's free pointer, which is the next offset at which an object can be given storage. All locations before this pointer have been used.

9.2 AREAS

The logical address space is divided into areas. An area defines a set of attributes on the virtual address space that it contains. While an area doesn't really have any virtual address space assigned directly to it, it does contain one or more regions which do. An area is identified by its area number, an integer between 0 and 255. The area number is used as an index into the various area descriptor tables described in Table 9-1 to obtain the appropriate characteristic of the area.

Table 9-1 Area Attributes

AREA-NAME	A symbol representing the name of the area.
AREA-REGION-LIST	The region number of the first region in this area.
AREA-REGION-BITS	A template for the REGION-BITS word for a region allocated in this area.
AREA-REGION-SIZE	The default size of this area's oldest regions.
AREA-MAXIMUM-SIZE	The maximum size this area is allowed to occupy or *MOST-POSITIVE-FIXNUM* if unlimited.

A further series of attributes are defined by the AREA-REGION-BITS word. When a new region is created it inherits the attributes of the area to which it belongs by using the AREA-REGION-BITS word as a template. These attributes will be described in detail in the paragraph on regions.

Logically, the area descriptor table is a table of 5-word entries as shown above. In reality, the table exists as five separate tables inuexed by the area number, each table corresponding to one of the five words. This makes it easy for the microcode to index into the table and makes the code fairly insensitive to changes in the entry size.

Areas can be created by user commands. The area in which general consing occurs can also be controlled from Lisp (see the Storage Management chapter of the Explorer Lisp Reference manual). A

program may use this feature to allocate related items in a contiguous portion of the virtual address space. This has the effect of increasing "locality of reference" on these data items, which can improve virtual memory paging performance.

9.3 REGIONS

A region is a block of contiguous virtual address space which is some multiple of the Address Space Quantum Size of 32 pages. When all the address space assigned to a region has been used by objects, new regions are automatically created as they are needed. The amount of address space allocated to an area's regions is generally unlimited, unless a particular area maximum size quantity was specified when the area was created.

Each region is identified by a number from 0 to 2047. Like an area number, this number is used as an index into any of the region descriptor tables in order to look up information about the region. The region characteristics found in these tables are summarized in Table 9-2.

Table 9-2 Region Characteristics

REGION-ORIGIN	Starting virtual address of the region.
REGION-LENGTH	The total amount of virtual address space assigned to this region in words.
REGION-FREE-POINTER	Offset into this region of the next free word that can be allocated.
REGION-GC-POINTER	Offset into this region of the next object which needs to be scavenged.
REGION-BITS	Defines various attributes of this region.
REGION-LIST-THREAD	Region number of the next region in this area's list of regions.

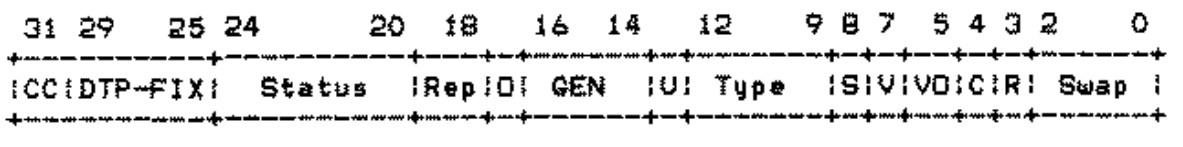
The address space allocated to this region are defined by its origin and its length characteristics. A region's allocation is only available for use by objects specifically created in this region's area; that is the address space cannot be used by just any storage request in general. The portion of a region's allocation actually used by objects occurs before the free pointer; the virtual address of the next free can be calculated by adding the region free pointer to the origin address.

All the regions in an area are linked together in a list. The list is anchored in the AREA-REGION-LIST entry for the area number. The next region in the list is found in the REGION-LIST-THREAD entry indexed by the first region; the third region is in the REGION-LIST-THREAD of the second entry; and so forth until a region number which is a negative number is encountered.

The free regions are also linked into a list. This is needed since when the garbage collector "flips" it frees a set of regions. Storage allocation needs to be able to find a free region easily.

9.3.1 Region Bits.

The REGION-BITS word defines a number of important attributes of the region. The fields within the region bits word are shown in Figure 9-1 and are discussed here.



CC:	CDR Code
Res:	Reserved, unused
Status:	Access and status bits to be used in the hardware virtual memory map
Rep:	Representation type 0 = list 1 = structure 2,3 unused
O:	Oldspace meta bit. 0 = old space or free 1 = new space, static space or fixed
GEN:	Region generation
U:	Region usage (for future expansion)
Type:	Space type (see space type code table)
S:	Scavenger Enable. Value: 1 = Scavenger can touch this area.
V:	Region zero volatility lock
VO:	Region volatility
C:	Cache Inhibit (Explorer II only)
R:	Reserved, unused
Swap:	Number of pages the Virtual Memory System should try to swap at a time (Explorer II only)

Figure 9-1 Region Bits Area Entry Description

A regions can store one of two types of data: list objects or structure objects. A structure object is any object other than a list. This is termed a region's representation type

The Oldspace meta bit is a bit that is set to zero in the first-level hardware maps for this region when the region is flipped to Oldspace. The map bit can then be used to efficiently implement the Garbage Collector's Oldspace Read Barrier.

A region's generation, volatility, and zero volatility lock properties are temporal attributes used to support the Temporal Garbage Collection algorithm. These are described in the section on Garbage Collection.

The scavenge enable bit is set when this region is part of the Scavenge Space defined by the garbage collector for a collection.

The cache-inhibit bit and the swapin quantum field are used in the Explorer II only. The cache-inhibit means that the virtual memory cache should be disabled for all pages in this region. The swapin quantum is the log (base 2) of the number of pages to try to swap in simultaneously on virtual memory reads in this region. A value of 3, for example, means try to swap in up to 8 (2**3) pages.

The space type attribute defines the storage allocation scheme that is used. The encoding of this field is shown in Table 9-3.

Table 9-3 Space Type Codes

Code	Region Type
0	Free
1	Oldspace
2	Newspace
3-8	Unused
9	Static
10	Fixed (static, not growable, no consing)
11	Extra PDL
12	Copyspace
13	Reserved for future use
14-15	Unused

9.4 STANDARD AREAS

When a machine comes up after a cold boot there are 25 FIXED areas reserved for use by the system itself. The first 11 areas are permanently wired down (not allowed to be swapped out by the virtual memory system) because they are either referenced heavily by the microcode or are referenced at or below the level of the virtual memory system. The system parameters file (LROY-QCOM) specifies these areas and their sizes. As described above, an area is indentified by a unique area number. The assignment of these area numbers for the standard areas is made by the LROY-QCOM file.

The fixed areas each contain a single region. They are special areas in that their regions may be smaller than the minimum region quantum size of 32 pages. This requires that they be specially handled by the storage allocation mapping system (see the discussion of the Address Space Map Area below).

The standard areas are as follows:

* Resident Symbol Area (wired) - Contains T and NIL symbols. This area's single region currently always starts at virtual address 0.

* System Communication Area (wired) - This area contains various values used by I/O routines and systems utilities. See discussion of SCA below.

* Scratch Pad Init Area (wired, read only) - Micro code variables are loaded into this area at startup.

* Micro Code Link Area (wired, read only) - Contains the microcode entry points for the miscellaneous operations.

* Region Origin Area (wired) - Contains the starting address for each region, indexed by region number.

* Region Length Area (wired) - Contains the length of each region, indexed by region number.

* Region Bits Area (wired) - Contains the region bits information for each region, indexed by region number.

* Region Free Pointer Area (wired) - Contains the region free pointer for each region, indexed by region number.

* Device Descriptor Area (wired) - Contains device descriptors for the I/O system.

* Disk Page Map Area (wired) - Contains the Disk Page Map Table for the Virtual Memory System

* Address Space Map Area (wired) - Contains the Address Space Map. The Address Space Map is a table indexed by the virtual address quantum and indicates the region number of the virtual address. If the region number in the address space map is zero, then either the virtual address has not been allocated to a region or the virtual address belongs to a fixed area. When a zero is found in the Address Space Map the fixed areas are searched to determine which area contains the virtual address. The region number is then determined from the area number, since for fixed areas the region number and area number are the same.

* Region GC Pointer Area (fixed) - Contains the GC pointer information for each region, indexed by region number.

* Region List Thread Area (fixed) - Contains the list thread for each region, indexed by region number.

* Area Name Area (fixed) - Contains the name of each area, indexed by area number.

* Area Region List Area (fixed) - Contains the first region number in each area, indexed by area number.

* Area Region Bits Area (fixed) - Contains the Region Bits word for each area, indexed by area number.

* Area Region Size Area (fixed) - Contains the default region size for each area, indexed by area number.

* Area Maximum Size (fixed) - Contains the maximum size for each area, indexed by area number.

* Support Entry Vector (fixed, read only) - Contains Lisp functions which are callable by microcode.

* Extra PDL Area (fixed) - The Extra PDL Area, or number consing area, is used to reduce the garbage generated when evaluating arithmetic expressions. All bignums and floating point numbers are first consed in the Extra PDL Area. Pointers into the Extra PDL Area are only allowed "in the machine" (see the chapter on garbage collection for a description of the parts of the processor that are "in the machine"). Before a pointer is written into main memory, a check is made to see if the pointer points into the Extra PDL Area. If the pointer being written points into the Extra PDL Area, then the object is copied out of the Extra PDL Area into the default consing area and the pointer is modified to reflect the

number's new address.

When the Extra PDL Area is full, all of the pointers in the machine are checked to see if they point into the Extra PDL Area. If a pointer into the Extra PDL Area is found, then the object is copied out of the Extra PDL Area into the default consing area and the pointer is replaced by a pointer to the copy. When there are no more pointers in the machine that point into the Extra PDL Area, then the Extra PDL Area contains only garbage. The address space is then reclaimed by setting the free pointer for each region in the Extra PDL Area to zero (currently there is exactly one region in the Extra PDL Area).

* Microcode Entry Area (fixed) - Contains indices into the Microcode Link Area for each microcoded function (see discussion of DTP-U-ENTRY in chapter on data types).

* Microcode Entry Debug Info Area (fixed) - Micro entry address or locative indirect micro-code-symbol-area.

* Scavenger State Area (fixed) - This single region area contains the stack used by garbage collection depth-first scavenging.

* Linear PDL Area (fixed) - The Linear PDL Area contains the Linear PDL (Push Down List, or stack) for the initial process. The Linear PDL (usually just called PDL) is the runtime stack for the process. The currently executing process will have the top part of its PDL cached in the processor's PDL Buffer.

Any memory reference to this area results in a page fault so that the virtual memory system can check if the target of the memory reference is really in the PDL Buffer.

* Linear Bind PDL Area (fixed) - contains the Special PDL for the initial process.

* Working Storage Area - The default cons area; most objects created by users are created in this area.

* Permanent Storage Area - Permanent data structures are placed here.

* Property List Area - Contains the property lists for symbols.

* Print Name String Area - Contains the print names for symbols.

* Control Tables Area

* Non-Resident Symbol Area - Contains most of the symbols in the kernel.

* Macro Compiled Program Area (read only) - Contains all compiled functions.

  In addition, any constant objects, such as lists, are also loaded into this area. This causes naive users to get mysterious error messages about trying to write in a read-only area when they try to do destructive operations (such as RPLACA) on constant objects in compiled functions.

* PDL Area - contains the linear PDL, except for the one used by the initial process which is in wired memory.

* SG And Bind PDL Area - The SG and Bind PDL Area contains the stack groups and the Special, or Binding, PDL for each process. The Special PDL contains the variable binding information for a process.

* Indirection Cell Area - This area is always pointed to by DTP-QC-YOUNG-POINTER's. It is used to keep track of pointers from older to younger generations.

* FASL Table Area - The FASL Table is placed here at load time.

* FASL Temp Area - Temporary structures created by the Loader are placed here.

* Debug Info Area - Contains debug info structures and fields.

9.5 SYSTEM COMMUNICATION AREA

The Systems Communication Area contains miscellaneous words that are needed for basic operation and do not rely on the rest of the machine operating. This information is shared by the microcode and Lisp. The file SYS:UCODE;LROY-QCOM contains the definitions of items in this area. The Systems Communication Area is wired and at the fixed address of 1000 (octal).

A map of systems communications areas is shown in Figure 9-2.

Octal Addresses		
1000-1046	:	Miscellaneous words
1047-1577	:	Not used
1600-	:	swap-in-rqb-origin

Figure 9-2 Map of Systems Communications Area

The miscellaneous words (1000 - 1046) are:

1. Area Origin Pointer - Virtual address of the Area Origin Area, which lists the starting virtual address of all fixed areas.

2. Valid Size - Number of words used in a saved band.

3. Object Array Pointer - Unused Unused

4. Ether Free List - Ethernet interrupt-handler variable.

5. Ether Transmit List - Ethernet interrupt-handler variable.

6. Ether Receive List - Ethernet interrupt-handler variable.

7. Band Format - Encodes format number in a saved band: 2000 -> new compressed format, otherwise old expanded format.

8. GC Generation Number - Reserved for value of %GC-GENERATION-NUMBER

9. Device Interrupt Table - Points to the Device Interrupt Table.

10. Temporary - Microcode bashes this at extra-pdl-purge.

11. Free Area Number List - Threaded through area-region-list, end=0.

12. Free Region Number List - Threaded through region-list-thread, end=0.

13. Memory Size - Number of words of main memory.

14. Wired Size - Words of low memory wired down; not all of these words are wired, this is really the virtual

address of the start of normal pageable memory.

15. Chaos Free List - Chaosnet interrupt-handler variable.

16. Chaos Transmit List - Chaosnet interrupt-handler variable.

17. Chaos Receive List - Chaosnet interrupt-handler variable.

18. Debugger Requests

19. Debugger Keep Alive

20. Debugger Data 1

21. Debugger Data 2

22. Major Version - Major version number of SYSTEM.

23. Desired Microcode Version - Microcode version this world expects. Note: this word may be stored with its data type field starting at bit 24 even though pointer fields are now 25 bits.

24. Highest Virtual Address - (Note: Should have this much room in the paging partition)

25. Pointer Width - 25

26. Descriptor Space Free Pointer - Current allocation pointer in the Device-Descriptor-Area.

27. Page Device Table - Unused

28. System Nupi Descriptor - Pointer to descriptor for system nupi

29. Processor Slot - Ucode stores A-SLOT-IM-IN here.

30. Overtemp Event - SIB will post overtemp events here.

31. Fiber Optic Warning Event - Microcode posts fiber optic warning event her.

32. Nupi Overtemp Event - NUPI Special event: microcode will post overheat special event here with formatter number embedded in bits <3:5> and non-zero value in bits <2:0>.

33. Physical Memory Map - Pointer to a memory table of memory board addresses in A-Memory.

34. Keybd Error Event - Keybd error condition; log of possible flaky keyboards.

35. Disk Retry Event - Disk retry condition; log of successful retries after disk errors.

36. Unused SIB Event - Place this address in all the unused SIB event locations

37. Parity Error Event - NuBus parity error; high 8 bits (0-7) with bit 8 on.

38. Parity Error Event 2 - NuBus parity error; part 2 - low 24 bits.

39. Syslog Wrap Event - System log wrap around event.

SECTION 10

Garbage Collection

10.1 INTRODUCTION

This section describes Garbage Collection (GC), the process by which storage no longer in use can be reclaimed. The discussion starts out at an abstract level, covering the general GC algorithm, then discussing the notion of Temporal Garbage Collection (TGC). We then delve further into the specifics of the microcode and virtual machine implementation that support GC and TGC. The last part of the section documents some low-level GC functions and variables.

10.2 BASICS

Most modern symbolic computing systems implement some form of automatic memory management. This involves allocating memory for objects from available free memory as they are requested, and returning their storage to the pool of available memory when the objects are no longer in use. Automatic memory management removes the burden of explicitly managing storage from the programmer. New objects are easily manufactured and returned on demand. Later, these objects can be reclaimed by the garbage collector when it can be determined the they are no longer being used by any program.

Here is an extremely simple example of how storage is allocated and then later becomes garbage.

```
STATEMENT 1:    (SETQ my-array (MAKE-ARRAY 100.))

                ... code using MY-ARRAY ...

STATEMENT 2:    (SETQ my-array nil)
```

Statement 1 creates an array object which the programmer holds in the variable MY-ARRAY. Then at the end of the program the variable is set to NIL. At this point, as long as the array has not been stored somewhere else, it becomes garbage and can be reclaimed.

There are several common garbage collection algorithms, among them Reference Counting, Mark and Sweep, and Copying. A good introduction to garbage collection in general can be found in T.J. McEntee's, "Overview of Garbage Collection in Symbolic Computing" Texas Instruments Engineering Journal Vol 3, No 1 (January- February) pp.130-139. The Explorer garbage collector is an incremental copying collector based on the Baker algorithm [H.G. Baker, "List Processing in Real Time on a Serial Computer," Communications of the ACM Vol 21, No 4 (April 1978) pp.280-294.].

10.2.1 Garbage Collection Spaces.

Garbage Collection bases its actions on the various space type properties of the address space. A summary of GC space type terminology is given in Table 10-1. Before garbage collection begins, the address space consists of Newspace (allocated virtual memory holding objects) and Free Space (address space not yet allocated). The size of Newspace gradually increases as programs request new objects, and garbage accumulates when the programs relinquish pointers to these objects.

In the simplest kind of copying garbage collector, garbage collection begins at some point by converting some subset of Newspace into Oldspace. The portion so converted (which may be all of Newspace) is termed the Collection Space, and the process of converting it to Oldspace is called flipping or starting a collection. Oldspace then becomes the domain in which garbage collection takes place; that is, the garbage collector copies all live objects (objects determined to be still in use) out of Oldspace to into a Copyspace. Once this is done, any objects left in Oldspace are garbage, so all of the address space in Oldspace can be reclaimed (made back into Free Space). This three-part flip, collect, reclaim process is termed a collection cycle.

Table 10-1 Space Type Terminology

ADDRESS SPACE	All of the system's addressable memory.
FREE SPACE	Unused address space (not yet allocated).
ALLOCATED SPACE	Used address space (allocated).
NEWSPACE	The portion of Allocated Space which can be garbage collected (excludes Static Space). Objects in Newspace may be moved by GC. Also space where new objects are created.
STATIC SPACE	The portion of Allocated Space which is NEVER subject to garbage collection. Static Space objects will not be moved (collected) by GC, but may be Scavenged during collection cycle.
COLLECTION SPACE	A subset of Newspace which is flipped to Oldspace and collected during a GC cycle. This can be all of Newspace or just one generation of it.
OLDSPACE	The space from which GC is evacuating live objects. An invisible space.
COPYSPACE	The space to which GC is copying live objects. Also part of Scavenge Space.
SCAVENGE SPACE	All the spaces that must be examined by the scavenger in order to find all pointers to live objects. These are all the spaces which might contain references to Oldspace.
DYNAMIC SPACE	All visible spaces containing dynamic, movable objects (Newspace + Copyspace).

Since the goal of a garbage collection cycle is to copy everything useful out of Oldspace and then reclaim the storage, a copying collector does not really do <u>garbage collection</u> at all, but actually <u>live data collection</u>. The net savings gained by garbage collection is then the difference between the original Oldspace size and the size of the Copyspace where only the in-use Oldspace objects have been collected. In the worst case, when all of Oldspace contains still-used objects, the size of the Copyspace will be the same as the original Oldspace.

Because Copyspace consumes portions of the Free Space in existence at the time of the Newspace-to-Oldspace flip, this worst-case assumption means that available Free Space must be at least the size of Newspace Collection Space in order to <u>guarantee</u> that the collection cycle can finish. If there is a lot of garbage in the Collection Space, then Copyspace will require much less free memory than the Collection Space Size. However, to be safe, all the GC space requirement computations make the worst-

case assumption.

There are some long-term system data structures that are always in use and would never be made into garbage. Since such objects would simply be copied and re-copied by the garbage collector, they are isolated in a space called Static Space. Static Space is set aside to hold objects intended to be permanent and is never considered for Collection Space. Its object will not be collected, although it must still be purged of any Oldspace references; as with Copyspace, such references must be replace by references to the object's new location in Copyspace. The benefit of Static Space is that GC will not expend effort repeatedly copying its live objects. However this also means that Static Space objects which do happen to become garbage cannot be collected (or at least not until action is taken to convert Static Space to collectable Newspace).

10.2.2 Scavenging.

The process of identifying all live objects in Oldspace and assuring that they are transported to Copyspace is known as Scavenging. The Scavenger begins with a set of well-defined objects which form the "root" of the tree of all live Lisp objects. This root is copied to Copyspace by the flip process itself. From the root the scavenger can traces through the tree, examining every object for Oldspace references. As more Oldspace objects are found, they are copied to Copyspace and references to them are replaced with the new Copyspace location. A marker is left behind in Oldspace to indicate that this object has been traced and to redirect other references. Any future references to the same Oldspace object found will detect the "already-traced" marker, hence avoiding cycles in the tree. Since the work of copying the object has already been done, all that is required is to update the reference with the redirection to Copyspace. The object evacuated to Copyspace may now itself contain more Oldspace references, so it too must be scavenged. Scavenging will continue until the tree has been traced entirely; that is, until all Copyspace objects have been examined and any Oldspace references in them eliminated.

10.2.3 Incremental Collection.

The type of collection described so far assumes that the entire collection takes place in one stop and collect; that is, no new objects are created while the collection is in progress. While such an assumption greatly simplifies the internals of the garbage collector, it is less than useful since collections on a large virtual address of, say, 50 MB can take an hour or more. The actual Explorer garbage collector is an incremental collector. Using such a collector, the collection process can occur while other programs are running on the machine. The

incremental collection proceeds gradually, without long delays between operations noticed by running programs.

An incremental collector is more difficult to implement because care must be taken to ensure that there is no unwanted interaction between the garbage collector and the other programs running on the system, called mutators, that are creating new objects and altering old ones while the garbage collector is active. This is accomplished by enforcing a set of barrier rules which define where new objects are created, where Oldspace references can occur and what happens when a mutator makes a reference to an object in Oldspace. These rules are outlined below.

1. All new objects created after a flip occur in Newspace.

2. Oldspace references may only occur in Oldspace itself and in the as-yet unscavenged portion of Copyspace.

3. No Newspace-to-Oldspace references are allowed, although there may be Newspace-to-Copyspace references.

For simplicity's sake, these barriers are implemented by a procedure called the Read Barrier; so called because it dictates an action to be performed whenever an object is read from virtual memory. The Read Barrier states that no Oldspace references may be seen by a mutator. Any such reference will be trapped, the object will be copied out to Copyspace if necessary, and the reference actually provided to the mutator will be the object in Copyspace. Since any reference the mutator has is guaranteed not to be to Oldspace, any new objects created after the flip cannot have Oldspace objects stored into them. Therefore, Newspace allocated after a collection cycle begins does not need to be scavenged.

In such a scheme the mutator ends up doing some of the work of the scavenger; that is, the copying of an object may take place because of a mutator's dynamic reference to it rather than because of the scavenger's tracing of the static tree. For the scavenger, this just means that the first reference it sees to that object will find it already copied, saving it some work. But we will see in later discussion that there are important differences in the order that objects are copied by the two mechanisms, and that the differences can have a significant impact on system performance.

It turns out to be convenient to tie the rate of incremental garbage collection to the rate at which new storage is being requested. Thus, a certain amount of collection work (scavenging) is done for each new storage word allocated. Thus the rate of garbage collecting is proportional to the rate of consing in the system. Appropriately, this gives the consing primitives semantics both of allocating storage and working to

reclaim it.

An account is kept of how much work the scavenger needs to do. This balance is increased when new objects are created, and decreases as the scavenger does its job. Any work accomplished due to dynamic referencing of objects acts is also applied to as a credit against the scavenger's work allotment.

10.2.4 Generational Garbage Collection.

The amount of work done in a collection cycle is proportional to the size of the Collection Space and, even more strongly, proportional to the amount of live data in the Collection Space. In light of this fact a scheme which concentrates GC efforts on small, well-defined Collection Spaces which are the most likely to contain garbage can greatly increase the efficiency of garbage collection and reduce the GC overhead imposed on the mutators.

A Generational Garbage Collector defines the Collection Space on the basis of the observation that a high proportion of newly created objects become garbage quickly, while older objects tend to have a much smaller proportion of garbage. This observation can be explained simply by realizing that memory representing such things as compiled system routines, editors, and window managers, rarely become garbage, while dynamic data structures of the currently operating program tend often to be used briefly then discarded.

The generational collector partitions the address space into a number of generations, ranging from very young to very old, each of which is small when compared to the entire virtual memory space. New objects are created in the youngest generations, which is where the generational collector concentrates the majority of its efforts and where the payback in garbage collected per unit of work done is likely to be the highest. As objects survive these collections, they may be promoted into higher and higher generations where collections need be less frequent since the concentration of garbage is lower. Finally, in the oldest generations there is almost no garbage at all; they are populated by long-lived objects which have proven their worth by surviving several collections.

10.2.5 Scavenge Space.

It is important to guarantee that there is no useful object remaining in Oldspace when it is reclaimed. This requires a proper definition of Scavenge Space, the spaces which must be subjected to scavenging in order to find all references to live Oldspace objects. In other words, when a collection begins, it must have the proper root set for reaching all live data in the Collection Space.

The definition of a proper, minimal Scavenge Space is the key to efficient generational collection. It would not be worthwhile simply to isolate memory with a high concentration of garbage if it were still necessary to trace the entire tree of live Lisp objects in order to collect that space. Some mechanism must be defined for remembering and isolating references only to the live young objects. Then, just these smaller live-object subtrees need to be traced.

This task is simplified by the observation that there are three sorts of references in a Collection Space made up of one generation:

1. Reference among objects in the collected generation.
2. References from objects in a younger generation to the generation being collected.
3. References from objects in an older generation to the generation being collected.

Consider each type of reference. Note that the first set of intra-generational references are guaranteed to be taken care of by the normal scavenging mechanism as long as all references into the generation are traced. As for the second set, it is possible with little expense to scavenging all younger generations in order to find these young-to-old references. There are two reasons for this. First, since most collections are of the youngest generations there will usually only be a small number of generations younger than the one being collected (possibly even none). Secondly, the size of these younger generations is kept small by collecting them frequently and promoting survivors to higher generations. The hard part, then, is the third set of old-to-young references because they are so sparsely scattered populated over the largest amount of space.

To keep track of these references it is convenient to implement a Write Barrier which tests all objects written to memory. If the barrier detects that a younger object is being stored into an older one, a trap is taken and the reference is recorded in a generational reference list. When this generation is later collected, its Scavenge Space consists of this reference list plus all younger generations plus the Copyspace created during the collection.

10.3 EXPLORER TGC IMPLEMENTATION

The discussion so far has provided an outline and motivation for the type of garbage collection algorithm in the Explorer system,

but has glossed over all the nitty gritty implementation details. As anyone versed in the art of garbage collection implementation can attest, such an explanation seems so simple and elegant on paper that one wonders if it is an accurate way to portray the incredibly intricate code which actually embodies it. The rest of this section attempts to address these low-level details.

10.3.1 Generations.

The Explorer garbage collection implementation new with Release 3 is termed Temporal Garbage Collection (TGC). It is an extension of the incremental Release 2 garbage collection algorithm which is very low-cost because of its concentration on memory in the lower generations. There are six "logical" generations listed in Table 10-2. In contrast, the Release 2 garbage collector only defined the equivalent of the Generation 3 and and Static Generation 3 levels (plus the super-temporary Extra-PDL number consing generation, described later).

Table 10-2 TGC Generations

YOUNGEST
Extra-PDL
Generation 0
Generation 1
Generation 2
Generation 3
Static Generation 3
OLDEST

10.3.2 Areas and Regions.

All the space-type properties for Explorer virtual memory are defined on a per-region basis. Regions, which are a basic storage management unit, are described along with their attributes in the section on Storage Management. Every object is in some region that is part of an area. Each region has a space type property (NEW, OLD, COPY, STATIC, FIXED, EXTRA-PDL) which roughly corresponds with the abstract garbage collection spaces described above. FIXED space is like STATIC space for most garbage collection purposes, and the EXTRA-PDL type exists simply to flag the super-temporary number consing generation.

10.3.2.1 Volatility.

Each region also has two temporal attributes: a Generation and a Volatility. The Generation indicates the age of the objects in the region, with 0 being the youngest and 3 the oldest. Volatility specifies the kinds of references allowed by objects in this region; more specifically, it is the age of the youngest generation an object in this region can point to directly (references younger that this will be stored indirectly). A summary of the different volatility level meanings is given in Table 10-3. Normal Newspace regions are always created with volatility equal to generation. This means they can contain references to other objects in their own generation and to any older generation.

Table 10-3 Volatility Level Meanings

Volatility 3	Can point to oldest objects only.
Volatility 2	Can point to generation 2 or 3 objects.
Volatility 1	Can point to generation 1, 2 or 3 objects.
Volatility 0	Can point to object in any generation (excluding Extra-PDL).

Object Creation on the Explorer takes place on a per-area basis. Object creation in an area will cause a young Newspace region to be created (usually generation 0). When generation 0 is flipped, these regions become Oldspace. When an object is copied out of Oldspace to Copyspace, it will always be to a Copyspace region in the same area. Garbage collection does not change the area in which an object resides.

10.3.2.2 Default Cons Generation.

Each area has a default cons generation attribute which specifies the generation in which objects in this area will first be created. As they survive collections they may be promoted to higher generations (but always in the same area). Nearly every non-FIXED area in the shipped configuration has a default cons generation of 0. Extensive testing has shown that this is the best policy from a performance standpoint. An area's default can be modified with the %SET-AREA-DEFAULT-CONS-GENERATION primitive, but this is not recommended.

Because we attempt to keep the size of generation 0 small enough that collection of it can take place entirely in main memory, there is an additional policy that objects above a certain size threshold (the value stored in the %MAX-GENERATION-0-OBJECT-SIZE counter) will be not be created in generation 0. Instead, they will be consed in the generation specified by MIN(1,default-cons-generation).

10.3.3 Automatic Collection Mode.

Automatic collection means that a special GC process will monitor generation sizes and cause collection cycles to occur automatically, when certain threshold sizes are reached, invisibly to the user. The system is shipped with automatic GC on (started up by the GC-ON function). The following policies are followed by the automatic collector.

The generation 0 flip threshold is computed as a fraction of installed physical memory in an attempt to limit the size of generation 0 to an amount that minimizes the number of pages needing to be swapped in or out during the collection. The controlling variable is GC-FRACTION-OF-RAM-FOR-GENERATION-0.

The highest generation that will be collected automatically is limited to 2 (a value of *GC-MAX-INCREMENTAL-GENERATION* higher that this will be normalize to 2). This is because an incremental collection of generation 3 would greatly interfere with interactive response.

The automatic collector will always promote survivors of generation 0 into generation 1, and survivors of generation 1 into generation 2 in an attempt to keep the size of the two youngest generations manageable. Generation 2 survivors, however, will not be promoted by the automatic collector.

Flip thresholds for generations 1 and 2 are computed using worst-case 100% survival assumptions and the maximum virtual memory size of the current configuration (which is roughly the smaller of 128 MB and the amount of swap space available). Because generation 2 survivors are not promoted, generation 2 can "fill up" such that a collection of it cannot be guaranteed to complete under the worst-case assumptions. In this case, generation 2 will be "shut down" (no longer collected) and the user will be notified.

A collection of an older generation will not start (even if the flip threshold is exceeded), unless the last collection was of a younger generation. This is intended to maximize the free space available for the older generation collection and avoid thrashing.

10.3.4 Batch Collections.

The FULL-GC and GC-IMMEDIATELY functions still exist to perform batch collections. When invoked, they will turn automatic GC off (after completing any pending generational collection), then collect each generation up to a user-specified maximum with the option to promote or not. While both can be used to perform any combination of max-gen/promote collection, they are meant to have different semantics.

GC-IMMMEDIATELY by default does not promote and collects generations 0, 1, and 2. Since these generations will usually contain most of the objects created since the system was booted, this is meant to be roughly "batch collect my working set".

FULL-GC by default promotes and collects all generations including 3. It assumes you intend to DISK-SAVE afterwards so it also cleans out some big data structures in order to try to reduce band size. It is also meant to be used after doing MAKE-SYSTEMS in order to clean up the garbage in the environment, and promote all the (sure-to-be-long-lived) code just loaded into generation 3 where it won't be subject to automatic collections.

10.3.5 Scavenging for TGC.

The scavenger starts out at the beginning of a Copyspace region or other region in the Scavenge Space and proceeds forward in a basically linear fashion. Every boxed word of Copyspace containing a pointer type must be checked for reference to Oldspace. When such a reference is found, the object is copied out (if not already done) and words of data type DTP-GC-FORWARD are left behind in every slot of the structure copied (even unboxed ones). The pointer field of the GC-FORWARD points to the corresponding cell in the Copyspace representation. The GC-FORWARD serves to indirect further references to the new location in Copyspace, ensuring that after garbage collection, the copied object will still be shared in the same way the original object in Oldspace was shared. Finally, the original Oldspace reference is replaced with the a reference to the new object in Copyspace.

As scavenging progresses in the region, it updates a scavenge pointer (the REGION-GC-POINTER) which delimits the portion of the space that has been scavenged so far. The storage before the scavenge pointer cannot refer to Oldspace since it has already been scavenged. Storage beyond the scavenge pointer has not been scanned, so may still contain pointers to Oldspace. When the scavenge pointer catches up with the allocation pointer (the REGION-FREE-POINTER) in all scavengable regions of all areas, scavenging is completed. This marks the end of the collection cycle and Oldspace can be reclaimed.

As has been noted in the GC literature, the scavenger in a normal copying collector works breadth-first, which is the least desirable from an object placement point of view. In breadth-first tracing, sibling nodes in a tree are made adjacent in Copyspace. But such objects are not very likely to be near to one another ("related") in a dynamic reference chain. A depth-first or "approximately" depth-first algorithm is preferable because it tends to copy objects closer to their offspring in the tree, and this is more likely to indicate that adjacent objects are part a dynamic reference sequence. This is important since object compaction in a virtual memory system can have a

significant impact on paging performance.

The TGC scavenger is object-oriented ("approximately" depth-first). Scavenging starts at the beginning of copy space and scans each word. If a word refers to an object in old space, it is copied to the end of Copyspace and pushed on an object stack in the SCAVENGER-STATE area, along with a count of the number of boxed Qs in the object needing to be scavenged. The pushed object is now scavenged, which might again cause a copy operation and other object stack push. This recursive scavenging continues until either the depth of the stack is reached or an object is completely scavenged. In the latter case the object is popped from the stack and scavenging of its predecessor continues. If the stack becomes empty, the scavenge pointer (the REGION-GC-POINTER) beyond the end of the first object pushed, and the next linear word of Copyspace is scavenged. The object-oriented scavenging process means that some words after the scavenge pointer may already have been scavenged, but we will scavenge them again. There is no extra work done the second time, of course, outside of the memory reference.

While we have found that this algorithm is better than the undirected breadth-first scavenger, it still leaves a lot to be desired. This may be because the object tree is so bushy, and good heuristics for deciding which offspring is most important to make adjacent are difficult. However, we have found that the order in which objects are dynamically referenced does tend to provide a good heuristic for objects placement. In other words, the mutator tends to be much better at causing objects to be copied than is even the best scavenger.

As a consequence, whenever a flip takes place instead of starting up the scavenger right away, we arrange to delay for a time in order to allow the mutator to move (maybe the most frequently referenced) objects according to usage pattern. The amount of the delay is expressed in terms of a scavenger work bias. In other words, we give the scavenger a work "credit," and it will not actually kick in until enough consing has been done to cancel this credit. (The two internal counters for cons work and scavenger work are %COUNT-CONS-WORK and %COUNT-SCAVENGER-WORK). The amount of credit is the same for all generations, and is equal to the worst-case assumption amount of consing that would have to be done to drive the entire generation 0 collection. Note that because of the very high concentration of garbage in generation 0, the actual amount of consing required to drive the collection is really quite small, so that if consing were allowed to drive scavenging the collection would complete practically right away.

10.3.6 Indirection Cells.

TGC uses an indirect reference in the form of a special "super-invisible" forwarding pointer in order to implement its generational reference list for old-to-young references. Detecting such references is done as part of a Write Barrier. Every object written to virtual memory is subjected to this Write Barrier. If an attempt is made to store an object younger than a region's volatility, into an object in a region, the Write Barrier action will be to store the young object in an Indirection Cell instead. A forwarding marker is then placed in the location where the object store was intended. This marker is a word of type DTP-GC-YOUNG-POINTER whose pointer field points to the indirection cell actually containing the object. Any dynamic reference to this cell will be automatically forwarded to the indirection cell, and the object there will be referenced instead.

Indirection cells are kept in a special INDIRECTION-CELL-AREA which contains them and nothing else. Regions in this area have a special interpretation for their generation and volatility properties; these attributes are used to classify the old-to-young references of the cells in the regions.

For example, consider Figure 10-1. An attempt to store a newly-created list into the value cell of a symbol in generation 3 has caused a GCYP to be stored in the value cell instead. The GCYP points to an indirection cell which now actually contains the generation 0 list object. The indirection cell is created in an INDIRECTION-CELL-AREA region with a generation of 3 (the generation of the older structure containing the GCYP) and with volatility of 0 (the generation of the young object).

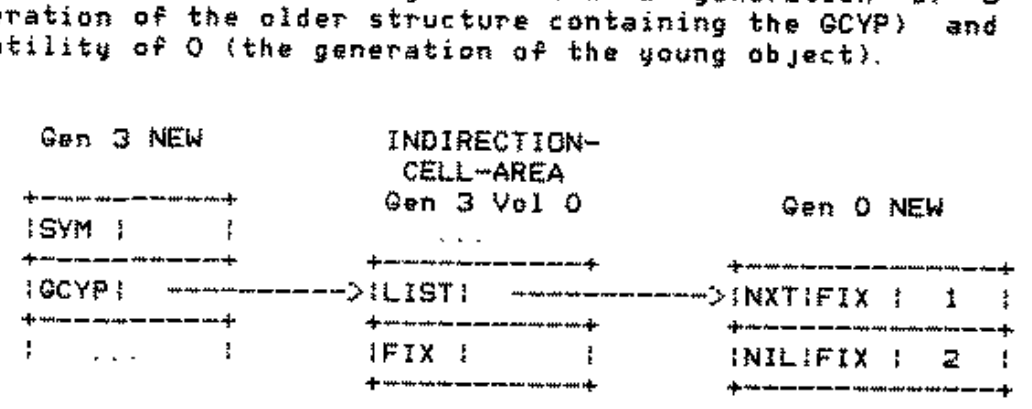


Figure 10-1 Indirection Cell Forwarding

Now when generation 0 is next collected all regions with volatility 0 will be part of the Scavenge Space because it is this in set of spaces where generation 0 references are confined. When the indirection cell is scavenged the list storage will be copied to Copyspace and the list pointer will be updated with the object's new location.

If the mutator sets the symbol to a new value before scavenging reaches the indirection cell, the new value will overwrite the indirection cell, not the value cell, even if the new value is a type which does not point to storage, say a FIXNUM. The symbol value cell will still point to the indirection cell which will then contain the FIXNUM. Now when scavenging occurs, no action is taken on this indirection cell since it contains an immediate object. This may indicate that the list previously pointed to is now garbage, if there are no other references to it.

If the mutator stores a new list, say one in generation 2, into the symbol, this same indirection cell may be reused. This is because volatility 0 means "can point to any object in generation 0 or higher", in other words "can point to anything".

On the other hand, let's imagine the indirection cell is volatility 1. This can occur if a large object which has been created in generation 1 was originally stored in the symbol. If at this point a new generation 0 object is stored into the symbol, then a new indirection cell will need to be made because a volatility 1 location is not allowed to point to a younger generation 1 object. Now in order to satisfy the object indirection rules, a GCYP will be placed in the old volatility 1 indirection cell. This GCYP will forward references to the new volatility 0 indirection cell.

The GCYP in the symbol value cell will remain there until a collection of generation 3 takes place in which the object pointed to by the symbol is no longer in a younger generation (that is, is itself now in generation 3 or is an immediate value).

Currently there is a back pointer stored with every indirection cell object, making each cell two words long. The back pointer is simply a FIXNUM containing the address of the GCYP which created this cell in its pointer field. This back pointer exists for debugging reasons only and has no other use in the TGC internals.

10.3.7 Following GCYP Forwarding.

The GCYP is like a single-cell forwarding pointer such as DTP-ONE-Q-FORWARD in that it indirects only a single location. It is followed on both reads and writes. On writes it must be followed so that the location overwritten is the indirection cell. Without this preread, any storage pointed to by the indirection cell would be "anchored" by the pointer and could not be garbage collected.

An indirection cell contains all significant fields of the displaced object, even the cdr code. This means that the GCYP-forwarding even in cases where just the cdr code is being looking

cdr code.

NOTE

The presence of GCYPs in memory has significant consequences on the class of pointer-manipulating subprimitives that can be used to modify fields of arbitrary memory words (generally the ones beginning with %P-). TGC has changed the semantics of these subprimitives. Consult the section on storage subprimitives for more details.

10.3.8 Promotion.

When young objects are promoted into higher generations after surviving a collection they may now be in the same generation as the objects which point to them. If this is the case, when this generation is collected any GCYPs pointing to them may be snapped out. The scavenging sequence here is: read the GCYP word; follow it to the actual object. If that object points to Oldspace copy it out to Copyspace. Now attempt to store the Copyspace reference into the word where the original GCYP was read from and invoke the Write Barrier. If a volatility violation still exists, the object will be stored in an appropriate indirection cell and the GCYP's pointer field will point to it. If there is no longer a volatility violation, then the object can be stored safely into the location. In this case the indirection cell was in Oldspace and will be reclaimed when the collection is finished.

In all but a few specially marked regions, promotion changes both the generation and the volatility of objects. Generation 0 volatility 0 regions are made generation 1 volatility 1, and so forth. Volatility must track generation so that objects can continue to point freely to other objects in their own generation. The exceptions to this rule are the locked-volatility regions described later.

Whether a collection will promote or not is dictated by a flag to the microcode flipper process. When promotion occurs, say in a generation 0 collection, all generation 0 Newspace is actually made into generation 1 Oldspace whereas a non-promoting flip would leave it generation 0. When copyspace is created for this area it too will be marked as generation 1. Then the process of reclaiming oldspace simply converts Copyspace generation 1 to Newspace generation 1 and we're done.

One problem that can arise is fragmentation of generation 1 Newspace due to repeated promoting generation 0 collections. Even if only a few words survive into Copyspace, an entire 16-page region must be allocated to hold these few words. This fragmentation is avoided by converting existing, partially-filled generation 1 Newspace into Copyspace before the collection. The scavenge pointer of such a region is initialized to the region's free-pointer rather than to 0 so as to avoid unnecessary scavenging of Newspace.

10.3.9 Locked Volatility 0 Areas.

There are certain data structures that are so rapidly changing and are so likely to contain younger references that it is impractical to allow them to contain GCYPs. Runtime stacks represented by Regular PDL arrays are an example of such a structure. The overhead of managing GCYP references, especially in light of the special virtual memory reference that is done in the top, cached portion of PDLs, would be great. Similar arguments can be made for the stack group structures themselves and for Special PDL arrays. In order to prevent GCYPs from ever being created in these structures they are isolated in a few areas (the PDL-AREA for Regular PDLs and the SG-AND-BIND-PDL-AREA for stack groups and Special PDLs) and these areas are made volatility 0 (can point to anything) and have a special Volatility Locked attribute which specifies that the volatility will never change (they will always be allowed to point to anything). The special FIXED system areas in low memory are also locked volatility 0 so that GCYPs can never be stored there. This avoids the overhead of looking for GCYPs when accessing the frequently-used system structures stored there. Moreover, since these regions contain system structures which can never become garbage, they are generation 3 and are treated as Static Space. The cost paid for these locked volatility 0 regions is that they must be scavenged for every flip (because by definition they may contain references to any generation 0). Since the amount of virtual memory in these regions generally remains small, this is an acceptable cost to pay for the processing simplicity it buys.

10.4 GC SUBPRIMITIVES AND VARIABLES

This subsection documents a number of internal garbage collection primitives and variables. Since they are part of the low-level GC implementation they are subject to change without notice. All are in the SYSTEM package.

10.4.1 Flipping, Scavenging and Reclaiming Oldspace.

%gc-flip-ready Variable
Set to T by the microcode when all of Oldspace has been scavenged and can be reclaimed. Set to NIL at flip time when scavenging is started.

gc-oldspace-exists Variable
T if there is any Oldspace anywhere meaning there is a collection in progress. Set to NIL when there is no Oldspace.

gc-maybe-set-flip-ready
Scans all regions looking for Oldspace and sets the values of %GC-FLIP-READY and GC-OLDSPACE-EXISTS appropriately. Called after warm-boot and by anyone before attempting a flip.

inhibit-scavenging-flag Variable
If NIL, the scavenger is enabled and will be invoked after every consing operation. Set by GC-ON and reset by the microcode when scavenging is done.

inhibit-idle-scavenging-flag Variable
If NIL, the scavenger may be called from the scheduler if the machine is idle. When T, idle scavenging is prohibited.

%scavenger-ws-enable Variable
When a generation collection begins scavenging is temporarily disabled to allow the mutator to move objects dynamically. When this variable is true the scavenger is on hold. When NIL, scavenging can occur (when INHIBIT-SCAVENGING FLAG is also NIL).

gc-flip-now (gc-type &optional (scav-work-bias 0))
The Lisp function that initiates a collection cycle. Does some housekeeping to prepare Copyspace then calls the microcode flip routine %GC-FLIP-NOW. The generation flipped is specified by FLIP-TYPE (2 bits generation, low bit promote).

Call this to flip a generation but leave the scavenger off. See its use in START-TRAINING-SESSION.

%gc-flip (flip-type-fields)
Microcode function that flips Newspace to Oldspace according to FLIP-TYPE-FIELDS. Bit 0 of FLIP-TYPE-FIELDS is the promote bit (1 = promote). Next two bits are the generation to flip. Next 21 bits are the scavenger bias amount in words.

This function is also responsible for setting the scavenge-enable flag in any regions that will be part of the collections Scavenge Space, and for flushing the subjecting

the contents of all registers in the machine state to the Read Barrier in order to get Copyspace bootstrapped. It is called from GC-FLIP-NOW.

%gc-scavenge (work-units)
Scavenge for WORK-UNITS worth of work. This is called by the batch collection routines with an arg of a a few thousand--big enough so that scavenging will complete as quickly as possible but will come up for air occasionally in case the user wants to try and do some work while it is going on.

When automatic GC is on scavenging occurs as a side-effect of consing.

gc-reclaim-oldspace
Lisp function that reclaims the address space of any existing Oldspace. Does nothing if there is no Oldspace. Otherwise batch scavenges until %GC-FLIP-READY is true (meaning scavenging done and OK to reclaim the Oldspace). Then for all Oldspace regions of all areas, deallocates any swap space associated with the address space, unlinks the Oldspace regions from their area region lists, and returns them to the free region pool by calling microcoded %GC-FREE-REGION.

%gc-free-region (region)
Microcoded function that returns REGION to the free region pool and flushes any hardware maps still set up by its virtual pages. Used by GC-RECLAIM-OLDSPACE on Oldspace regions after scavenging is complete.

10.4.2 GC Predicates.

gc-in-progress-p
True if we're in the middle of a collection (Oldspace exists and can't yet be reclaimed).

gc-active-p
True if the automatic flipper process is active; else NIL.

current-collection-type
Returns generation and promote flag for current collection if automatic GC is active; else NIL.

10.4.3 Starting and Stopping Automatic GC.

arrest-gc reason

unarrest-gc reason

Arrest (or unarrest) the GC flipper process for reasons REASON.

gc-off-temporarily

gc-off-temporarily-back-on
Use to turn automatic GC off temporarily then back on as, for example, might be required during a patch.

10.4.4 TGC Control.

%set-area-default-cons-generation area generation
Primitive for changing the default generation in which new objects will be created in this area.

disable-tgc
Disables TGC by setting the default cons generation of all areas to 3 so there is no more young consing. Then does a full promoting collection to get rid of any indirection cells. Not recommended. Can be undone by ENABLE-TGC.

10.4.5 Number Consing.

number-cons-area Variable
The area number of the area where BIGNUMS, ratios, full-size floats and complex numbers are are consed. Normally this variable contains the area number of the EXTRA-PDL-AREA. This enables number consing, the low-overhead garbage collection of extended numbers. To disable number consing, set this variable to the number of another area.

number-gc-on (&optional on-p t)
Used to turn number consing on or off. Actually sets the variable NUMBER-CONS-AREA to the EXTRA-PDL area for true ON-P, and to the BACKGROUND-CONS-AREA for ON-P of NIL.

10.4.6 Miscellaneous.

%gc-generation-number Variable
A FIXNUM which is incremented whenever the garbage collector flips, converting one or more regions from Newspace to Oldspace. If this number has changed, the address of an object may have changed. Comparing this number with a hash table's internal GC generation number is used to cause EQ hash tables to rehash after a GC. The value cell is actually forwarded to a slot in the System Communication Area so that the changes to its value can live across a DISK-SAVE.

%region-cons-alarm Variable
Counter incremented whenever a new region is allocated.

%page-cons-alarm Variable
Counter incremented whenever a fresh page is allocated.

%gc-cons-work (nqs)
Informs the GC microcode that nqs Qs have been allocated. There is no need to do this if storage is allocated via the normal microcoded consing routines.

%gc-scav-reset (region)
Tells the scavenger to forget any work done so far in REGION and remove REGION for the cons cache. Returns T if the scavenger was actually looking at region; otherwise NIL.

SECTION 11

Function-Calling

11.1 INTRODUCTION

This section explains the functional objects in the Explorer system and how they are called. It is not necessary to understand the earlier virtual machine (VM1) to understand the mechanisms of this virtual machine (VM2), though it may prove helpful in understanding our motivations. The following sections describe the various kinds of functional objects, VM1 function-calling and the motivations behind the VM2 design, the layout of stack-frames, the structure of the compiled-function object, and the actual function-calling mechanisms.

11.2 FUNCTIONAL OBJECTS

Objects of almost every user-visible data-type in the Explorer system can be called as functions (the exceptions being numbers and characters); they may be divided into four categories:

1. Compiled functions: the objects which are produced by the Lisp COMPILE function or by loading the file generated by COMPILE-FILE.

2. Interpreted functions: objects which are represented in list structure.

3. Indirect functions: objects that may be applied to arguments, but which when applied find an internal functional object and apply it instead.

4. Non-functions: objects that may be applied to arguments, but which do some action specific to the data type.

11.2.1 Compiled Functions.

Compiled functions are of data-type DTP-FUNCTION, and are directly interpreted by the microcode. Compiled functions are the most important part of the function-calling code. Most of this chapter deals with calling compiled functions. (The section on FEF LAYOUT describes the layout of a compiled function object.)

11.2.2 Interpreted Functions.

Interpreted functions are produced by DEFUN or LAMBDA, and are passed on to the Lisp interpreter. Objects of data-type DTP-LIST, DTP-STACK-LIST, or DTP-LOCATIVE are treated as interpreted functions. When the microcode encounters an interpreted function, it collects a list of the arguments and passes the argument list and the function to SYS:APPLY-LAMBDA, the microcode's interface to the Lisp interpreter, which it finds in the support vector. (See the section on Support-Vector for more details).

11.2.3 Indirect Functions.

There are several different kinds of indirect functions: symbols, closures, instances, named-structures, and funcallable hash-arrays. They are discussed below in order of complexity.

11.2.3.1 Symbols.

The simplest of the indirect functions is the symbol (DTP-SYMBOL). When a symbol is called, the microcode calls the contents of its function cell instead. Most compiled code does not actually call symbols; rather than a pointer to the symbol, the function will contain an invisible pointer (DTP-EXTERNAL-VALUE-CELL-POINTER) to the symbol's function cell. It is this indirection mechanism that allows the developer to recompile a function without having to recompile its callers.

11.2.3.2 Closures.

Closures are also quite simple: a closure is a pair, one part of which is a functional object to be called. The other part of the pair is a list of pointers to value cells and values to bind them to, if the closure is a dynamic closure (DTP-CLOSURE), or an environment structure, if the closure is a lexical closure (DTP-LEXICAL-CLOSURE). When a dynamic closure is called, the microcode performs all the special-bindings, then calls the closure's functional object. When a lexical closure is called, the microcode calls the closure's functional object and places

the environment structure in a special local variable in the call-frame. (See the section on Closures for more detail). In both cases, a previous environment is restored for the function to execute within, though the type of the environment is quite different.

11.2.3.3 Instances, Named-Structures, and Funcallable Hash-Arrays.

Instances, named-structures, and funcallable hash-arrays are closely related. A named-structure is an array (DTP-ARRAY) with its named-structure-flag (see the sub-section on Arrays) turned on. A funcallable hash-array is a named-structure with its funcall-as-hash-array flag turned on. An instance (DTP-INSTANCE) contains a pointer to its flavor's method table which is a funcallable hash-array. (See the sub-section on Calling an Instance).

A call to a simple named-structure is turned into a call to SYS:NAMED-STRUCTURE-INVOKE found through the support vector, (See the sub-section on Support Vector), with the structure as the first argument and all the other arguments following. SYS:NAMED-STRUCTURE-INVOKE looks for the SYS:NAMED-STRUCTURE-INVOKE or :NAMED-STRUCTURE-INVOKE property on the type-symbol of the structure, and if found calls it with the arguments and (if the named-structure-invoke property is a closure) the structure itself.

Calling a funcallable hash-array causes a hash-table lookup of the first argument in the specified hash-array. If a functional object is found, it is called, otherwise SYS:INSTANCE-HASH-FAILURE (another function found through the support vector) is called. Calling an instance binds SELF, then binds any special instance variables, then picks up the funcallable hash-array that is its flavor's method table and calls it as described above. (More detail on the structure of flavors and the calling of instances may be found in the sub-section on Calling an Instance).

11.2.4 Non-Functions.

There are three kinds of non-functions that can be called as functional objects: arrays, stack-groups, and microcode-entry functions.

11.2.4.1 Arrays.

Calling an array (DTP-ARRAY), if that array is not a named-structure, is equivalent to doing AREF with the array as the first argument and the other arguments as the indices. In fact, that is how it is implemented. Calling a named-structure is described in the section on Indirect Functions.

11.2.4.2 Stack-Groups.

Calling a stack-group (DTP-STACK-GROUP) uses the function SYS:CALL-STACK-GROUP from the support vector to resume that stack-group. (See the sub-section on Support Vector.)

NOTE

Stack-groups are described from the user's point of view in the Explorer Lisp Reference manual and internally in the section on Stack Groups.

Multiple arguments and values are not yet supported, but all the hooks are in place. Currently, SYS:CALL-STACK-GROUP is passed all the arguments and the stack-group itself, with the stack-group last, and returns one result.

11.2.4.3 Microcode-Entry Functions.

A microcode-entry function (DTP-U-ENTRY) is exactly what its name states, a function that is a direct entry into the microcode, which is interpreted by the hardware. There are very few microcode-entry functions (about ten); most of the time, entry to the microcode is accomplished by macroinstructions. (See the section on Macroinstructions). However, the standard macroinstruction mechanisms do not allow for operations that accept an indefinite number of arguments (a &REST arg). Microcode-entry functions overcome this limitation by using the standard function-calling mechanism to determine the number of arguments and acting accordingly; microcode-entry calling will be discussed in more detail in the section on U-Entries.

NOTE

Interpreted versions of misc-ops and aux-ops (See the sub-section on misc-ops) are generated at build time. They are simply compiled functions that invoke the appropriate instruction. This is a difference from VM1, in which all interpreted versions of misc-ops were microcode-entry functions.

11.2.5 Obsolete Functional Objects.

There are several functional-object data-types in VM1 that have been dropped in VM2:

* DTP-STACK-CLOSURE has been replaced by DTP-LEXICAL-CLOSURE. Both implement lexical closures (Section on Lexical-Closure), but the newer implementation is a complete redesign that corrects many flaws and offers some new features and improved performance.

* DTP-ENTITY and DTP-SELECT-METHOD have been eliminated. Entities were a variant of dynamic closures that bound SELF as well as their own bindings. Select-methods were essentially a list of operations and functions. When called, the first argument was taken as the operation to look up the function. An entity with a select-method as its functional part was the forerunner of the instance in the class system that predated the flavor system. The only holdover in VM2 is the Lisp macro DEFSELECT, which will now expand into a different structure with the same functionality.

11.3 HISTORY AND MOTIVATIONS

VM2 function-calling differs considerably from its VM1 predecessor. VM1 function-calling used what we have always called an "upside-down" stack layout: a call-block would be "opened" by pushing some state words and the function, then the arguments would be pushed; the last argument would "activate" the call-block, actually starting the function call. Multiple-value calls and other special calls would push additional information words ("ADI" words) before opening the call-block. VM2 function-calling, in contrast, uses a "right-side-up" stack layout, pushing all arguments before doing any sort of call. There is no such thing as an open call-block. Also, there are no ADI words; all information supplied at call time is kept in one word, and other information is kept in a fixed number of registers.

The compiled-function object (called a FEF for historical reasons) is changed as well. The VM1 FEF carried a list, the argument-descriptor list (ADL), that described the type, position, and initialization of every argument and local. Later work added the "fast-arg-option" word, a condensed form of the ADL for simpler cases. Release 1 of the Explorer system modified the header word of the FEF to contain even more condensed information for even simpler cases. The VM1 FEF also contained information about any special arguments, which would be bound by the function-calling microcode. In VM2, the ADL and special-binding information have been eliminated. The microcode initializes all locals and optional arguments to NIL, and the

compiler generates code for non-NIL initializations and for special-bindings. All information about the number of arguments and locals is present either in the header word or, if there are too many arguments or locals for the counts to fit there, in an optional word called the long-args word. Unlike VM1, the long-args word is not always present. Also, the name of the function, which in VM1 was kept in the FEF itself, has been moved to the FEF's debugging-info (see the section on FEF LAYOUT).

The guiding principles of VM2 function-calling have been simplicity and speed. VM1 function-calling is horrendously complex because it developed from a design intended to be perfectly general, but which had all sorts of speed hacks added on later. All this complexity makes it extremely slow in the slow cases and not very fast in the fast cases. The open/active call-block idea has also lost its appeal: most of the current function's state is kept on the stack where it requires an extra cycle to access, probably for historical debugging reasons, and the necessity of checking for open call-blocks adds overhead to most stack operations. Our guidelines in designing VM2 function-calling were:

* Keep all the state of the current running function in registers.
* Invert the stack: replace the open-call-block/active-call-block distinction with the simpler scheme of pushing all the arguments first, then doing the call.
* Do not copy the arguments (this guideline had an important influence on the timing of operations in the microcode).
* Let the compiler do all the hard work: non-NIL initializations of optional variables, special-bindings.
* Simplify the FEF: for the common cases, keep the information needed by the microcode in the header word; for the less-common cases, use one other word.
* Simplify the state: most of the important call and return state is kept in one word.

The remaining sections will describe VM2 function-calling in detail.

11.4 STACK LAYOUT

A snapshot of the PDL is shown in Figure 11-1. There are several call-frames shown. C is the current running function and its frame is shown in most detail. C was called by B, which was

itself called by A. Note how the state for B is between C's arguments and locals. C's state is not shown, because it is in the five registers M-CALL-INFO, M-ARGUMENT-POINTER, M-LOCAL-POINTER, M-FEF, and LOCATION-COUNTER. The values of M-ARGUMENT-POINTER and M-LOCAL-POINTER are indicated in Figure 11-1 by the two arrows marked "argument-pointer" and "local-pointer." LOCATION-COUNTER is described in the section on Macroinstructions; note that when the registers are pushed on the stack, its offset from the beginning of the function is saved instead.

A function is called by pushing its arguments and executing a CALL instruction. The CALL instruction pushes NILs for any unsupplied optional arguments, pushes the state of the calling function, pushes NILs for any locals of the called function, sets up the state registers, then causes macroinstruction execution to continue at the first instruction of the new function.

There are a few areas of added complexity:

* If there are any optional arguments, the CALL instruction leaves the count of the supplied optionals on top of the stack when it finishes. The intent is that the function's first macroinstruction will use this count to initialize the unsupplied optionals (this macroinstruction is frequently a DISPATCH instruction). The function is free to ignore this count, since it will not interfere with anything; it is likely to be ignored if all optional arguments default to NIL.

* If the function is lexpr-funcalled (applied, in Common Lisp terms), the last argument is a list that is spread into its individual elements at call time. In VM1, the list was spread unconditionally by a (MISC) %SPREAD D-LAST, which spread the list and activated the call-block at the same time. In VM2, we try to be more efficient by spreading the list during the CALL instruction: If the function has a &REST argument and enough arguments are supplied that part or all of the lexpr-list would be collected in that &REST argument, we do not spread that part of the list, storing it directly into the &REST argument instead.

* If the object being called is a lexical closure, the closure's environment object is temporarily pushed on the stack between the time the closure is taken apart and the time the environment is stored into its reserved local (see the section on Closures).

* If the object is being called with a self-mapping-table (see the section on Instance-Calling) supplied, the self-mapping-table is supplied on the stack above the arguments and is picked up before any optional-argument

processing.

* If the object is being called with a macroinstruction call-destination of D-TAIL-REC, it means that this call is destructively tail-recursive.

NOTE

Non-destructive tail recursion uses the destination D-RETURN, which has little special effect at call time, though much at return time.

In destructive tail-recursion, the callee's arguments are copied down over the arguments of the caller, obliterating the caller's frame, and the callee's state is modified so that it returns directly to its caller's caller. This kind of tail recursion is known colloquially as "frame-eating" tail recursion, and is enabled by setting compiler optimization levels to emphasize size and speed and deemphasize safety.

NOTE

In contrast, VMI destructive tail-recursion was decided at runtime, controlled by a user-settable variable and some instructions that, allowed or prohibited tail-recursion when it might not otherwise appear possible. It did not work very well, and added the overhead of checking to every return.

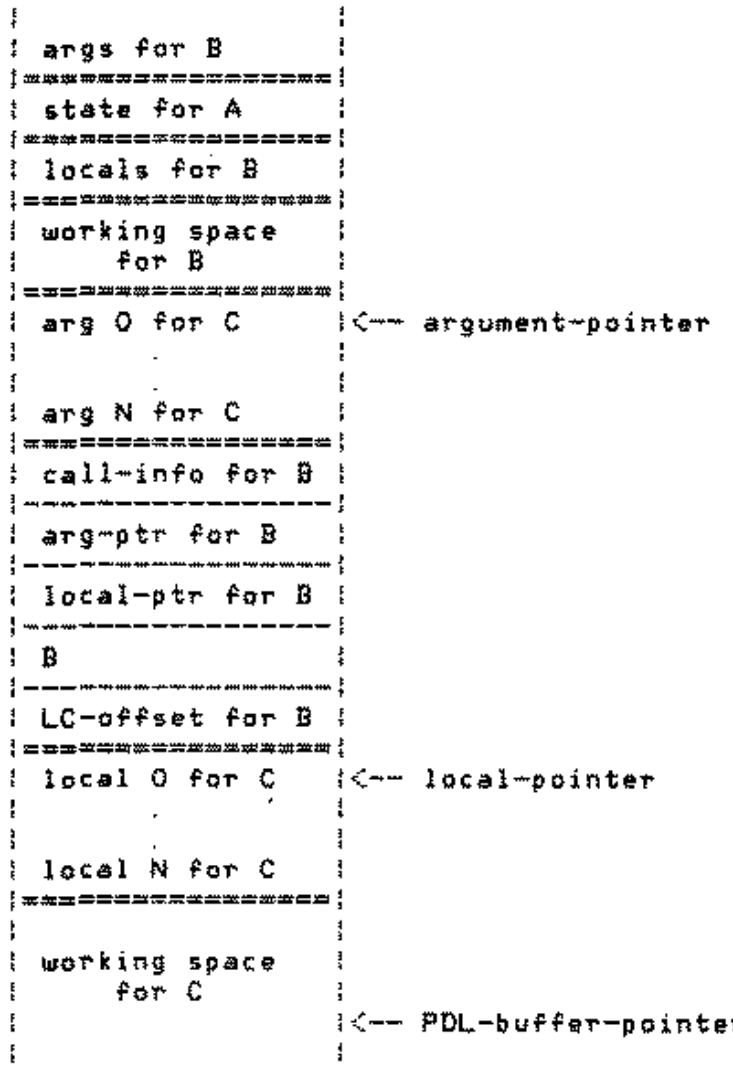


Figure 11-1 Call-Frame Layout

11.5 THE CALL-INFO WORD

Following the design guidelines shown in the section on motivations, most of the call and return information is supplied and kept in one word, the call-info word, a fixnum whose pointer field is laid out as pictured in Figure 11-2. In the generic CALL, the call-info word is pushed on the stack after the arguments, though there are several short-form call instructions that use a call-info word derived from the instruction. There

are three types of fields in the call-info word: call fields, return fields, and state fields.

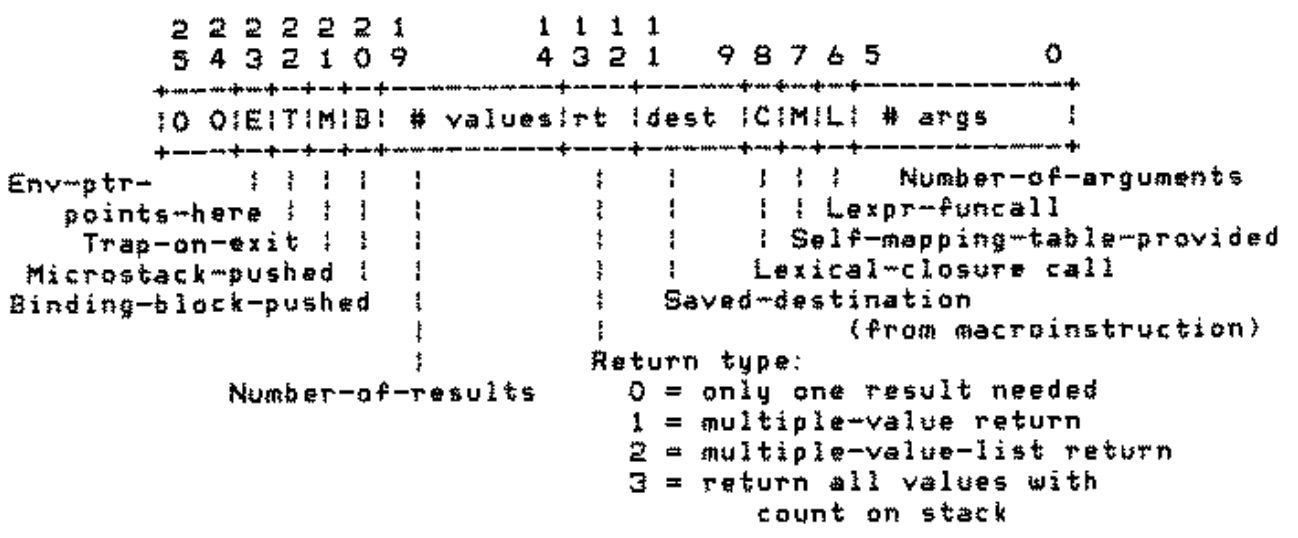


Figure 11-2 Call-Info Word Layout

11.5.1 Call Fields.

Call fields are those containing information about the kind of call. These are usually supplied by the compiler when it generates a call-info word, or by the short-form call instructions in their constant call-info words. Call fields are all in the low nine bits of the call-info word, so most call variations can use a call-info word generated by a PUSH-NUMBER instruction, as long as multiple values are not needed. The call fields are:

* The number-of-arguments field, %%call-info-number-of-arguments, which allows for up to 63 arguments (though more actual arguments can be supplied using lexpr-funcall).

* The lexpr-funcall flag, %%call-info-lexpr-funcall-flag, set when doing a LEXPR-FUNCALL or APPLY.

* The self-mapping-table-provided flag, %%call-info-self-map-table-provided, set when supplying the self-mapping-table on the stack.

* The lexical-closure-call flag, %%call-info-lexical-closure-call, which, strictly speaking, is not a call field, because it is set by the microcode at call time.

This flag indicates that a lexical closure is being called and that its environment object is on the stack.

11.5.2 Return Fields.

Return fields contain information about the kind of return that the caller expects from the call. These fields are supplied at call time, but are copied from the caller's call-info word if the macroinstruction destination is D-RETURN or D-TAIL-REC. There are two fields:

* The number-of-results field, %%call-info-number-of-results, which allows for up to 63 values.

* The return-type field, %%call-info-return-type, which indicates the form of returned values:

  - %only-one-result-needed (0): the caller only wants one result. This is special-cased for speed.

  - %normal-return (1): 0 to 63 values, as a block. When the function returns, there will be this many values left on the stack for the caller (or ultimate caller). Excess returned values will be trimmed and missing values will be filled in with NILs.

  - %multiple-value-list-return (2): all the returned values will be collected into a list, which will be returned as a single value.

  - %return-all-values-with-count-on-stack (3): the number-of-results field will be ignored and all values will be returned. The number of values will be left on the stack. This form of return is needed for situations involving indefinite numbers of values, usually associated with throws and unwind-protects, where, due to tail-recursion, it is not possible to determine how many values to return at call time. Calls with this return-type are often followed by RETURN-N instructions.

11.5.3 State Fields.

State fields are fields used by the microcode to keep track of the state of the frame, usually to preserve information from the call or from the execution of the function until return time, when it will be needed. The state fields are:

* The saved destination, %%call-info-saved-destination:

the destination field from the macroinstruction is copied here; the field is in the same position as it is in the macroinstruction so it can be copied in one microinstruction with selective-deposit. The field is three bits wide instead of two because the microcode recognizes a fifth destination not available from macrocode: D-MICRO. This destination is used when "calling out" from microcode to macrocode (See the section on Calling "Out").

* The "marked-frame" flags, including:

  - Environment-pointer-points-here, %%call-info-env-ptr-points-here: set when a lexical-closure object is created that uses this frame in its environment (See the section on Closures); indicates that this frame must be "unshared."

  - Trap-on-exit, %%call-info-trap-on-exit: manipulated by the debugger commands C-X, M-X and C-M-X and the function breakon; causes an exit-trap when returning from this frame.

  - Microstack-pushed, %%call-info-microstack-pushed: set when the microstack is saved on the special PDL; usually set by calling out to macrocode, so that returning from the macrocode function will restore the microcode's state.

  - Binding-block-pushed, %%call-info-binding-block-pushed: set when a special-binding is done; indicates that there is a block of bindings to be undone when returning.

11.6 FEF LAYOUT

The compiled-function object, the FEF, is shown in Figure 11-3. Every FEF contains both boxed and unboxed words. The boxed words are collectively called the "FEF header" and contain information about the function and any constants that it needs. The unboxed words contain the macroinstructions.

The words in the FEF header are:

1. Word 0 has type DTP-FEF-HEADER and contains the following fields:

   a. Special form flag (%%FEF-HEADER-Special-Form, 1 bit) --- indicates whether there are any &QUOTE or &FUNCTIONAL arguments.

b. SUBST flag (%%FEF-HEADER-Subst, 1 bit) --- set if the function was defined by DEFSUBST.

c. Self-mapping table required (%%FEF-HEADER-Self-Mapping-Table, 1 bit) --- when set, a self-mapping-table must either be provided or fetched during the call.

d. Call type (%%FEF-HEADER-Call-Type, 3 bits):

...%fef-call-simple (0) = simple call (no optionals or locals)

...%fef-call-optionals (1) = optional arguments

...%fef-call-lo als (2) = local variables

...%fef-call-rest (3) = &REST arg (with or without other locals)

...%fef-call-optionals-and-locals (4) = optional args and local variables

...%fef-call-optionals-and-rest (5) = optional and &REST args

...%fef-call-unused (6) = [unused]

...%fef-call-long (7) = use the longs-args word.

e. Number of optional arguments (%%FEF-HEADER-Number-Optional-Args, 3 bits).

f. Number of required arguments (%%FEF-HEADER-Number-Args, 4 bits).

g. Number of local variable slots (%%FEF-HEADER-Number-Locals, 4 bits).

h. Offset of the word containing the first instruction (%%FEF-HEADER-Location-Counter-Offset, 10 bits).

2. Word 1 is a fixnum specifying the length of the FEF in words (used by the storage management code, see the section on Storage Management).

3. Word 2 points to the debugging info, which is a structure of type SYS:DEBUG-INFO-STRUCT. This structure has five slots: the name of the function, its true arglist, its interpreted definition if there

is one (often useful for functions which are declared INLINE), its local-map (which describes the layout of the local variables in the frame), and a plist which contains other fields, such as :MACROS-EXPANDED, :DOCUMENTATION, and :VALUES.

4. The long-args word is optional. It is needed when the counts will not fit in word 0, and its presence is indicated by the Call Type (%FEF-CALL-LONG). It contains:

   a. Optional arguments flag (%%fef-long-args-optionals, 1 bit) --- set if there are optionals.

   b. Local variables flag (%%fef-long-args-locals, 1 bit) --- set if there any locals.

   c. &REST arg flag (%%fef-long-args-rest-arg, 1 bit) --- set if there is a &REST argument.

   d. Minimum number of arguments (%%fef-long-args-min-args, 6 bits) --- the number of required arguments.

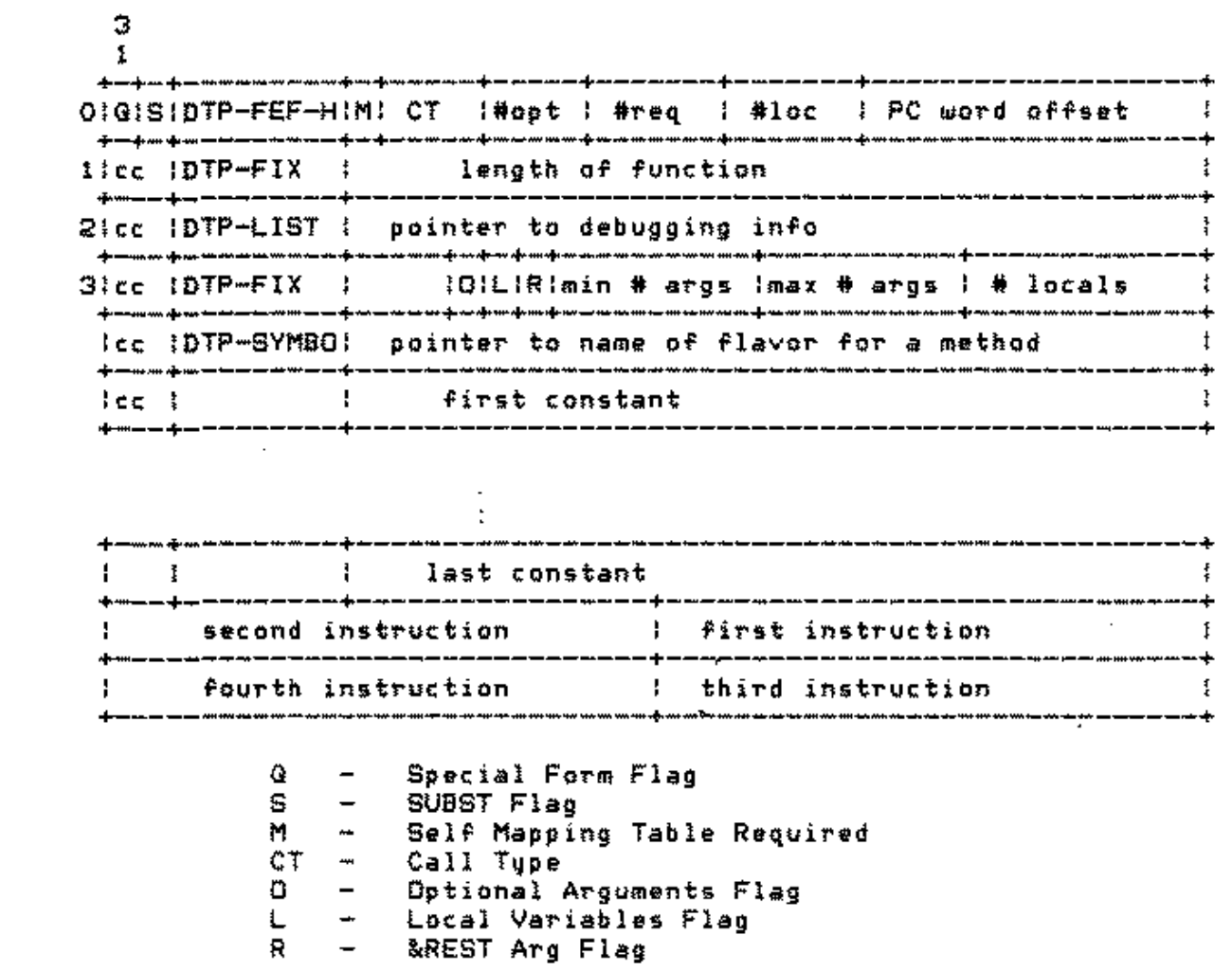
   e. Maximum number of arguments (%%fef-long-args-max-args, 6 bits) --- the sum of the number of required arguments and the number of optional arguments. The &REST argument, if present, is not counted here.

   f. Number of local variable slots (%%fef-long-args-number-of-locals, 7 bits).

      The maximum/minimum representation of the arguments (instead of the optional/required form in word 0) is a historical holdover, as is the seven-bit locals field (only 6 bits worth are accessible, but the old stack-closure code used to allocate some of its storage using this field).

5. The flavor name is also optional, and is present when the self-mapping table bit is set in word 0. If the flavor-name word is present, but the long-args word is not, the flavor-name word will be word 3; when both are present, it will be word 4.

The remaining header words are constants used in the function, indicated in disassembled code by FEFn, where n is the offset from the beginning of the FEF. Many of these constants will be external-value-cell pointers to value-cells and function-cells of symbols.



Q - Special Form Flag
S - SUBST Flag
M - Self Mapping Table Required
CT - Call Type
O - Optional Arguments Flag
L - Local Variables Flag
R - &REST Arg Flag

Figure 11-3 The Layout of a FEF

11.7 CALLING A FEF

11.7.1 The Instructions.

There are several kinds of CALL macroinstructions, depending on the grouping chosen. They may be divided into mainops and auxops (see the section on Macroinstructions) or general and special-case, and there are shadings in each division. The instructions

are:

CALL-0 through CALL-6:
These seven instructions all call a function with the number of arguments specified in the instruction, to return one result, nothing fancy. They all take the standard "register" and offset for the function, but are coded to favor FEF-relative arguments that are external-value-cell pointers.

CALL-N:
Similar to CALL-0 and friends, but takes the number of arguments from the top of the stack.

(AUX) COMPLEX-CALL:
The most general of the call instructions, COMPLEX-CALL takes both the function and a call-info word on the stack. This instruction actually exists in four forms to express the four call destinations: COMPLEX-CALL-TO-INDS, COMPLEX-CALL-TO-PUSH, COMPLEX-CALL-TO-RETURN, and COMPLEX-CALL-TO-TAIL-REC.

(AUX) APPLY:
This special-case instruction takes the function from the stack and lexpr-funcalls it with one argument, to return one result. It exists in four forms just like COMPLEX-CALL.

(AUX) LEXPR-FUNCALL-WITH-MAPPING-TABLE:
This special-case instruction takes the function and a self-mapping-table from the stack, and lexpr-funcalls the function with one argument and self-mapping-table provided, to return one result. It exists in four forms just like COMPLEX-CALL.

Other instructions may be added later.

11.7.2 The Actions.

The pattern of actions taken by the various call instructions resembles a tree: there are several beginnings, all of which merge at various points into a single main path, which itself branches out into several subroutines. All the auxop calls follow the same path after five or six microinstructions to pick up the function and figure out the call-info word. The mainop calls have the FEF-relative path broken out separately, but merge quickly. They do not handle the complex situations that the auxop calls do. The auxop calls merge with the mainop calls before dealing with the arguments and locals. With all that merging and separating in mind, here is the generalized sequence of operations of the call macroinstruction:

1. Find the function, the new argument-pointer, the new

call-info word, and the number of supplied arguments; all must be set up before dispatching on the data-type of the function. If the self-mapping-table-provided bit is set in the call-info word, pick up the new self-mapping-table.

2. Dispatch on the data-type of the function to the appropriate handler. At this point, the function is guaranteed to be in MD, the new argument-pointer in M-G, the new call-info word in M-F, and the number of supplied arguments in M-H. Most functional objects will eventually return here, having led to a compiled-function (DTP-FUNCTION) object; this dispatch pushes its own address, and falls through on DTP-FUNCTIONs. The other routines that extract functions from objects must not disturb most registers.

3. Calculate the location-counter offset from the previous function. Since the location-counter always points at the next instruction, this offset will be used to continue the previous function when this one returns. Read the header word of the FEF. Check for PDL-buffer overflow. Check for lexical-closure calls, if the bit is set in the call-info word, pick up the environment object from the stack and save it for later. If metering is turned on, record this function entry. Check for lexpr-funcalls, and spread the last argument if necessary; if part of the lexpr-list matches up with a &REST argument in the function, do not spread it only to collect it again, but make the list the &REST argument directly. If the trap-on-calls bit is set, trap now. note If the call-type of the function is %FEF-CALL-LONG, we do not try this nicety, because determining the existence of a &REST argument requires another memory reference that is not convenient at this point.

4. Handle the call-destination: If the call-destination is D-INDS (D-IGNORE) or D-PDL (D-STACK), do nothing. If the call-destination is D-RETURN, copy the return fields (see the section on Return-Fields) from the previous function's call-info word. If the call-destination is D-TAIL-REC (D-REALLY-TAIL-RECURSIVE), copy the return fields like D-RETURN, copy the call-destination and all the state from the previous function, and copy the arguments down over the previous function's frame, so that a return from this function will return to the previous function's caller instead of the previous function itself. Note that the call-destination is a two-bit field that is in two different places in the instruction, one for the mainops and one

for the auxops.

5. Take care of the arguments, checking for too few or too many, pushing NILs for any unsupplied optional arguments, and setting up the stack-list for any &REST argument (all arguments are pushed with cdr-code CDR-NEXT, so &REST-argument handling also must change the last one to CDR-NIL (see the section on Cdr-Codes)).

6. Advance the instruction stream. Call and return instructions do not use the mechanisms of the main macroinstruction loop, but instead include those mechanisms in the code, so that the next step can be hidden under the memory reference here. Note that only page-faults and interrupts may be checked for here, not sequence-breaks, because of the incomplete state of the call frame.

7. Push the five state words of the previous function. The old argument-pointer, local-pointer, and location-counter offset are all pushed as fixnums; the previous function and its call-info word are pushed as is, except that the call-info word is pushed with cdr-code CDR-ERROR to make call-frame tracing easier.

8. Take care of the locals, which involves pushing NILs for them, storing any lexical-closure environment object into its reserved local, placing any &REST argument in local 0, setting up the new local-pointer, and leaving the number of supplied optionals, calculated in step argument-handling, atop the stack. If there are non-NIL defaults, the first instruction of the function will probably be a dispatch-type instruction that will use that number to decide which optionals to initialize.

9. Pick up the first macroinstruction word and decode it, starting execution of this function.

11.8 CALLING ANYTHING ELSE

11.8.1 Calling the Interpreter.

Whenever a CALL instruction encounters a list, stack-list, or locative, it treats it as an interpreted function. The microcode builds a faked call-frame for the list-call, then collects a list of all the arguments and passes the function list and the argument list as the two arguments to the function SYS:APPLY-

LAMBDA, which it finds through the support vector (see the section on Support-Vector). It calls SYS:APPLY-LAMBDA with a destination of D-RETURN, so the result of the call will also be the result of the list-call. The only unusual details involve the possible need to diddle the cdr-codes of the arguments on the stack to make a stack-list of the arguments.

11.8.2 Calling an Instance.

An instance (DTP-INSTANCE) is a pointer to a word whose data type is DTP-INSTANCE-HEADER. The instance itself is the words following the header, while the header word points to the instance descriptor. All instances of the same flavor have headers pointing to the same instance descriptor, which contains the information common to all those instances.

Calling an instance first binds the variable SELF to the transported instance then looks in the instance descriptor for any other special-bindings to do, then looks in the instance descriptor for the method's function. If the function is not an array, it simply calls it (in case of a non-hash-table implementation).

NOTE

We may eventually make SELF and SYS:SELF-MAPPING-TABLE into lexical variables.

If it is an array, it is treated as a hash-array, and the first argument is the key. The modulus of the hash-array must be a power of two for the simple hashing algorithm used, which simply looks at the low bits of the symbol that is the key. Each entry has three words: key, locative to function, and mapping-table. If the key does not match, the microcode looks at the next entry, etc, until it either finds a match or an entry whose key is unbound (DTP-NULL) (this is different from the usual hashing algorithm). When it finds a match, it binds SYS:SELF-MAPPING-TABLE if the mapping-table slot is non-NIL, then dispatches on the data type of the new function.

If no matching entry is found, or if the key or the hash-array turns out to be gc-forwarded, the microcode calls out to SYS:INSTANCE-HASH-FAILURE, which will rehash the hash-array if necessary, or signal an error if the key is truly not in the table.

11.8.3 Calling a Microcode-Entry Function.

The pointer field of a DTP-U-ENTRY function is an index into the MICRO-CODE-ENTRY-AREA, a table of microcode entries. Entries in the table are either fixnums, indicating true microcode entries; NIL, indicating invalid entries; or other objects, which are the definitions for microcode entries that are not presently microcoded. The fixnums that indicate true microcode entries are themselves indices into the MICRO-CODE-LINK-AREA, a table of information about the microcode routine indicated by the microcode entry. The MICRO-CODE-LINK-AREA is built by the microassembler and lives in the microcode band.

The microcode builds a faked call-frame on the stack, then checks the arguments using two fields in the link-area data:

* The %%microcode-entry-args field indicates how many required arguments the microcode entry takes. This field is exactly equivalent to the number-of-requireds fields in the FEF header and long-args word.

* The %%microcode-entry-rest bit is 1 if the microcode-entry function takes a simulated &rest argument. If so, any number of arguments may be supplied; if not, the maximum number of arguments is the number in %%microcode-entry-args.

One complication arises from the fact that microcode-entry functions are under no compulsion to obey the normal conventions for dealing with arguments. In particular, they are likely to pop the arguments off the stack during their actions. To cope with this behavior while preserving a call-frame in case of error, the arguments are copied to the top of the stack before calling the microcode routine.

The final step in calling a microcode-entry function is to jump to the beginning of the microcode routine. The starting address is in the %%microcode-entry-index field in the link-area data. Before the jump itself, the microcode sets up return addresses so that the end of the microcode routine causes a function return. At the moment, microcode-entry functions may only return one value (while some miscops do return multiple values, they are not microcode-entry functions).

11.8.4 Calling "Out".

"Calling out" is the term used for any function call that is originated from the microcode. The microcode calls "out" from the microcode level to the macroinstruction level, hence the name. The main difference between calling out and normal calling is that since calling out is done from microcode, the microcode state must be preserved. The most important part of the microcode is the micro-pc return stack (microstack or UPCS, for short), which is saved as a block on the special PDL during the

call. The %%call-info-microstack-pushed bit in the call-info word is used to keep track of which calls saved the microstack.

The call destination D-MICRO exists for the use of calling out. Calls with this destination do not read a macroinstruction when they return; they simply do a microcode return to the routine that called out, and continue executing microinstructions. When the microcode routine finishes, only then is the next macroinstruction fetched and executed. This destination makes it possible for the microcode to call out to Lisp, receive results, and continue, possibly indefinitely. Calling out can also use the other call destinations, though D-RETURN is the most common.

The most common reason to call out is that the calculation is too complicated to do in microcode; for example, arithmetic on rational and complex numbers is handled by the two functions SYS:NUMERIC-ONE-ARGUMENT and SYS:NUMERIC-TWO-ARGUMENTS, which the arithmetic microcode calls out to. Another example is EQUAL and EQUALP, which are recursive in microcode. Since the microstack is only 64 words, there is a real danger of overflowing it when comparing deep structures. EQUAL and EQUALP solve this problem by calling out to themselves, thus saving the old microstack and obtaining more room to work. The function-calling microcode also occasionally calls out. When calling an interpreted function, it calls out to SYS:APPLY-LAMBDA.

There are other ways that the microcode uses calling out. The most unusual is that at boot time, the initial function is called out to by the boot microcode. This way, there is always a valid frame at the bottom of the initial stack-group's stack, and if this function is returned from (though unlikely, it can be forced from the debugger), the microcode will loop back and call it again. It is thus also unnecessary to make a special check for the bottom of the stack when returning.

11.8.5 The Support Vector.

Calling out is inextricably linked with the support vector. The support vector is an area of memory known by the microcode that contains some symbols needed by the microcode. Table 11-1 lists the contents of the support vector. Many of these symbols are functions that are called out to.

SYS:NAMED-STRUCTURE-INVOKE is used as described in the section on Array-Call when calling a named-structure as a function. SYS:APPLY-LAMBDA is used when calling interpreted functions (see the section on Interpreted Functions). SYS:EXPT-HARD, SYS:NUMERIC-ONE-ARGUMENT, SYS:NUMERIC-TWO-ARGUMENTS, SYS:LDB-HARD, and SYS:DPB-HARD are used for the difficult cases of some arithmetic functions and for byte fields too large for the microcode LDB and DPB. EQUAL, EQUALP, and EQUALP-ARRAY are used when the microcode recursion becomes too deep, as described in

the section on Equal-Call-Out. SYS:DEFSTRUCT-DESCRIPTION is used when doing TYPEP of named-structures; it is the property of the named-structure-symbol that contains all the internal information about the structure. SYS:INSTANCE-HASH-FAILURE is the function that calling an instance will call out to if the key is not found or if the hash-table needs rehashing (see the section on Instance-Calling). PRINT, *PACKAGE*, and the unbound marker are currently unused.

SYS:INSTANCE-INVOKE-VECTOR is an array of keyword symbols that represent messages to send to an instance when trying to do several of the basic Lisp operations to that instance. For example, trying to take the CAR of an instance will cause the :CAR message to be sent to that instance. The contents of the instance-invoke vector are :GET, :GETL, :GET-LOCATION-OR-NIL, :CAR, :CDR, :SET-CAR, and :SET-CDR. While :GET and its cousins are relatively useful in that they make GET a more generic function, the others seem to be remnants of an attempt to do everything using message-passing, since almost no flavor accepts the messages.

Table 11-1 Contents of the Support Vector

Index	Function
0	PRINT
1	SYS:NAMED-STRUCTURE-INVOKE
2	SYS:DEFSTRUCT-DESCRIPTION
3	SYS:APPLY-LAMBDA
4	EQUAL
5	*PACKAGE*
6	SYS:EXPT-HARD
7	SYS:NUMERIC-ONE-ARGUMENT
8	SYS:NUMERIC-TWO-ARGUMENTS
9	unbound marker
10	SYS:INSTANCE-HASH-FAILURE
11	SYS:INSTANCE-INVOKE-VECTOR
12	EQUALP
13	EQUALP-ARRAY
14	SYS:LDB-HARD
15	SYS:DPB-HARD

11.9 RETURNING

11.9.1 Basics.

Returning is governed by the return instruction itself, and by three fields in the call-info word,

* Number-of-results field
* Return-type field
* Saved-destination field

The return instruction indicates how many values are being returned, the number-of-results field indicates how many are wanted, the return-type field indicates how the values are to be returned, and the saved-destination field indicates where the values are to be returned. The return-type field and the saved-destination field are adjacent in the call-info word so the return microcode can dispatch on the combination of the two and decide quickly what is to be done. There are five call destinations, four of them accessible from compiled code. D-PDL is the "normal" destination; the results of the call will be left on the stack, first one farthest from the top. D-INDS (called D-IGNORE in VM1) means that no values are to be received; the function was either called for effect or just to set the "indicators." The first value is placed in the "indicators" (the register M-T), and a succeeding conditional branch can look at that value in deciding whether or not to branch.

NOTE

In VM1, multiple-value calls allocated a block on the PDL to receive the values and arranged that this block would be on top of the stack after the return. In VM2, there is no reserved block; the values are simply copied over the frame of the function that is returning.

D-RETURN and D-TAIL-REC are the two forms of tail-recursion. D-RETURN is non-destructive tail-recursion; it behaves exactly like a D-PDL call until return time. The return microcode will scan back through the stack until it finds a D-INDS or D-PDL frame, and return directly to that frame, through all intervening D-

RETURN frames. D-TAIL-REC is destructive (colloquially, "frame-eating") tail-recursion: at call time, it assumes the state of the caller's caller and copies the arguments over the caller's frame, thus obliterating it. It actually splices out intervening D-TAIL-REC frames at call time, so that at return time it does not have to do the scanning that D-RETURN returns do. This offers substantial space savings and some time savings, at the expense of making the code difficult to debug. The compiler will use D-TAIL-REC instead of D-RETURN when it is safe to do so and the compiler optimizations are set such that speed and space are emphasized and safety is deemphasized. Both D-RETURN and D-TAIL-REC calls copy the return fields (see the section on Return-Fields) from the current call-info word into the new call-info word at call time, D-TAIL-REC because that is part of assuming its caller's caller's state, D-RETURN because it is handy to have that information available before it starts scanning.

The fifth call destination is D-MICRO, used when the microcode calls out to Lisp (see the section Calling-Out). It acts exactly like D-PDL, except that where returns involving the other four destinations restore state and continue at the next macroinstruction, a D-MICRO return will restore state and continue at the next microinstruction.

There are two basic return instructions, known in the microcode as RETURN-1 and RETURN. RETURN-1 is special-case code for handling single-value returns, because they are by far the most common, while RETURN handles the multiple-value cases. All the return macroinstructions (which will be discussed later) are built on these two. The return-types are described in the section on Return-Fields, but they can be grouped into pairs as far as returning is concerned. Only-one-result-needed and multiple-value-list situations are similar in that only one value has to be handled, while normal-return and return-all-values situations may require the movement of several values. As a result, RETURN-1 will end up handling returns of the first pair, even though from the actual macroinstruction it may not appear so.

Adding to the complexity of returning is the necessity of cleaning up after the function. This breaks up into two parts, cleaning up after the frame and making sure the values are safe to return. Cleaning up after the frame means looking at the marked-frame flags (see the section on State-Fields) and acting accordingly. If the environment-pointer-points-here flag is set, some lexical closure has included part of this frame in its environment, and the values must be copied out of the stack before the frame can be removed. The binding-block-pushed flag indicates that this frame has a matching frame on the special PDL whose bindings must be restored. The trap-on-exit flag is a debugging aid that allows the user to examine the frame being exited and the values being returned. The microstack-pushed flag indicates that a block on the special PDL contains some saved

microstack words that must be restored.

To make sure the values are safe to return, we pass them through the "return barrier." This barrier makes sure that stack-lists are copied out to the heap and that lexical closures do what they have to do, which involves setting the environment-pointer-points-here flag in appropriate frames and copying out the values from this frame that are needed in the closure's environment object. Returning passes every value through the return barrier for every frame returned from.

11.9.2 Details.

Returning a single value is fairly straightforward. RETURN-1 dispatches on the combination of the return-type and the saved call-destination, picking up the top of the stack into M-T along the way (every return sets the indicators). It then passes the value through the return barrier and checks the marked-frame flags, and restores the state of the preceding frame. Next, it advances the instruction stream to pick up the macroinstruction at which it will continue (see the step on instruction-stream in the Call-Action section), and during the memory reference checks for PDL-buffer overflow. Finally, it picks up the macroinstruction and macro-decodes it while pushing the returned value on the stack.

The destination and return-type differences are small:

* D-INDS returns do not push the result.

* Multiple-value-list returns make a list of the result or results first, then treat the list as a single value.

* D-RETURN returns continue the steps above for successive frames until one of them has a call-destination of D-INDS or D-PDL.

* D-MICRO returns back up the location-counter and re-advance the instruction stream to restore the contents of the macroinstruction buffer to what it was at the time of the call-out, but do not macro-decode it.

* If the return-type was normal-return, NILs are pushed if necessary to match the number of results wanted.

* If the return-type was return-all-values-with-count, a 1 is pushed above the single value.

Multiple-value returns are slightly more complex. All the simple cases that happen to be single-value returns, such as multiple-value-list or only-one-result-needed situations, use the single-value code described above. The harder cases have to handle

passing more than one value through the return barrier, trimming excess values if more are being returned than are wanted, and pushing the count if the return-type is return-all-with-count. The hardest case is D-RETURN, because the obvious is not the most efficient. Multiple-value D-RETURN returns leave the values right where they are until the ultimate frame is found, then copy them all at once. They also do the trimming of values early and the augmentation with NILs late, because that minimizes the number of values that have to be passed through the return barrier and copied. Otherwise, the algorithm is the same as the single-value return.

There are a lot of return macroinstructions:

* RETURN is a mainop that returns one value and taking it from any mainop source.

* The auxops RETURN-0 through RETURN-63 return the indicated number of values, taking them from the top of the stack, first value deepest.

* The auxop RETURN-N is like RETURN-0, etc, but it takes the number of values from the top of the stack before taking the values from there.

* The auxop RETURN-LIST takes a list of values and returns them as if they were individual values. It could be emulated by spreading the list, pushing the length of the list, and using RETURN-N, but it tries to be smart about multiple-value-list returns and excess values.

* The auxops RETURN-T and RETURN-NIL return T and NIL, respectively. We seem to do that a lot, so they are special-cased.

* The auxop RETURN-PRED returns T if the indicators are non-NIL, else it returns NIL. The auxop RETURN-NOT-INDS is the opposite, returning T if the indicators are NIL and T otherwise.

11.10 CATCH AND THROW

Catches and throws are among the most radically redesigned parts of VM2, relative to VM1. In VM1, catches were open call-blocks for the function *CATCH, with special ADI to record the restart-PC and other things, whose first argument was the catch tag. Throws would scan the list of all open call-blocks to find the open catches. Multiple-value throws would essentially do a multiple-value return on all but the first value, then do a single-value throw with it. In VM2, things are different. Catch-blocks are entirely divorced from call frames. Throws scan

a list of the open catch-blocks, and multiple-value throws are handled cleanly. This section will describe the VM2 catch and throw mechanisms.

11.11 CATCH

One of the design goals in VM2 catch and throw was to make catches cheap and push the expense onto throws, figuring that many catches are set up, but few throws are taken. Another was to clearly delimit the scope of each catch (in VM1, catches are frequently closed by popping the open call-block off the stack, just like discarding any other data). These goals led to the following rules:

1. All catches are opened with the auxop %OPEN-CATCH or a variant.

2. All catches are closed with the auxop %CLOSE-CATCH. It may not be assumed that returning from a function will close any open catches remaining within its frame, for example. The only implicit closing of a catch is when it is thrown through on the way to another catch.

3. Unwind-protects are exactly like any other catch, except that (a) they are closed with the auxop %CLOSE-CATCH-UNWIND-PROTECT, and (b) the restart-PC of normal catches points at the matching %CLOSE-CATCH, while the restart-PC of an unwind-protect points at the first instruction of the undo-forms.

4. The %CLOSE-CATCH-UNWIND-PROTECT of an unwind-protect will detect if the catch was thrown to, and if so will continue the throw. The throw does not retain control throughout the throw; rather, there are a series of throws that add up to the complete one, each bounded by the intervening unwind-protects.

A catch-block contains six words:

1. %catch-block-catch-tag, a Lisp object that is the tag for this catch. Tags are matched using EQ, so it is usually a symbol, but is occasionally a fixnum or stack-list (condition-handling frequently uses stack-list tags).

2. %catch-block-restart-pc, a fixnum that is the relative PC of the instruction at which execution is to resume if this catch is thrown to.

3. %catch-block-number-of-results, which is either a fixnum indicating the number of results that will be accepted by this catch if thrown to, NIL meaning that it wants all the values with the count on top, or T

meaning to collect all the thrown values into a list. The function of this word is analogous to a combination of the return-type and number-of-results field in the call-info word (see the section on Return-Fields) when returning from a function. This field does not come into play unless this catch is the target of a throw.

4. %catch-block-special-pdl-level, a locative recording the level of the special-binding PDL at the time the catch was opened. This word is necessary to undo any special bindings that may have been done inside the catch, but it is only used if the catch is thrown to.

5. %catch-block-saved-catch-pointer, a locative or NIL that indicates the previous open catch. A special microcode register, M-CATCH-POINTER, points to the most recent open catch, and the catch-blocks form a singly-linked list. If there are no open catches, M-CATCH-POINTER is NIL. M-CATCH-POINTER is saved in each stack-group.

6. %catch-block-tag-being-thrown, initially NIL, but filled with the tag being thrown to when this catch is involved in a throw. It is this word that enables %CLOSE-CATCH to decide whether to continue the throw.

The first two words, the tag and the restart-PC, are pushed before opening the catch. %OPEN-CATCH pushes a 1 as the number of results and %OPEN-CATCH-MV-LIST pushes T, but %OPEN-CATCH-MULTIPLE-VALUE requires that its number of results be pushed for it. The other three words are set up by the open-catch microcode.

%CLOSE-CATCH is designed to be fast. It assumes that if there is no need to continue the throw, the preceding instructions have left the values in the desired state, and it just splices the catch block out of the stack, leaving the values on top. It also assumes that any specials bound within the catch have been unbound, so it does no unbinding. Its algorithm is:

First check the tag-being-thrown word. If it is NIL, there has been no throw. If it is non-NIL, but the catch-tag is not T, there has been a throw, but this catch is not an unwind-protect, so the throw does not need to be continued. In these two cases, simply update M-CATCH-POINTER and copy everything above the catch-block down over it, splicing it out. It is thus very fast to open and close catches if they are not needed.

NOTE

NIL is therefore not allowed as a tag argument to THROW.

If there was a throw and the catch-tag is T, then this catch is an unwind-protect, and the throw needs to be continued. If the tag-being-thrown word is T, then the throw was actually an *UNWIND-STACK (see the section on Unwind-Stack).

NOTE

T is also not allowed as a tag argument to THROW.

If the number-of-results word in the catch-block was NIL, then the number of thrown values is atop the stack; otherwise, all values above the catch block are taken to be thrown values. In any case, the throw is continued by jumping into the throw code, which is described next.

11.12 THROW

Throwing is a two-pass process. In the first pass, the throw microcode searches the list of catch-blocks for one whose tag matches the throw-tag. In the second pass, the throw microcode unwinds the stack, cleaning up frames as if it were returning through them (see the section on Returning), leaving the thrown values above the target catch-block.

The simple description above leaves out all the complexity of the operation. For instance, the first pass also looks for unwind-protects, because although it is required that a matching catch exist, the throw may actually be to an intervening unwind-protect. Also, a catch with a tag of NIL will catch any throw; such a catch is produced by the macro CATCH-ALL. CATCH-ALL differs from UNWIND-PROTECT in that there are no undo-forms and the throw is not continued. Unlike VM1, CATCH-ALL does not receive any extra values like the throw-tag; it is simply a catch that catches everything.

Another area of complexity is hardware PDL buffer cache management. Returns via D-RETURN (see the section on Returning) also unwind one or more frames as they return through them. Adjacent call frames are known to be relatively close together, and D-RETURN chains are likely to be short, so is makes sense to refill the PDL-buffer whenever needed in the course of the

return. Catch-blocks, by contrast, tend to be more widely separated, so frequent PDL-buffer refills are not usually a good idea. The alternative is to access the PDL-buffer through the virtual-memory interface, which is much slower but which does not involve reading so many words. The compromise that the microcode actually uses is to work through the PDL-buffer for the frames that are there, but to use the virtual-memory path for the deeper frames that are not, refilling only at the end to bring the target frame into the PDL-buffer.

A third complexity is the handling of values. Once the throw has unwound to the target frame, it must check the catch-block for the number of values wanted and the form they are to take. It is at this point that missing values are augmented with NILs, excess values are trimmed, and lists are built for the multiple-value-list case. Values are passed through the return barrier at the beginning of the second pass, after it is known where they are going to be copied to, and it is only done once (unlike returning, which passes them through for each frame).

NOTE

Multiple-value returns, when the return-type is multiple-value-list, build the list first and treat it as a single-value return. While it might also be a good idea for throws, it is not done at this time.

Just like returning, there are two kinds of throw instructions, THROW, which throws a single value, and THROW-N, which throws any number of values, taking the count from the top of the stack. Both instructions are auxops, and the compiler generates both from the Lisp THROW special form, choosing the appropriate one from context. In structure they are very similar, the difference being that THROW can pick up its single value into M-T, while THROW-N must leave its values on the stack undisturbed while it unwinds. THROW can therefore restore all the state of the target frame and refill the PDL-buffer before putting its value on the stack, while THROW-N must copy its values, possibly through the virtual-memory path, to their destination before refilling.

11.13 UNWIND-STACK

*UNWIND-STACK is a mating of throw and return. It exists for stack-manipulating programs such as the debugger, and as such is a subset of the *UNWIND-STACK of VM1. In VM1, *UNWIND-STACK was the general form of THROW: THROW was *UNWIND-STACK with the count and action arguments NIL. *UNWIND-STACK takes four

arguments: tag, value, count, and action. The tag argument is for partial compatibility with the VM1 *UNWIND-STACK; it must be T. The value argument is the same as the value argument to THROW; it may be any Lisp object. The count argument is the number of call frames to unwind; it must be a fixnum or NIL. The action argument is called when the unwinding is finished; it must be a functional object or NIL. Either the count or the action must be non-NIL; the case of both NIL was equivalent to THROW in VM1 but is an error in VM2.

NOTE

VM2 *UNWIND-STACK behaves exactly as the VM1 version did when this argument was T; the functionality that is not implemented in VM2 is the use of other tags.

*UNWIND-STACK will unwind the frames on the stack, doing all the cleanup it would do if it were throwing through them. It also cleans up after catches along the way. There are three possible situations:

* If the count is a fixnum, it is the number of frames to unwind. If the action is NIL, then the value will be returned from the last frame unwound.

* If the count is NIL, it indicates that all frames in the stack should be unwound. The action must then be non-NIL, and will be called with one argument, the value, after all the frames have been unwound. The action is not permitted to return in this situation. It is often useful for the action to be a stack-group; the debugger frequently uses it in this way.

* If both the count and the action are non-NIL, then the action will be called, with the value as its argument, after count frames have been unwound. The action may return, and its values will be returned as if from the last frame unwound. This case is rarely used.

*UNWIND-STACK honors unwind-protects like throws do. It stops at the unwind-protect catch, leaves the value, the count, and the action atop the stack, and writes T into the tag-being-thrown word. Since T is not permitted as the tag argument to a throw, CLOSE-CATCH recognizes it as the sign of an *UNWIND-STACK that needs to be continued, and calls *UNWIND-STACK. Since the tag-being-thrown slot is the top slot of the catch-block, the four arguments are therefore right on top of the stack.

SECTION 12

Closures

12.1 INTRODUCTION

A closure is a functional object with some extra information attached to it. In simplest terms, it is a function coupled with some representation of the environment in which it was created. The Explorer system supports two kinds of closures: Dynamic Closures which capture some part of the dynamic variable binding environment in which they are created, and Lexical Closures which capture the surrounding lexical (textual) variable environment in which the closure function was written.

Each of these closures is a separate primitive functional object having a dedicated data type and a unique storage format for its environment structure. Each also implies a different set of support operations which must occur when the closure is funcalled. This section covers these topics.

12.2 DYNAMIC CLOSURES

Conceptually, a dynamic closure object is a function and a remembered binding environment. The remembered binding environment is necessary to resolve possible binding conflicts caused by free references to variables in functions passed as arguments (funargs). The implementation of dynamic closures on the Explorer system is closely tied with the variable binding method chosen for the system.

Some implementations of Lisp use deep binding, where the binding stack can be thought of as an a-list of symbol/value pairs. In this case accessing a variable requires an ASSOC, and takes time proportional to the number of bindings on the a-list. In a deep binding implementation, there is one global binding stack (or one per process, in multi-processing systems). Dynamic closures in such a system can "close over" the entire environment simply by saving a pointer to the current global binding a-list when the dynamic closure is created.

Explorer system Lisp, like MacLisp, uses shallow binding. The goal of shallow binding is to keep the time needed to access a symbols value to a small constant: typically only that of a single memory reference. Conceptually, shallow binding can be

thought of as a scheme where there is a stack associated with each symbol which contains a history of the symbol's bindings. The top of the symbol's binding stack (which can be accessed quickly) always contains its current binding.

Shallow binding on the Explorer is a slight variation on this conceptual model. Each symbol's value cell (called the symbol's internal value cell since it is the internal physical location within the symbol data structure itself) always contains the current binding for the function being run. Previous bindings are saved on the binding stack (the Special PDL or SPECPDL) by the BIND primitive and are restored when the binding construct is exited. BIND does this by saving the actual contents of the symbol's internal value cell on the Special PDL along with a pointer (of type DTP-LOCATIVE) to the internal value cell.

Dynamic closure binding is a special kind of binding which may be shared; that is, other functions and dynamic closures in the same binding environment as the CLOSURE function can access and change the dynamic closure's closed over variables. This sharing is accomplished by using a new data type called DTP-EXTERNAL-VALUE-CELL-POINTER (EVCP). It can be thought of as a kind of "environment pointer," referring to the symbol's value when it was closure-bound.

Since bindings are associated with the symbol and not on a global a-list, dynamic closure binding on the Explorer system is on a per-symbol basis. The function CLOSURE takes two arguments: the first argument is a list of symbols (the symbols whose bindings are to be saved), and the second is a function object (such as a LAMBDA expression, or a compiled-code object). First, CLOSURE CDRs down its first argument, assuring that each of the symbols has an external value cell. Whenever it finds one which doesn't, it allocates a word from free storage, places the contents of the symbol's internal value cell into the word, and replaces the internal value cell with a DTP-EXTERNAL-VALUE-CELL-POINTER to the word. Then, CLOSURE allocates a block of 2*N+1 words of storage, where N is the length of CLOSURE's first argument. In the first word of the block, CLOSURE stores its second argument. Then for each symbol in its first argument, it stores a pointer to the internal value cell, and a copy of the symbols EVCP. Finally, CLOSURE returns an object of datatype DTP-CLOSURE which points at the block. This is the dynamic closure itself.

A symbol's internal value cell contains an EVCP only when its binding is being remembered by some dynamic closure; at other times the internal value cell just contains the symbol's value. The presence of the EVCP when a variable is closure-bound allows any modifications to the variable by functions or dynamic closures in CLOSURE's binding environment to be seen by the function closed over. An EVCP is treated in the usual manner (described above) by the BIND and UNBIND operations, but is treated as an invisible pointer by SET (the primitive function

for updating a symbol's value) and by SYMEVAL (the primitive function for accessing a symbol's value). The word pointed to by the EVCP is called the symbol's external value cell. Thus, SET and SYMEVAL access and modify the symbol's external value cell, while BIND and UNBIND refer to the internal value cell.

When a CLOSURE is invoked as a function, the first thing that happens is that the saved environment is restored: the contents of the internal value cells of the current environment are saved on the binding PDL. Then the EVCPs stored in the dynamic closure are placed in the symbols' internal value cells, restoring the saved environment. Finally, the function is invoked with the arguments passed to the dynamic closure.

12.3 LEXICAL CLOSURES

While a dynamic closure is a function and an environment which captures a set of dynamic bindings, a lexical closure is a function and a lexical environment which captures a set of variables shared between the function and its lexical parent (and/or any higher lexical grandparents). A dynamic closure takes a snapshot of the global, fluid dynamic binding environment at the point in time that it is created. The environment of a lexical closure, on the other hand, is neither fluid nor temporally changing. A lexical environment is fully defined when the functions are written. Hence all environment is statically determined at compilation time.

When discussing lexical closures it is convenient to distinguish between a closure function, which is the functional part of a lexical closure, and a closure parent function, or just parent function, which is a function inside whose textual context the lexical closure is defined. Any time a function defined within an outer (parent) function freely refers to a local variable name or argument name belonging to its parent function, the function is considered a lexical closure. In the SILLY-ADDER function shown in Table 12-1, SILLY-ADDER is the parent function, and the internal anonymous LAMBDA function is a lexical closure.

Table 12-1 Some Lexical Closure Examples

```
(DEFUN silly-adder (num)
   (funcall #'(lambda () (+ num 1))))

(DEFUN make-summer ()
   (LET ((sum 0))
     #'(lambda (n) (INCF sum n))))

(SETF (SYMBOL-FUNCTION 'add-to-sum) (make-summer))
(add-to-sum 5) ==> 5
(add-to-sum 2) ==> 7
(add-to-sum 6) ==> 13
```

However, such extremely simple creations of lexical closures as that in SILLY-ADDER can be completely "compiled out" and reduced to in-line code. Although the anonymous LAMBDA function accesses its parent's argument, the compiler can statically determine that neither it nor its shared binding of NUM needs to be protected or preserved outside of the execution environment of SILLY-ADDER itself. Hence, it can just compile into a simple (+ NUM 1).

On the other hand, the MAKE-SUMMER parent function creates a lexical closure which is returned as the value of a call to MAKE-SUMMER, so the internal LAMBDA function cannot be compiled out. Furthermore, it shares the variable SUM with its lexical parent MAKE-SUMMER, hence it is a lexical closure. Let's call the closure returned by a call to MAKE-SUMMER ADD-TO-SUM. Calls to ADD-TO-SUM now side-effect the SUM variable in its lexical parent. The lexical closure ADD-TO-SUM, then, must contain both a functional object to perform the computation, and an environment object which captures the referenced variables in its textual scope. The lexical closure environment data structure must provide links back to the defining lexical parent. Moreover, the defining lexical parent MAKE-SUMMER must have a way of keeping track of the closures it creates (since subsequent calls will create more closures). Each such closure must have a separate, protected lexical environment. Hence stack frames of lexical parent functions must have reserved slots for maintaining information about their offspring. The following discussion covers the data structures for maintaining these links.

NOTE

The implementation of lexical closures is quite different in Explorer Release 3 from earlier releases. The following discussion applies to Release 3 and later only.

12.3.1 Lexical Closure Implementation. The lexical closure object is represented by a word of type DTP-LEXICAL-CLOSURE which points to a cons cell whose car is the function and whose cdr is a locative to the environment. The layout of a lexical closure can be seen in Figure 12-1.

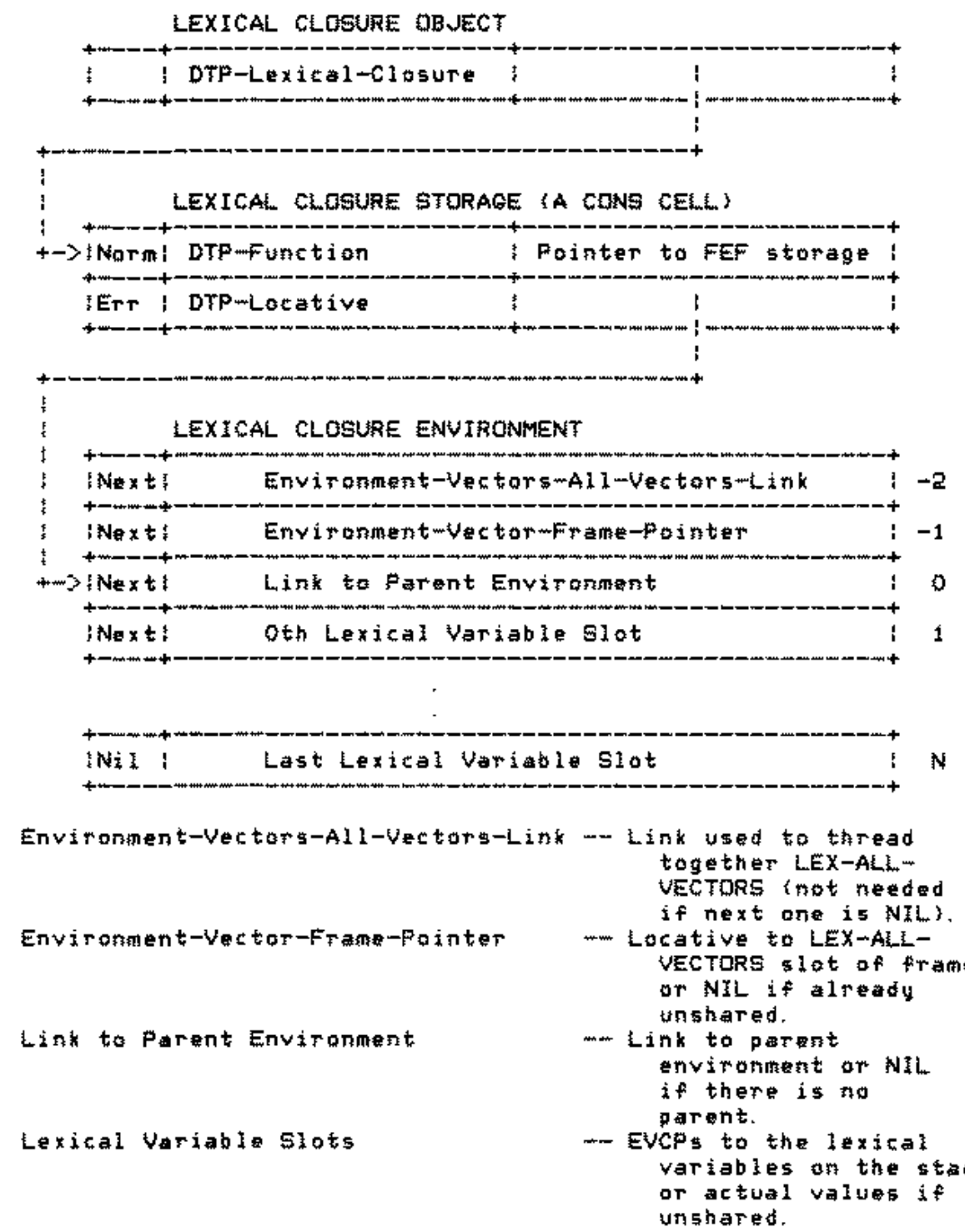


Figure 12-1 Lexical Closure Structure

12.3.2 Dedicated Locals.

The implementation of lexical closures makes use of up to four dedicated local variable slots the function's runtime stack frame. In all cases it is determinable at compile time exactly which of these locations need be dedicated/reserved within a function. Whenever a particular one (or more) of these is <u>not</u> required, its location is available for general local storage within the function. These locals, if present, have the following uses:

Offset	Name	Description
2	LEX-PARENT-ENV-REG	Parent lexical environment pointer and primary lexical "base register"
3	LEX-ENV-B-REG	Secondary lexical "base register"
4	LEX-CURRENT-VECTOR-REG	Current environment pointer
5	LEX-ALL-VECTORS-REG	Head of environment list

Note that local slot 0 is reserved for &REST args and local 1 to hold the mapping table in combined flavor methods. The symbolic names are in the SYSTEM package and are defined in file SYS:UCODE;LROY-QCOM.

Compile time conditions for dedicating locals:

LEX-PARENT-ENV-REG (Initialized by CALL microcode)

Dedicated if function is a closure or if function contains a MAKE-LEXICAL-CLOSURE macroinstruction.

LEX-ENV-B-REG (Initialized to NIL)

Dedicated if compiler determines that a second lexical base register is needed.

LEX-ALL-VECTORS-REG (Initialized to NIL)

Dedicated if any MAKE-LEXICAL-CLOSURE macroinstruction is emitted which indicates a push onto the environment list.

LEX-CURRENT-VECTOR-REG (Initialized to NIL)

Dedicated if function contains a MAKE-LEXICAL-CLOSURE macro instruction.

12.3.3 Calling a Lexical Closure.

When a lexical closure is called, the microcode will store the new environment pointer in LEX-PARENT-ENV-REG of the called function. This allows efficient access to the environment from within the function. This approach requires that all lexical closures dedicate LEX-PARENT-ENV-REG as an environment pointer. Furthermore, any function creating an environment must use LEX-PARENT-ENV-REG as an environment "parent" pointer. Simple functions (non closures) are called with no special treatment of LEX-PARENT-ENV-REG in the called function.

12.3.4 Free Lexical References.

References to non-local lexical variables must specify the variable being referenced and which higher lexical environment the variable belongs to. Such references can be compiled into one of the general form instructions LOAD-FROM-HIGHER-CONTEXT, STORE-IN-HIGHER-CONTEXT, or LOCATE-IN-HIGHER-CONTEXT (returns a locative to the variable). Each of these takes from the stack a context descriptor, which is a FIXNUM whose lower 12 bits are the offset within a particular lexical environment. This defines the variable itself. The next 12 bits specify which environment An environment value of 0 represents the immediate parent (taken from LEX-PARENT-ENV-REG); 1 represents the grandparent, and so forth.

The majority of non-local lexical references, however, are either to variables in the immediate lexical parent or in the lexical grandparent. Hence, most non-local lexical references are handled more efficiently by using MAIN-OP instructions with a special base register field which specifies "higher lexical context" addressing (register value 3). The 6-bit offset field of the instruction is then used divided into a 5-bit lexical environment offset and one bit which specifies the base register (0 = LEX-PARENT-ENV-REG, 1 = LEX-ENV-B-REG) These base registers (actually in the function's local block) are assumed to point to valid environment structures. The 5-bit displacement field allows reference to the first 32 slots of the lexical environments pointed to by LEX-PARENT-ENV-REG or LEX-ENV-B-REG. The compiler will revert to the <access>-HIGHER-CONTEXT form instruction only when the short form addresses cannot be used; that is, the lexical environment displacement is greater than 31 or when access to a third environment frame not covered by a base register is required.

It is left up to the compiler to determine which particular environment pointers will reside in LEX-PARENT-ENV-REG and LEX-ENV-B-REG at given points in the code. While it is unlikely that the compiler would choose to have other than the parent pointer

reside in LEX-PARENT-ENV-REG, it is quite possible that other than the grandparent pointer might reside in LEX-ENV-B-REG. (Remember that LEX-PARENT-ENV-REG must point to the lexical parent when MAKE-LEXICAL-CLOSURE is executed.) The compiler makes the choice based on the number of references to each lexical level.

Environment pointers are obtained by the instruction LOCATE-LEXICAL-ENVIRONMENT, which accepts an integer specifying the number of generations to go back (0 = parent, 1 = grandparent, etc.) and returns a pointer to the requested environment object on the PDL. This instruction, of course, begins its search at LEX-PARENT-ENV-REG. The simplest approach for the compiler would be to allocate LEX-ENV-B-REG on a global basis, although for some functions local allocation might win. The anticipated "typical" code sequence on entry to a closure would be something like:

```
     LOCATE-LEXICAL-ENVIRONMENT  1     ;Grandparent
     POP    LOCAL!3                    ;LEX-ENV-B-REG
     .
     .
```

after which we could have:

```
     .
     .
     PUSH   LEX-A!7             ;7th slot of PARENT
     .
     .
     POP    LEX-B!4             ;4th slot of GRANDPARENT
```

Should the compiler determine that a closure has no need of LEX-ENV-B-REG (as a base register) then it is free to use it as a normal local slot. The short form lexical reference instructions reduce the size of the code (1 instruction vs. 2) as well as reducing the time required to access grandparent (or higher) environments. The use of the LEX-ENV-B-REG base register allows us to incur the lookup cost only once; thereafter references cost the same as references to the parent environment.

12.3.5 Constructing Lexical Closures.

Lexical closure objects are constructed using the MAKE-LEXICAL-CLOSURE or MAKE-EPHEMERAL-LEXICAL-CLOSURE macroinstruction. These instructions take two arguments from the stack: the environment descriptor list and the FEF pointer to use. They return (on the stack) a pointer to the constructed lexical closure object. These instructions also use three implicit arguments, found in the invoking function's local slots, as follows:

1. LEX-CURRENT-VECTOR-REG is accessed to retrieve the pointer to the lexical environment to be used in the

constructed lexical closure. If LEX-CURRENT-VECTOR-REG contains NIL, then it is necessary to construct a new environment object according to the descriptor list mentioned above. In this case the pointer to the new environment object is stored into LEX-CURRENT-VECTOR-REG as a side effect of the macro instruction.

2. LEX-ALL-VECTORS-REG is used to maintain a list of environment objects constructed by the current function. See below for details.

3. LEX-PARENT-ENV-REG is used as the lexical parent pointer should it be necessary to construct a new environment object.

The two instructions differ in whether the (possibly new) environment object will be pushed onto the environment list at LEX-ALL-VECTORS-REG. For MAKE-LEXICAL-CLOSURE, the environment object (whether new or old) is pushed onto the list at LEX-ALL-VECTORS-REG, provided that it is not already the first element of the list. For MAKE-EPHEMERAL-LEXICAL-CLOSURE, no attempt is made to push the environment onto the LEX-ALL-VECTORS-REG list.

The decision as to whether an environment object needs to be added to the environment list is made by the compiler and is based on whether the closure being created can possibly outlive the current function's stack frame.

12.3.6 Lexical Environments.

Lexical environment objects are constructed as a side effect of the MAKE-LEXICAL-CLOSURE macroinstruction. Environment objects are always consed in the heap. Each object is a cdr-coded list containing a pointer to its parent environment followed by some number of slots which may contain either the value of a variable or an indirect pointer (EVCP) to the variable's value cell.

Value entries in a freshly constructed environment correspond to locals/args of the current function which are freely referenced and which the compiler has determined are "read only"; i.e., they are never stored into. In this case each constructed environment gets its own copy of the value of the local/arg rather than the EVCP used in the general case.

12.3.7 Environment Descriptor List.

The environment descriptor list is used as a template when constructing an environment object. The first element of the list is the number of slots to be created, and each remaining element is a FIXNUM which defines one lexical variable slot, in the format shown in Figure 12-2.

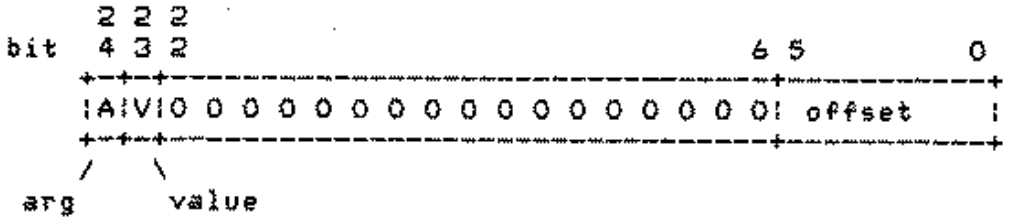


Figure 12-2 Lexical Variable Slot Descriptor

The sign bit is 0 for a local variable, or 1 for an argument. The next bit is the value/reference flag; when it is 0, the slot will hold a DTP-EXTERNAL-VALUE-CELL-POINTER to the variable; when 1, the value of the variable is copied directly into the environment slot. If the number of slots is greater than the number of of slot descriptors provided, then the remaining slots are initialized with a value of NIL and do not correspond to any local variables; the compiler uses such slots as value cells for lexical variables which are in excess of the maximum number of 64 that can be allocated in the stack.

12.3.8 Managing Environments.

We have already discussed the creation of new environments in our discussion of the operation of MAKE-LEXICAL-CLOSURE with respect to LEX-CURRENT-VECTOR-REG, although we did not indicate <u>how</u> LEX-CURRENT-VECTOR-REG might ever be set back to NIL once an environment had been constructed. The motivation for forcing a <u>new</u> environment object to be constructed is to allow representation of environments containing different instances of selected variables. To accomplish this also requires the use of the LEXICAL-UNSHARE instruction.

The LEXICAL-UNSHARE instruction creates an instance of a variable by copying its value (from the stack) into the heap and then relocating all references (from environments in the environment list) to point to the heap allocated value. References to this variable in <u>subsequent</u> environments will point (via EVCPs) into the stack, thus referencing a different instance of the variable. There is also a LEXICAL-UNSHARE-ALL instruction which can be used to unshare all of the local variables at once.

Thus we need to set LEX-CURRENT-VECTOR-REG to NIL whenever we have reason to unshare any of the variables of the environment. This is done implicitly by the LEXICAL-UNSHARE and LEXICAL-UNSHARE-ALL microcode.

The UNSHARE instructions will access the environment list via LEX-ALL-VECTORS-REG. This list is also accessed implicitly upon RETURNing from a frame when the <u>environment pointer points here</u>

(EPPH) flag is set in the frame's Call-Info word. This flag is set only when an environment object is pushed onto the environment list at LEX-ALL-VECTORS-REQ. The microcode must perform an UNSHARE-ALL operation when this flag is set.

COMPILER NOTE: Locals/args which are determined to be read only need not be unshared since copies of the values will have been distributed at the time the environment was constructed.

SECTION 13

Flavor Internals

13.1 INTRODUCTION

This description is intended for programmers familiar with Flavors; this is not introductory material. In order to introduce the principle components of the flavor system, a simplified example follows:

(DEFFLAVOR NEW-FLAVOR (IV1 IV2) (BASE) :GETTABLE-INSTANCE-VARIABLES)
This macro expands into code that creates a FLAVOR STRUCTURE where most of the flavor internals structures are stored. All the available information (Instance variables, dependencies, keywords) is put in there. This STRUCTURE is placed on the SYS:FLAVOR property of NEW-FLAVOR.

(DEFMETHOD (NEW-FLAVOR :MESSAGE-1) (A) (+ A IV1))
This macro expands into a function definition whose name is (:METHOD NEW-FLAVOR :MESSAGE-1). The method name and its function definition are going to be recorded in the METHOD-TABLE of NEW-FLAVOR.

(MAKE-INSTANCE 'NEW-FLAVOR)
This will first create the remaining flavor data structures needed for the instance to be functional (method-hash-table and mapping-table) (this is called composition). Next, an instance of NEW-FLAVOR will be created and returned.

(SEND INSTANCE :MESSAGE-1 7)
This will cause the microcode to access the flavor structure to get the function attached to :MESSAGE-1 [(:METHOD NEW-FLAVOR :MESSAGE-1) in that case] through the method-hash-table and call that method with the special variable SELF bound to instance.

13.2 FLAVOR NAMES

Flavor names are symbols which get interned as usual. The SYS:FLAVOR property holds the complete flavor description, contained in a structure of type FLAVOR. SYS:COMPILATION-FLAVOR can be used to get the flavor structure from the symbol; it can get the temporary flavor structure used by the compiler during a file compilation.

13.3 FLAVOR INSTANCES

An Instance is an object with type DTP-INSTANCE that points to a block of memory beginning with a word of type DTP-INSTANCE-HEADER. The pointer field of the header points to the FLAVOR STRUCTURE (see Flavor Structure section) where the microcode finds the METHOD-HASH-TABLE, the flavor name, etc. The function (SYS:instance-flavor <instance>) returns the flavor structure of an instance. The instance variables are stored right after the header word and can be accessed by offset using (%instance-ref instance index).

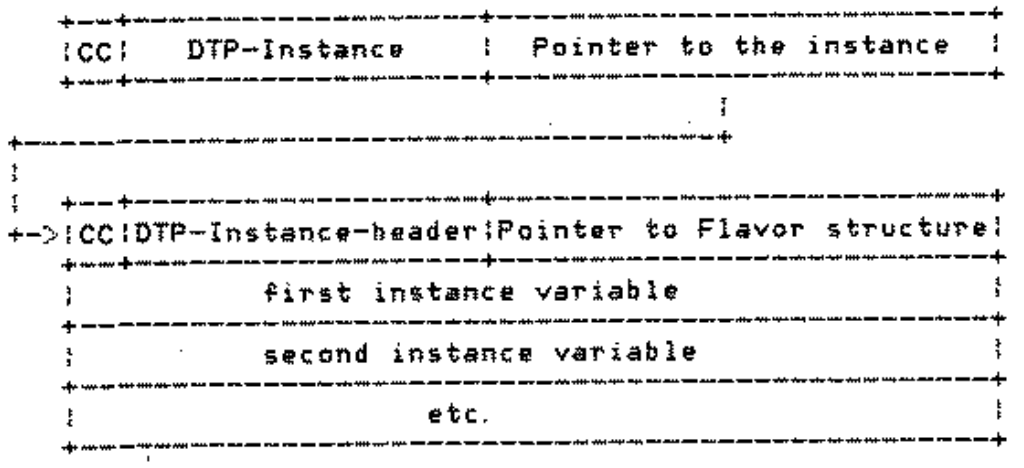


Figure 13-1 Flavor Instance

13.4 FLAVOR STRUCTURE

This structure is used at DEFFLAVOR time, compile time and runtime. It is shared by all the instances of that flavor. All of the inter-Flavor pointers are symbols, not structure references, this allows for flavor redefinition and renaming without pointer repair. Note that since the structure is used for everything, some slots are empty (NIL) if the flavor is not fully composed. In particular, if a structure has a DEPENDS-ON-ALL slot of NIL, it is assumed that its components are yet to be figured out. If a structure has a METHOD-HASH-TABLE slot of NIL, it is assumed that its methods are not composed yet. Its description is included in Table 13-2, so the programmer can figure out the accessor functions for each slot.

13.5 FLAVOR METHOD-TABLE (FLAVOR-METHOD-TABLE slot of Flavor structure)

This Flavor structure slot is used to store information about the methods. It is used to compose the methods and to do the function spec handling (for FDEFINE, FDEFINITION, FDEFINEDP, etc.). For the previous simple example the value of this slot would be:

```
((:MESSAGE-1
   NIL NIL
   ((:METHOD NEW-FLAVOR :MESSAGE-1)
    #<DTP-FEF-POINTER (:METHOD NEW-FLAVOR :MESSAGE-1) 20346350>
    NIL))
 (:IV2
   NIL NIL
   ((:METHOD NEW-FLAVOR :IV2)
    #<DTP-FEF-POINTER (:METHOD NEW-FLAVOR :IV2) 20346337>
    NIL))
 (:IV1
   NIL NIL
   ((:METHOD NEW-FLAVOR :IV1)
    #<DTP-FEF-POINTER (:METHOD NEW-FLAVOR :IV1) 20346326>
    NIL)))
```

Its format is: a list of message entries, each of those consisting of: (MESSAGE COMBINATION-TYPE COMBINATION-OPTION METH1 METH2 ....) where:

- MESSAGE is the message symbol.
- COMBINATION-TYPE is the type described by the DEFFLAVOR declaration :METHOD-COMBINATION (for example, :CASE :PROGN ...). If NIL, then :DAEMON is used.
- COMBINATION-OPTION is either :BASE-FLAVOR-FIRST, :BASE-FLAVOR-LAST or NIL (the default) (see DEFFLAVOR option).
- METHx is a list structure: (FUNCTION-SPEC FUNCTION-OBJECT PLIST) where FUNCTION-SPEC is the function-spec of the method, FUNCTION-OBJECT is the method and PLIST is a property list used to store the source filename, the previous definition, etc.

The Method-Table is updated in two places:

- At DEFFLAVOR time when there is a :method-combination keyword. An entry is made for that message, specifying the combination type and the

combination option.

- At FDEFINE time: any Method Fdefined will get an entry in the table. Note that FDEFINE will not modify the combination type and the combination option; by default they will be NIL.

13.6 INSTANCE VARIABLES ACCESS

There are three kinds of instances variables:

- SPECIALS: These are special variables whose value cell is forwarded to the correct place in the instance. The binding is done by the microcode at SEND time. These variables are produced by using the DEFFLAVOR keyword :SPECIAL-INSTANCE-VARIABLES.

- UNMAPPED: These variables are assumed to have a fixed place in the instance even when mixed with other components. Therefore, they can be referenced by direct offset in any method. These variables are produced by using the DEFFLAVOR keyword :ORDERED-INSTANCE-VARIABLES.

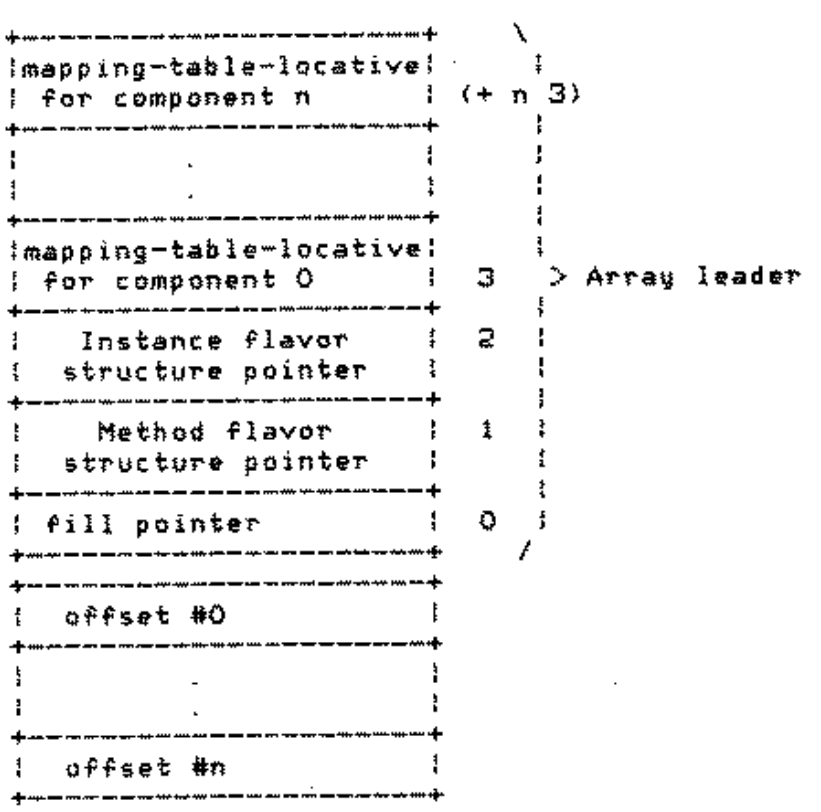
- MAPPED: These variables are not going to be at the same place from one flavor to the other because they are mixed with different components. However, their methods (except combined methods) are expected to work on any flavor that includes it. The microcode gets the offset within the instance through a mapping table, which is an array. Each instance variable is addressed using its corresponding mapping table index, and the array element at that index is the offset of the instance variable within the instance. These variables are produced by default.

13.7 FLAVOR MAPPING TABLES

Mapping table structure: It is an array with a fill-pointer. The length of the array is the number of mapped instance variables and each element of the array is the offset of the instance variable within the instance. When a combined method calls a method that comes from one of its component flavors, then it must pass another mapping table as well. The reference to that mapping table is contained in the leader of the primary

mapping table starting at array-leader index 3.

Table 13-1 Mapping Table



These mapping tables are found:

- In the METHOD-HASH-TABLE (see section FLAVOR METHOD HASH TABLE)

- In the COMPONENT-MAPPING-TABLE-ALIST slot of the flavor structure. This an alist whose key is the method-flavor and cdr is a locative to the right mapping table. It is used by GET-MAPPING-TABLE-LOCATION when building the method-hash-table.

- In the FLAVOR-COMPONENT-MAPPING-VECTOR which gathers them all, the contents of this accessor is an ART-Q-LIST vector containing locatives to the mapping tables.

The mapping tables are created when GET-MAPPING-TABLE-LOCATION finds out that they do not exist. So it will occur during execution of COMPOSE-METHOD-COMBINATION.

13.8 FLAVOR METHOD HASH TABLE (FLAVOR-METHOD-HASH-TABLE slot of Flavor structure

The method-hash-table is used mainly by the microcode to figure out what function to call for a given message. The FLAVOR-METHOD-HASH-TABLE slot points to a hash array whose key is the message symbol and values are: a locative to a cell that points to the method object and a pointer to the mapping table. Right now the hash function is just a LOGAND mask on the symbol pointer (as a consequence the hash-table size has to be an exact power of two). If the hash fails, the function SYS:INSTANCE-HASH-FAILURE is called from the microcode. That function will figure out if the hash table needs to be rehashed and if there were any forward references from the flavor structure slot to the real hash array. If this is the case, the forwarding pointer is snapped out and the function will retry. That prevents from having to follow slow forwarding pointers and not knowing about them.

Note that the FLAVOR-METHOD-HASH-TABLE slot is filled in at the last moment; therefore, it is essentially used at runtime. Any type of method lookup will use the method table instead because it is always up to date.

13.9 FLAVOR COMPOSITION (COMPOSE-FLAVOR-COMBINATION)

This function is invoked either at COMPILE-FLAVOR-METHODS time or at instantiation time. It must be completed before the Method Composition can take place. This operation looks at all the components of a flavor and if they are defined, finds out about all the instance variables that make up that instance. This operation must be completed before attempting to compose the methods. Upon its completion, the slot DEPENDS-ON-ALL will be non-null. The first step is COMPOSE-FLAVOR-INCLUSION which goes through the flavor components and returns an ordered list of all the components. Then the instance variables are gathered, taking into account the ordered instance variables.

13.10 METHOD COMPOSITION (COMPOSE-METHOD-COMBINATION)

This function is called right after the Flavor Composition is complete. This operation will figure out, for each message the right method to invoke. If it requires creating a combined method, it will do so. Upon its completion, the slot FLAVOR-

METHOD-HASH-TABLE will be non-null. First the "magic list" is created; it is similar to the FLAVOR-METHOD-TABLE but gathers all the non combined methods from all the component flavors. The message ordering can be altered by the DEFFLAVOR keyword :METHOD-ORDER. Within each message, methods are inserted so the default order is base-flavor-last. Once that magic list is built, any :default method that does not need to be there is removed, then the method combination is done for each message. The method combination handler (located on the SYS:METHOD-COMBINATION property) is invoked on the message sublist, returning the combined method for that message (see the Method Combination section). Next, the FLAVOR-METHOD-HASH-TABLE is updated or created putting the method and the mapping tables for the messages in the hash table. The mapping tables are created at that time if they did not exist before.

13.11 METHOD COMBINATION

The handlers for method combination are called by COMPOSE-METHOD-COMBINATION with the flavor and the magic list sublist for a given message. Its role is to return a method for the specified flavor and message. It will create the combined methods as necessary. The handlers select the daemon they need (:before :after ...), then they check the combined method with the function HAVE-COMBINED-METHOD. If it returns a method, then it means that there was an up-to-date combined method and that is the handler. If it returns NIL, then MAKE-COMBINED-METHOD is called and will create one.

13.12 HAVE-COMBINED-METHODS

This function will try to get a Combined method that matches the magic list for that operation by getting the combined method of each of the component flavors starting in BASE-FLAVOR-LAST order. The magic list used to create a combined method is either recorded in the COMBINED-METHOD-DERIVATION debug info field or in the function's plist.

13.13 INSTANCE VARIABLE ADDRESSING

The compiler produces symbolic references to the instance variables. At load time, the function FLAVOR-VAR-SELF-REF-INDEX is called and will return a fixnum that represents an offset and a flag, %%SELF-REF-RELOCATE-FLAG, that tells the microcode if it should go through the mapping table or not. If the offset is small enough (< 32) then the SELF addressing mode is used, otherwise a FEF-constant is used which will have a DTP-SELF-REF-

POINTER data type and its pointer field will be the offset. The value of the offset will be the position of that instance variable in the FLAVOR-UNMAPPED-INSTANCE-VARIABLES or FLAVOR-MAPPED-INSTANCE-VARIABLES slot of the structure. If the variable is not present in either of those, it is assumed mapped and pushed on FLAVOR-MAPPED-INSTANCE-VARIABLES. The mapping tables are updated if they exist, otherwise they will be created during execution of COMPOSE-METHOD-COMBINATION.

13.14 MAPPING TABLE REFERENCE

When a combined method calls a method for one of its component flavors, it passes a new mapping table for that component flavor. The DTP-SELF-REF-POINTER addressing mode is used, the offset represents the array-leader index (see how component mapping tables are referenced from the main mapping table in the mapping table section). A flag, %%self-ref-map-leader-flag, is set to distinguish between a mapping table reference and an instance variable reference.

13.15 METHOD INTEGRATION

In order to reduce combined method overhead, when the SPEED compiler switch is greater than the SAFETY compiler switch, the variable SYS:*INTEGRATE-COMBINED-METHODS* is set to T and the compiler will expand the daemons inline during execution of MAKE-COMBINED-METHOD (this function calls OPTIMIZE-METHOD-BODY-AND-ARGS which does the expansion based on SYS:*INTEGRATE-COMBINED-METHODS*.

13.16 COMPILE FLAVOR METHODS

When Compiling to a file, this macro will cause the creation of a new Flavor Structure in the compiler's environment and a COMPOSE-FLAVOR-COMPOSITION and COMPOSE-METHOD-COMPOSITION is performed on it. During the execution of COMPOSE-METHOD-COMPOSITION, any combined method being created will be dumped on the object file as a side effect. At load time, it will insure that the flavor is fully composed, checking the FLAVOR-DEPENDS-ON-ALL and FLAVOR-METHOD-HASH-TABLE slots.

13.17 DEFFLAVOR

The flavor structure is created at this point, with many slots set to NIL. FLAVOR-DEPENDS-ON-ALL and FLAVOR-METHOD-HASH-TABLE

are set to NIL so that MAKE-INSTANCE or COMPILE-FLAVOR-METHODS will know they have to compose that flavor.

13.18 DEFMETHOD

This macro will produce a function object that gets a :self-flavor declaration to declare which variables are going to be instance variables.

13.19 METHOD LOADING

When a method is loaded or fdefined, METHOD-FUNCTION-SPEC-HANDLER is invoked and will insert the method description in the METHOD-TABLE slot (see METHOD TABLE section). Any :FASLOAD-COMBINED type of method will be turned into a :COMBINED method and will supersede the old definition only if the old definition is not FEF-EQUAL to the new one; this is due to the fact that without method integration, the combined methods do not contain user code by themselves but only calls to the appropriate non- combined methods. Thus if the flavor composition did not change the old combined method, it is still valid even if the individual non-combined methods were changed.

13.20 MAKE-INSTANCE (INSTANTIATE-FLAVOR)

INSTANTIATE-FLAVOR will get the real flavor structure (chasing it through abstract flavors), get the area to cons in using the function stored in the INSTANCE-AREA-FUNCTION property of the flavor plist, check that the flavor is composed (DEPENDS-ON-ALL non-null), check that the methods are composed (METHOD-HASH-TABLE non-null) then calls the %ALLOCATE-AND-INITIALIZE-INSTANCE miscop that creates the instance structure with the instance variables set to DTP-NULL, then the instance variables will be initialized to values coming from the init-plist and the default-values defined at DEFFLAVOR time. Finally, the :INIT message is sent.

13.21 INSTANCE CALLING (SEND)

The microcode is responsible for the instance calling. It gets the method from the METHOD-HASH-TABLE (See Method Hash Table section). If the <mapping-table-flag> is set for that method then it will look for a mapping table as the second value in the hash entry, if it not there, it will lookup the COMPONENT-MAPPING-TABLE-ALIST. Given that, it will bind SELF to the instance, SELF-MAPPING-TABLE to the mapping table and call the

method.

Table 13-2 Flavor Structure Definition

```
(defstruct (FLAVOR :named :array (:constructor make-flavor)
                   (:alterant nil)
                   (:make-array (:area permanent-storage-area))
                   (:conc-name nil) (:callable-constructors nil)
                   (:predicate nil) (:copier nil))

  flavor-instance-size          ;1+ the number of instance
                                ;  variables
  flavor-bindings               ;List of locatives to instance
                                ;  variable internal value cells.
                                ;  MUST BE CDR-CODED!!
                                ;Fixnums can also appear.  They
                                ;  say to skip whatever number of
                                ;  instance variable slots.
  FLAVOR-METHOD-HASH-TABLE      ;The hash table for methods of
                                ;  this flavor.  NIL -
                                ;  method-combination not composed
                                ;  yet.  T means abstract flavor
                                ;  with COMPILE-FLAVOR-METHODS done.
  FLAVOR-NAME                   ;Symbol which is the name of the
                                ;  flavor.  This is returned by
                                ;  TYPEP.

  FLAVOR-COMPONENT-MAPPING-TABLE-ALIST  ;Alist of component
                                        ;  flavor names vs.
                                        ;  locatives into vector
                                        ;  containing mapping
                                        ;  tables.

;; End of magic locations known in microcode and GCDM.

flavor-local-instance-variables ;Names and initializations,
                                ;  does not include inherited
                                ;  ones.
flavor-all-instance-variables   ;Just names, only valid when
                                ;  flavor-combination composed.
                                ;Corresponds directly to
                                ;  FLAVOR-BINDINGS and the
                                ;  instances.
FLAVOR-METHOD-TABLE             ;Defined above.

;; End of locations depended on in many other files.

FLAVOR-DEPENDS-ON               ;List of names of flavors
                                ;  incorporated into
                                ;  this flavor.
flavor-depended-on-by           ;List of names of flavors
                                ;  which incorporate this one.
                                ;The above are only immediate
                                   dependencies.
```

```
  flavor-includes                ;List of names of flavors to
                                 ;  include at the end rather
                                 ;  than as immediate depends-on's.
  flavor-package                 ;Package in which the DEFFLAVOR
                                 ;  was done.
  FLAVOR-DEPENDS-ON-ALL          ;Names of all flavors depended on,
                                 ;  to all levels, including this
                                 ;  flavor itself.  NIL means
                                 ;  flavor-combination not
                                 ;  composed yet.  This is used by
                                 ;  TYPEP of 2 arguments.
  (flavor-which-operations ())   ;List of operations handled, created
                                 ;  when needed.  This is NIL if it
                                 ;  has not been computed yet.

  ;;This is the list of instance variables accessable from
  ;;this flavor which are mapped by mapping tables with this
  ;;flavor as the method-flavor.

  (flavor-mapped-instance-variables ())

  ;; Redundant copy of :DEFAULT-HANDLER property,
  ;;for speed in calling it.

  (flavor-default-handler ())
  (flavor-inittable-instance-variables ())   ;Alist from init
                                             ;  keyword to name
                                             ;  of variable
  (flavor-init-keywords ())      ;option
  (flavor-plist ())              ;Esoteric things stored here as
                                 ;  properties (see Table 13-3)
          )
```

Table 13-3 Flavor-Plist Properties

```
:ORDERED-INSTANCE-VARIABLES
:DEFAULT-HANDLER
:OUTSIDE-ACCESSIBLE-INSTANCE-VARIABLES
:ACCESSOR-PREFIX
:REQUIRED-INSTANCE-VARIABLES
:REQUIRED-METHODS
:REQUIRED-FLAVORS
:SELECT-METHOD-ORDER
:DEFAULT-INIT-PLIST
:DOCUMENTATION: NO-VANILLA-FLAVOR
:GETTABLE-INSTANCE-VARIABLES
:SETTABLE-INSTANCE-VARIABLES
:SPECIAL-INSTANCE-VARIABLES
:ABSTRACT-FLAVOR
:ALIAS-FLAVOR
:INSTANTIATION-FLAVOR-FUNCTION
:RUN-TIME-ALTERNATIVES or
```

:MIXTURE
SYS:RUN-TIME-ALTERNATIVE-ALIST is the alist of lists of flavors
vs. names we constructed for those combinations.
SYS:ADDITIONAL-INSTANCE-VARIABLES
SYS:COMPILE-FLAVOR-METHODS
SYS:UNMAPPED-INSTANCE-VARIABLES
SYS:MAPPED-COMPONENT-FLAVORS
SYS:ALL-INSTANCE-VARIABLES-SPECIAL
SYS:INSTANCE-VARIABLE-INITIALIZATIONS
SYS:ALL-INITTABLE-INSTANCE-VARIABLES
SYS:REMAINING-DEFAULT-PLIST
SYS:REMAINING-INIT-KEYWORDS
:INSTANCE-AREA-FUNCTION the one specified for this fl.
SYS:INSTANCE-AREA-FUNCTION the one to be used (maybe inherited)
:REQUIRED-INIT-KEYWORDS the ones specified for this fl.
SYS:REQUIRED-INIT-KEYWORDS all required ones incl. inherited.

NOTE

The slots stored as properties of FLAVOR-PLIST have accessor functions of the form FLAVOR-<property-name>, like FLAVOR-UNMAPPED-INSTANCE-VARIABLES.

The convention on these is supposed to be that ones in the keyword packages are allowed to be used by users. Some of these are not used by the flavor system, they are just remembered on the plist in case anyone cares. The flavor system does all its handling of them during the expansion of the DEFFLAVOR macro.

SECTION 14

Stack Groups

14.1 INTRODUCTION

This chapter explains stack groups, the building blocks of multiprocessing.

A stack group is the fundamental unit of computation in the Explorer. It is the main data structure behind the implementation of a process. Interrupt context-switching, co-routines, and generators are facilitated by the use of stack groups.

At all times, there is exactly one active stack group, which corresponds to the process currently being run on a time-sharing system. Although there is no time-sharing between users on the Explorer, it is still useful to support multiple processes. For example, when a message is received from the network, some other stack group is activated to handle it.

14.2 THE STACK GROUP DATA STRUCTURE

The term stack group refers to the fact that each process must have its own control stack for remembering function call/return data and arguments and local data, and each process must have a Linear Binding stack to save the values of special variables. A stack group is a pointer of datatype DTP-STACK-GROUP, which points to an array header word the same way a DTP-ARRAY would. The reason for using an additional datatype is so that any routine will always be able to distinguish a stack group array from all other arrays. The array also has its own array type, ART-STACK-GROUP-HEAD, for the same reason.

The stack group structure is an array with no body, only a leader. The array leader holds many relevant data including a pointer to the regular push down list, a pointer to another array holding the Linear Binding PDL for the stack group, the PDL pointers for both PDLs, and various saved microcode registers (the Linear Binding PDL is also called the Special PDL or SPECPDL; see description below).

A useful feature is that by binding appropriate special variables, such as the default cons area and debugging

information; an error handler can be made a function of which stack group is active; each may have its own. This is because each stack group has a separate Linear Binding PDL.

The array leader contains miscellaneous data related to running and maintaining a stack group in the system. This data is divided up into sections according to how the data is used. The static section contains data such as the stack group name, and size and limits on the PDLs. This data is set up when the stack group is created and doesn't normally change during the course of system operation. This information is loaded when the stack group is entered; but since it doesn't change, the data needn't be saved when the stack group is left. The debugging section has information that the error handler needs to determine what error has occured and how to restart the stack group. This information is read directly by the error handler as needed and will not be loaded or saved on stack group entry or exit. The next section of the stack group contains information used to determine which operations are valid on this stack group. A layout of the stack group data structure follows:

14.2.1 Static Section.

SG-NAME	Name of stack group for conversing with user.
SG-REGULAR-PDL	The array serving as the regular PDL of this stack group. It must be an ART-REG-PDL array.
SG-REGULAR-PDL-LIMIT	Max PDL pointer value before overflow.
SG-SPECIAL-PDL	Special PDL for stack group. It must be an ART-SPECIAL-PDL array.
SG-SPECIAL-PDL-LIMIT	Max special PDL pointer value before overflow.
SG-INITIAL-FUNCTION-INDEX	Position in regular PDL of the topmost function pointer cell (normally 3).

14.2.2 Debugging Section.

SG-TRAP-TAG	Symbolic tag corresponding to SG-TRAP-MICRO-PC. Determined by looking up the trap PC in the

	microcode error table. Properties of this symbol drive error reporting and recovery.
SG-RECOVERY-HISTORY	Available for complex SG debugging routines to leave tracks in for debugging purposes. (This may be done away with in the future.)
SG-FOOTHOLD-DATA	Structure which saves dynamic section of "real" SG when executing in the error handler foothold.

14.2.3 High Level Section.

SG-STATE	The stack-group state. This has fields describing the high-level state of this stack group. Stack group states are listed in Table 14-1.
SG-PREVIOUS-STACK-GROUP	SG which caused resumption of this one.
SG-CALLING-ARGS-POINTER	Pointer to argument-block which last called this SG.
SG-CALLING-ARGS-NUMBER	Number of args in last call to this SG.
SG-TRAP-AP-LEVEL	Used for stepping. When stepping compiled functions, will cause a STEP-BREAK trap when PDL pointer is below this virtual address.

14.2.4 Dynamic Section.

SG-REGULAR-PDL-POINTER	Saved PDL pointer, stored as fixnum index into SG-REGULAR-PDL.
SG-SPECIAL-PDL-POINTER	Saved special PDL pointer, stored as fixnum index

	into SG-SPECIAL-PDL.
SG-TRAP-MICRO-PC	Micro-address from which a trap was signalled. Used as a key to lookup the TRAP-TAG in the microcode error table.
SG-SAVED-M-FLAGS	Saved processor flags as fixnum. The flags are shown in Table 14-2.
SG-PDL-PHASE	The index into the hardware PDL buffer cache of the Q at the top of the PDL. This preserves index-phasing of the pdl buffer across SG switches.
SG-SAVED-VMA	VMA register in a locative. The data type is stored in SG-VMA-M1-M2-TAGS.
SG-VMA-M1-M2-TAGS	Tags of VMA, M-1, and M-2 packed into a fixnum.
SG-M3-M4-TAGS	Tags of M-3 and M-4 packed into a fixnum.
SG-AC-1	Pointer fields of "raw" accumulators packed into fixnums. Tags are in one of previous packed-tag regs.
SG-AC-2	
SG-AC-3	
SG-AC-4	
SG-AC-A	Accumulators ...
SG-AC-B	
SG-AC-C	
SG-AC-D	
SG-AC-E	
SG-AC-F	
SG-AC-G	

SG-AC-H	
SG-AC-I	
SG-AC-J	
SG-AC-K	
SG-AC-L	
SG-AC-Q	
SG-AC-R	
SG-AC-S	
SG-AC-ZR	
SG-AC-T	Result register, pseudo indicators.
SG-CATCH-POINTER	Typed pointer to catch block.

Table 14-1 Stack Group States

Value	Name	Meaning
0	SG-STATE-ERROR	Should never get this.
1	SG-STATE-ACTIVE	Actually executing on the machine.
2	SG-STATE-RESUMABLE	Reached by interrupt or error recovery completed. Just restore state and continue.
3	SG-STATE-AWAITING-RETURN	After doing a "legitimate" sg-call. To resume this, reload sg, then return a value.
4	SG-STATE-INVOKE-CALL-ON-RETURN	To resume this, reload sg, then simulate a store in destination-last. The error system can produce this state when it wants to activate the foothold or perform a retry.
5	SG-STATE-INTERRUPTED-DIRTY	Get this if forced to take an interrupt at an inopportune time.
6	SG-STATE-AWAITING-ERROR-RECOVERY	Immediatedly after error, before recovery.
7	SG-STATE-AWAITING-CALL	The SG is ready to be called.
8	SG-STATE-AWAITING-INITIAL-CALL	Initial state of the SG, before it has been called.
9	SG-STATE-EXHAUSTED	State when the SG has finished running.

Table 14-2 Processor Flags

Bit(s)	Name	Meaning
0	QBBFL	This flag is no longer used.
1-2	CAR-SYM-MODE	CAR of symbol mode: 0: error 1: error except (CAR NIL) is NIL 2: NIL 3: error
3-4	CAR-NUM-MODE	CAR of number mode: 0: error 1: NIL 2: error 3: error
5-6	CDR-SYM-MODE	CDR of symbol mode: 0: error 1: error except (CDR NIL) is NIL 2: NIL 3: property-list
7-8	CDR-NUM-MODE	CDR of number mode: 0: error 1: NIL 2: error 3: error
9	DONT-SWAP-IN	Flag for creating fresh pages.
10	TRAP-ENABLE	When set, enable error trapping.
11-12	MAR-MODE	1-bit = read-trap, 2-bit = write-trap
13	PGF-WRITE	Flag used by page fault routine.
14	INTERRUPT-FLAG	In a microcode interrupt.
15	SCAVENGE-FLAG	In scavenger.
16	TRANSPORT-FLAG	In transporter.
17	STACK-GROUP-SWITCH-FLAG	Switching stack groups.
18	DEFERRED-SEQUENCE-BREAK-FLAG	Sequence break pending but inhibited.
19	METER-ENABLE	Metering enabled for this stack group.
20	TRAP-ON-CALL	Trap on attempting to activate new frame.

14.3 SPECIAL PDL

The Special PDL, also known as the Linear Binding PDL, is used to hold saved bindings of special variables. The Explorer uses shallow binding, so the current value of any symbol is always found in the symbol's value cell. When a symbol is bound, its previous value is saved on the Special PDL, and the new value is placed in the value cell. When a stack group switch is performed, the current process's Special PDL is saved in the stack group.

The Special PDL also serves some other functions. When a micro-to-macro call is made, the processor micro-stack (UPCS) of the machine is stored there (this is needed because the hardware UPCS is of a small fixed size).

The Special PDL is block oriented. The blocks are delimited by setting the SPECPDL-BLOCK-START-FLAG in the first binding made in a block. The data type of the top word (last pushed) of a block determines what kind of block this is, as shown in Table 14-3.

SPECPDL-BLOCK-START-FLAG and SPECPDL-CLOSURE-BINDING are stored in the CDR-Code field of Q's on the Special PDL. The CDR-Code is not otherwise used on the Special PDL. SPECPDL-CLOSURE-BINDING (bit 31) indicates that this binding was made "before" entering the function (i.e., by closure binding, or by the binding of SELF for a method). SPECPDL-BLOCK-START-FLAG is bit 30.

A normal binding block is stored as a pair of Q's for each binding; the first Q is a locative pointer to the bound location, and the second is the saved contents of the location. Note that any location can be bound; usually these locations will be the value cells of symbols, but they can also be array elements, etc. (only of arrays of type ART-Q or ART-Q-LIST).

The processor micro-stack (UPCS) blocks are always pushed onto the Special PDL all at once, and so are never "open." However, the normal binding blocks are created one pair at a time. To keep track of this, when a macrocompiled function is running, the binding-block-pushed bit is set in the call-info-word if a binding block has been opened on the Special PDL. This assures that when a compiled function is done, the call-info-word will correctly reflect whether it has done any bindings that must be popped off the Special PDL. If the bit is set, all of the bindings of the top-most block of the Special PDL must be undone. If not set, it means that no pairs have yet been pushed.

Micro-to-micro calls can also cause bindings, and in order to keep that straight, a bit on the UPCS is set to indicate that a block was bound.

The Special PDL is pointed to by the location SG-SPECIAL-PDL-POINTER in the stack group and by A-GLBNDP when the stack group

is executing on processor. There is an area devoted to storing both Special PDLs and stack groups called the SG-AND-BIND-PDL-AREA.

Table 14-3 Special PDL Block Type

DATATYPE	USE
LOCATIVE FIXNUM	The block is a normal binding block. This is a block transferred from the processor micro-stack (UPCS). Each word in the block should be a fixnum containing the old contents of the UPCS. Only the active part of the stack stack is transferred.

SECTION 15

Storage Subprimitives

15.1 STORAGE SUBPRIMITIVES

NOTE

Unless otherwise noted, all symbols and functions named in this section are in the SYSTEM package. The byte specifiers can be found in SYS:UCODE;LROY-QCOM. Most of the subprimitives are microcoded MISC-OPs defined in SYS:UCODE;DEFOP. Some that are Lisp-coded are defined in SYS:KERNEL;STORAGE-MACROS and SYS:KERNEL;STORAGE-INTERNALS.

Subprimitives are functions which are not intended to be used by the average program, only by system programs. They allow one to manipulate the environment at a level lower than normal Lisp. Many primitive Lisp functions are implemented on the Explorer system using subprimitives to accomplish their work. In this sense, subprimitives often take the place of what would be individual machine instructions or low-level subroutines on other systems. In fact, many subprimitive functions are simple Lisp-callable interfaces to miscellaneous operation macro instructions (miscops) written in Explorer microcode. Others are written in Lisp, but violate normal system storage conventions in order to achieve efficiency.

Subprimitives by their very nature cannot do full checking. Improper use of subprimitives can destroy the environment. They come in varying degrees of dangerousness. Generally, those whose names begin with % can ruin the Lisp world just as readily as they can do something useful. Subprimitives without a % in their names are usually not directly dangerous. This section describes subprimitives that manipulate storage; these are among the most dangerous ones. The next section describes other subprimitives and low-level system variables.

NOTE

Because they are used at the lowest level of the Explorer system implementation, subprimitives may change in the way they function or even be removed entirely from the system without notice. Most programs should thus generally not use subprimitives directly.

15.2 DANGERS OF SUBPRIMITIVES

The most common problem you can cause using subprimitives is the creation of illegal pointers: pointers that are not allowed to exist in the machine state according to storage conventions. If you create such an illegal pointer, one of several things may happen. It might cause a system crash or a bad data type trap immediately. Or it might not be detected until much later when another part of the system (most likely the garbage collector) sees it, notices that it is illegal, and halts the machine.

Subprimitives can also be used to alter the contents of nearly any location in memory, causing unpredictable results if the memory location is part of a sensitive system data structure. In this context even CAR, CDR, RPLACA, and RPLACD can be considered subprimitive functions. If they are given a locative instead of a list, they access or modify the addressed cell without regard to any object that may contain the cell.

The pointer-manipulating subprimitives are the most likely ones to cause problems (in general the ones beginning with %P-). These primitives can be very powerful when used properly, but can be very dangerous. There are strict conventions that dictate how these primitives should be used. Most of these conventions have to do with the garbage collector, and some newly-imposed restrictions stem from the temporal garbage collection algorithm (TGC). Some of these storage conventions are described here, along with the subprimitives themselves, and some are discussed in the sections on Internal Storage Formats and Garbage Collection.

NOTE

Extreme caution should be exercised when using any of the pointer-manipulating functions. At a minimum, before using them you should fully understand the notion of boxed versus unboxed storage, the role of primitive data types (both of these are covered in the Internal Storage Formats section), and the TQC-imposed restrictions on subprimitive use. If you have any questions regarding the correct use of the functions, please contact a Texas Instruments systems analyst.

NOTE

Unless indicated otherwise, all the subprimitives listed below are in the SYSTEM package.

15.3 STORAGE LAYOUT DEFINITIONS

The following special variables have values which define the most important attributes of the way Lisp data structures are laid out in storage. In addition to the variables documented here, there are many others that are more specialized. Variables whose names start with %% are byte specifiers; those beginning with % are numeric constants.

%%q-cdr-code Constant
The field of a boxed memory word that contains the cdr-code.

%%q-data-type Constant
The field of a boxed memory word that contains the data type code.

%%q-pointer Constant
The field of a boxed memory word that contains the object address or immediate data.

%%q-pointer-within-page Constant
The field of a boxed memory word that contains the part of the address that lies within a single page.

%%q-typed-pointer Constant
The concatenation of the %%Q-DATA-TYPE and %%Q-POINTER fields.

%%q-all-but-typed-pointer Constant
This is now synonymous with %%Q-CDR-CODE, and therefore obsolete.

%%q-all-but-pointer Constant
The concatenation of all fields of a memory word except for %%Q-POINTER.

%%q-all-but-cdr-code Constant
The concatenation of all fields of a memory word except for %%P-CDR-CODE.

%%q-high-half Constant

%%q-low-half Constant
The halves of a memory word. These fields are generally only useful for storing into unboxed memory locations.

cdr-normal Constant

cdr-next Constant

cdr-nil Constant

cdr-error Constant
The values of these four variables are the numeric values that go in the cdr-code field of a memory word.

15.4 DATA TYPES

q-data-type
The value of Q-DATA-TYPES is a list of all of the symbolic names for data types. These are the symbols whose print names begin with "DTP-". The values of these symbols are the internal numeric data type codes for the various internal data types. The section on Internal Storage Formats contains a list of all internal data types along with a description of each.

%data-type (x)
Returns the data type field of x, as a FIXNUM.

q-data-types(type-code)
Given the internal numeric data type code, returns the corresponding symbolic name. This "function" is actually an array.

data-type (arg)
DATA-TYPE returns a symbol that is the name for the internal data type of arg. The TYPE-OF function is a high-level primitive that is more useful in most cases; normal programs should always use type-of (or, when appropriate, TYPEP) rather than data type. Note that some types as seen by the user are not distinguished from each other at this level, and some user types may be represented by more than one internal type. For example, DTP-EXTENDED-NUMBER is the symbol that DATA-TYPE would return for either a double-float or a BIGNUM, even though those two types are quite different.

Some of these type codes occur in memory words but cannot be the type of an actual Lisp object. These include header types such as DTP-SYMBOL-HEADER, which identify the first word of a structure, and forwarding or "invisible" pointer types such as DTP-ONE-Q-FORWARD.

15.5 POINTER MANIPULATION

%pointer (x)
Returns the pointer field of x, as a fixnum. For most types, this is dangerous since the garbage collector can copy the object and change its address.

%make-pointer (data-type pointer)

%make-pointer-offset (data-type pointer offset)
These functions make up an object (pointer) created from the specified DATA-TYPE field and with pointer field POINTER (or POINTER plus OFFSET).

DATA-TYPE should be an internal numeric data type code; the value of one of the DTP- symbols. POINTER may be any object; its pointer field is used. In the case of %MAKE-POINTER-OFFSET, OFFSET may also be any object (just its pointer field is used) but is usually a FIXNUM. The types of the arguments are not checked; their pointer fields are simply added together. This is useful for constructing locative pointers into the middle of an object. However, remember that it is illegal to have a pointer to untyped data, such as the inside of a FEF or a numeric array.

The resulting object is returned, which means it is subjected to the Write Barrier so must be a valid Lisp object. Hence DATA-TYPE must not be a data type that is illegal "in the machine" (such as DTP-NULL or most of the invisible forwarding types).

%MAKE-POINTER and %MAKE-POINTER-OFFSET are extremely dangerous functions. They can be used to create objects with illegal data types or pointers to nonexistent memory (in which case the machine will crash immediately). More subtly, they can create objects with valid data types and pointer fields, but which do not point at valid structures (an array object pointing to a symbol header, for example, or into the middle of the unboxed portion of a FEF). Any of these can cause severe or fatal problems for the low-level system and for the garbage collector.

Note that it is always safe to create a FIXNUM data object with either of these two functions. It is creating pointer type objects that can be dangerous.

%pointerp (object)
T if object points to storage. For example, (%POINTERP "foo") is T because "foo" is an array which points to extended storage. (%POINTERP 5) is NIL since 5 is a FIXNUM.

%pointer-type-p (data-type)
T if the specified DATA-TYPE is one which points to storage. For example, (%POINTER-TYPE-P DTP-Fix) returns NIL.

%stack-frame-pointer
Returns a locative pointer to its caller's stack frame. This function is not defined in the interpreted Lisp environment: it only works in compiled code. Since it turns into a miscop instruction, the "caller's stack frame" really means "the frame for the FEF that executed the %stack-frame-pointer instruction."

15.6 POINTER ARITHMETIC

Addition, subtraction and comparison of object addresses should not be done with normal Lisp arithmetic functions since the largest positive Lisp FIXNUM fits in 24 bits while high virtual addresses can use the 25th bit. The Lisp functions thus will consider FIXNUMS with the 25th bit set as a negative number, whereas address arithmetic considers them large positive numbers. Furthermore, with Lisp functions like +, a BIGNUM is created if two 24-bit FIXNUM values added together overflow the 24-bit field. The pointer field of the resulting BIGNUM object is, of course, an object reference (probably pointing to the EXTRA-PDL area or into WORKING-STORAGE somewhere where the actual BIGNUM storage exists). Address arithmetic, on the other hand, requires that a 25-bit FIXNUM with the high bit set be returned when two such 24-bit values are added.

The following primitives should always be used to do pointer arithmetic. Bad pointer arithmetic is a frequent cause of bugs

in system code. However, it should be remembered that pointer arithmetic can itself be dangerous, depending on the context, even when the right functions are used. For example, if you use %POINTER-PLUS to generate an address inside of an object, you must make sure that the address thus generated cannot change (due to garbage collection) while you are using it. This can be done by protecting small sections of critical code with a WITHOUT-INTERRUPTS form or larger sections of code with a INHIBIT-GC-FLIPS form. Either will guarantee the object's address will not change inside its body by preventing a flip from occurring.

%pointer-difference (ptr1 ptr2)
Returns a FIXNUM which is ptr1 minus ptr2. No type checks are made. For the result to be meaningful, the two pointers must point into the same object, so that their difference cannot change as a result of garbage collection.

%pointer-plus (ptr1 ptr2)
Adds PTR1 and PTR2 together using address arithmetic; the result is always a FIXNUM suitable for use by the %P-routines (if, of course it represents a valid address). The data type field of PTR1 and PTR2 are ignored, but for this to be meaningful one is usually an object and the other is a FIXNUM meant as an offset.

%pointer= (ptr1 ptr2)
T if the pointer fields of PTR1 and PTR2 are the same. This is like EQ except that data types are ignored.

%pointer> (ptr1 ptr2)
T if PTR1 has a strictly higher virtual memory address than PTR2. Data type fields are ignored.

%pointer>= (ptr1 ptr2)
T if PTR1 has a higher virtual memory address than PTR2 or the same. Data type fields are ignored.

%pointer< (ptr1 ptr2)
T if PTR1 has a strictly lower virtual memory address than PTR2. Data type fields are ignored.

%pointer<= (ptr1 ptr2)
T if PTR1 has a lower virtual memory address than PTR2 or the same. Data type fields are ignored.

convert-to-unsigned (num)
Returns a number that corresponds to NUM (usually a FIXNUM) considered as a 25-bit unsigned quantity. Thus if FIXNUM is a positive FIXNUM, it is just returned. If it is a negative FIXNUM, a BIGNUM is returned.

convert-to-signed (num)
Returns a number that corresponds to NUM (usually a BIGNUM)

considered as a 25-bit unsigned quantity. This is the reverse of CONVERT-TO-UNSIGNED.

(CONVERT-TO-SIGNED (CONVERT-TO-UNSIGNED <num>)) = <num> always holds true if NUM is a FIXNUM.

%logdpb (value ppss word)
A FIXNUMs-only form of DPB. The low order SS bits of VALUE replace the PPSS field of WORD. Always returns a FIXNUM. Does not complain about loading/clobbering the sign bit. Data type fields of VALUE and WORD are ignored.

%logldb (ppss word)
A FIXNUMs-only form of LDB. Returns a FIXNUM obtained from the uninterpreted 32-bits of WORD. The result may be negative if the field width is 25. Signals the ARGTYP error if PPSS is not a FIXNUM or if it specifies a field greater than 25 bits wide.

15.7 FORWARDING

An invisible pointer or forwarding pointer is a kind of pointer that does not represent a Lisp object, but just resides in memory. There are several kinds of invisible pointers, and there are various rules about where they may or may not appear. These are detailed in the sections on Internal Storage Formats and Garbage Collection. The basic property of an invisible pointer is that if the Explorer system reads a word of memory and finds an invisible pointer there, instead of seeing the invisible pointer as the result of the read, it does a second read at the location addressed by the invisible pointer, and returns that as the result instead. Writing behaves in a similar fashion. When the Explorer system writes a word of memory it first checks to see if that word contains an invisible pointer; if so it goes to the location pointed to by the invisible pointer and tries to write there instead. Many subprimitives that read and write memory do not do this checking, hence violate normal rules about handling internal storage formats (they do this because they are used in low-level system routines that implement the storage formatting).

The simplest kind of invisible pointer has the data type code DTP-ONE-Q-FORWARD. It is used to forward a single word of memory to some place else. The invisible pointers with data types DTP-HEADER-FORWARD and DTP-BODY-FORWARD are used for moving whole Lisp objects (such as cons cells or arrays) somewhere else. The DTP-EXTERNAL-VALUE-CELL-POINTER is similar to DTP-ONE-Q-FORWARD; the difference is that it is not "invisible" to the operation of binding. If the (internal) value cell of a symbol contains a DTP-EXTERNAL-VALUE-CELL-POINTER that points to some other word (the external value cell), then SYMEVAL or SET operations on the

symbol consider the pointer to be invisible and use the external value cell. The operation of binding the symbol, however, saves away the DTP-EXTERNAL-VALUE-CELL-POINTER itself, and stores the new value into the internal value cell of the symbol. This is how closures are implemented.

The other forwarding pointer types DTP-GC-FORWARD and DTP-GC-YOUNG-POINTER are not the same kind of forwarding pointers. They can never be seen by any program other than the garbage collector.

structure-forward (old-object new-object)
This causes references to OLD-OBJECT to be forwarded to NEW-OBJECT. Does this by creating new storage, copying the contents of OLD-OBJECT into it, then storing invisible DTP-HEADER-FORWARDs/ DTP-BODY-FORWARDs into OLD-OBJECT. It returns OLD-OBJECT.

An example of the use of STRUCTURE-FORWARD is ADJUST-ARRAY. If the array is being made bigger and cannot be expanded in place, a new array is allocated, the contents are copied, and the old array is structure-forwarded to the new one. This forwarding ensures that pointers to the old array, or to cells within it, continue to work. When the garbage collector goes to copy the old array, it notices the forwarding and uses the new array as the copy; thus the overhead of forwarding disappears eventually if garbage collection is in use.

follow-structure-forwarding (object)
Normally just returns object, but if object has been structure-forwarded, returns the object at the end of the chain of forwardings instead. If object is not exactly an object, but a locative to a cell in the middle of an object, a locative to the corresponding cell in the latest copy of the object is returned.

forward-value-cell (from-symbol to-symbol)
This alters FROM-SYMBOL so that it always has the same value as TO-SYMBOL, by sharing its value cell. A DTP-ONE-Q-FORWARD invisible pointer is stored into FROM-SYMBOL'S value cell. Do not do this while FROM-SYMBOL's current dynamic binding is not global, as the microcode does not bother to check for that case and something bad will happen when FROM-SYMBOL'S binding is unbound. The microcode check is omitted to speed up binding and unbinding.

This is how synonymous variables (such as *TERMINAL-IO* and TERMINAL-IO) are created.

To forward one arbitrary cell to another (rather than specifically one value cell to another), given two locatives, do

```
(%P-STORE-DATA-TYPE-AND-POINTER
    from-location DTP-One-Q-Forward to-location)
```

follow-cell-forwarding (loc evcp-p)
LOC is a locative to a cell. Normally LOC is returned, but if the cell has been forwarded, this follows the chain of forwardings and returns a locative to the final cell. If the cell is part of a structure which has been forwarded, the chain of structure forwardings is followed, too. If EVCP-P is true, EVCPs are followed; if it is NIL they are not.

15.8 ANALYZING STRUCTURES

%find-structure-header (pointer)
Finds the structure into which POINTER points and returns an object reference to it. For example, given a locative into an ART-Q array, %FIND-STRUCTURE-HEADER would return the array. Works by searching backward in memory for a header, so it is illegal to give it a pointer into unboxed storage. It is a basic low-level function used by such things as the garbage collector. The data type of POINTER is ignored, but is generally a locative or a FIXNUM. Structure forwarding is not followed.

In structure space, the "containing structure" of a pointer is well-defined by the object's header. In list space, however, the containing structure is considered to be the contiguous, cdr-coded segment of list surrounding the location pointed to. Hence, it will return the length of an entire cdr-coded list segment if given a pointer anywhere into that segment; and will return 2 if given a pointer to either the CAR or the CDR of any simple CONS cell.

If a CONS of a cdr-coded list has been copied out by RPLACD, the contiguous list measured includes that CONS cell pair (so actually may be one greater than the contiguous segment).

find-structure-header (pointer)
Identical to %FIND-STRUCTURE-HEADER but first follows any structure-forwarding markers between what POINTER points to and the eventual actual object storage.

%find-structure-leader (pointer)
Identical to %FIND-STRUCTURE-HEADER except that if the structure is an array with a leader, this returns a locative pointer to the leader-header, rather than returning the

array-pointer itself. Thus the result of %FIND-STRUCTURE-LEADER is always the lowest address in the structure. This is the one used internally by the garbage collector. Structure forwarding is not followed.

find-structure-leader (pointer)
Identical to %FIND-STRUCTURE-LEADER but first follows any structure-forwarding markers between what POINTER points to and the eventual actual object storage.

%find-structure-header-safe (ptr)

%find-structure-leader-safe (ptr)
Just like %FIND-STRUCTURE-HEADER and %FIND-STRUCTURE-LEADER but are safe (and slow). Returns NIL if PTR isn't valid virtual memory; else returns the object (or locative to the array leader). These are safe because they parse memory forward from the region origin. They will FERROR if anything strange is found in the parsing. These even do intelligent things when given Oldspace and Copyspace addresses. See the documentation string for more details.

They used for debugging only. They are too slow for system code.

%structure-boxed-size (object)
Returns the number of "boxed Q's" in OBJECT. This is the number of words at the front of the structure which contain Lisp objects. Some structures, for example FEFs and numeric arrays, contain additional "unboxed Q's" following their boxed Q's. Note that the boxed size of a PDL (either regular or special) does not include Q's above the current top of the PDL. Those locations are technically boxed, since they are part of a Q-type array, but their contents are considered unboxed garbage and are not looked at by the garbage collector.

OBJECT's data type is not checked, but it better point to valid, boxed storage. Structure forwarding is not followed.

%STRUCTURE-BOXED-SIZE is used internally by the garbage collector to get a count of the number of words in a structure that need to be scavenged. This number is sometimes different from the real number of legitimate boxed words in an object. For example, it never counts forwarded body words as boxed (although forwarded leader and header words are always counted as boxed). This is because after scavenging the leader and the header of a forwarded structure, all the work is done. Also, will return 0 for any BIGNUM because the BIGNUM header word technically doesnt have to be scavenged since its pointer field just contains flag bits.

%structure-total-size (object)
Returns the total number of words occupied by the representation of OBJECT, including all boxed and unboxed words (and garbage past the top-of-stack pointers of PDLs). OBJECT's data type is not checked, but it better point to valid, boxed storage.

structure-boxed-size (object)
Same as %STRUCTURE-BOXED-SIZE but first follows any structure-forwarding markers between what POINTER points to and the eventual actual object storage.

structure-total-size (object)
Same as %STRUCTURE-TOTAL-SIZE but first follows any structure-forwarding markers between what POINTER points to and the eventual actual object storage.

%structure-size-safe (ptr &optional (include-leader t))
A safe way to find the real size of an object. Returns NIL if PTR isnt valid virtual memory. PTR should point at a structure header in structure space (but just FERRORs if not). Returns 4 values: total size in words, number of boxed words, space type of PTR (:NEW :COPY :OLD) and a flag which if T means it is a forwarded structure (RPLACD-forwarded in list space; structure forwarded in structure space).

Used only for debugging. Too slow for system code.

dump-memory (ptr &key (length 20) (base 8.))
Dumps raw memory contents formatted in a couple of interesting ways starting at address specified by PTR (which can be any object) for :LENGTH number of words in base :BASE. Check the documentation string for other args and meanings.

dump-objects (ptr &key num-objects)

dump-objects-in-region region &key (start-offset 0)
Start dumping out a brief representation of all objects starting either at address PTR (for DUMP-OBJECTS) or at the start of region REGION plus :START-OFFSET (for DUMP-OBJECTS-IN-REGION). Will just keep dumping until end of a region is reached or until :NUM-OBJECTS specified runs out. More keywords are available; check the documentation strings.

15.9 CREATING OBJECTS

%allocate-and-initialize (data-type header-type header second-word area size)
This is the subprimitive for creating most structure-type objects. AREA is the area in which it is to be created, as a FIXNUM or a symbol. SIZE is the number of words to be allocated. The value returned points to the first word allocated and has data type DATA-TYPE. The words allocated are initialized with interrupts disallowed so that storage conventions are preserved at all times. The first word, the header, is initialized to have HEADER-TYPE in its data-type field and HEADER in its pointer field. The second word is initialized to SECOND-WORD. The remaining words are initialized to NIL. The cdr-codes of all words except the last are set to CDR-NEXT; the cdr-code of the last word is set to CDR-NIL. Note that programs should generally not rely on the cdr-code field of non-CONS cells being in a known state.

%allocate-and-initialize-array (header data-length leader-length area size)
This is the subprimitive for creating arrays, called only by MAKE-ARRAY. It differs from %ALLOCATE-AND-INITIALIZE because arrays have a more complicated header structure.

%allocate-and-initialize-instance (header area size)
Allocates storage for an instance, sets header type to DTP-INSTANCE-HEADER and sets data type to DTP-INSTANCE. Fills allocated space with NIL and places HEADER in word 0.

The basic functions for creating list-type objects are CONS and MAKE-LIST; no special subprimitive is needed.

15.10 RETURNING STORAGE

return-storage (object &optional force-p)
With the advent of Temporal Garbage Collection (TGC), the RETURN-STORAGE facility becomes less useful as well as more difficult to implement without breaking the garbage collector. Hence RETURN-STORAGE now does nothing (and returns NIL) in the normal case. The normal case is when TGC is enabled (the value of %TGC-ENABLED is true, which is the default), or when the optional hidden FORCE-P argument to RETURN-STORAGE is NIL (the default).

When either the FORCE-P argument is true or TGC is disabled, this function attempts to return object in order to free storage. If it is a displaced array, it returns the displaced array itself, not the data that the array points to. RETURN-STORAGE does nothing if the object is not at the end of its region (i.e., if it was neither the most recently allocated non-list object in its region, nor the most recently allocated list in its region).

If you still have any references to object anywhere in the Lisp world after this function returns, the garbage collector can get a fatal error if it sees them. Since the form that calls this function must get the object from somewhere, it may not be clear how to call return-storage legally. One of the only ways to do it is as follows:

```
(DEFUN func ()
  (LET ((object (MAKE-ARRAY 100)))
    ...
    (RETURN-STORAGE (PROG1 object (SETQ object nil)))))
```

so that the variable object does not refer to the object when RETURN-STORAGE is called. Alternatively, you can free the object and get rid of all pointers to it while interrupts are turned off with WITHOUT-INTERRUPTS.

If RETURN-STORAGE is forced when TGC is enabled, and if the object being returned ever contained pointers to younger objects, those younger objects will not be able to be collected by the garbage collector until a full, promoting garbage collection is performed. For this reason, it is recommended that code force NILs into all boxed slots of a data structure being returned before using RETURN-STORAGE.

15.11 COPYING DATA

%BLT and %BLT-TYPED are subprimitives for copying blocks of data, word aligned, from one place in memory to another with little or no type checking. The acronym BLT is short for BLock Transfer.

%blt (from to count increment)

%blt-typed (from to count increment)
Copies COUNT words, separated by INCREMENT. The word at address FROM is moved to address TO; the word at address FROM+INCREMENT is moved to address TO+INCREMENT, and so on until COUNT words have been moved.

Only the pointer fields of FROM and TO are significant; they may be locatives or even FIXNUMs. If one of them must point

to the unboxed data in the middle of a structure, you must make it a FIXNUM, and you must do so with interrupts disabled, or else garbage collection could move the structure after you have already created the FIXNUM.

%BLT-TYPED assumes that each copied word contains a data type field and that every destination location is boxed memory. That is, %BLT-TYPED subjects its data to both the Read Barrier and the Write Barrier. Use %BLT-TYPED when copying FROM boxed storage TO boxed storage.

%BLT assumes that all source and destination locations are unboxed memory. That is, it subjects its data to neither the Read Barrier nor the Write Barrier. Use %BLT when copying FROM unboxed storage TO unboxed storage.

Copying FROM boxed storage TO unboxed storage must be handled specially. See the subsection on TGC Subprimitive Semantics.

It is actually safe to use either %BLT or %BLT-TYPED on data which is formatted with data types (boxed) but whose contents do not now point to storage and never have. This includes words whose contents are always FIXNUMs or short floats, and also words which contain array headers, array leader headers, or FEF headers. Whether or not such words go through the Read Barrier and Write Barrier makes no difference since there is no special processing for these types in either one.

15.12 SPECIAL MEMORY REFERENCING

This section describes the most dangerous storage management subprimitives. They are dangerous for several different reasons. First, since they read a virtual memory address specified by a pointer field, they can cause a crash if given an invalid virtual address (for example, one which used to contain an object but is now free because of garbage collection). Others are also dangerous because the allow the manipulation of the various fields of memory words, and often circumvent the normal Read Barrier/Write Barrier processing that normally occurs.

In addition to these dangers, which have always existed, there are some new problems use of these subprimitives can cause due to the TGC algorithm (described in the section on Garbage Collection). A later subsection explains these new restrictions. They have to do with which primitives must be used on boxed data and which are safe to use on unboxed data, so that part of the descriptions below should be paid special attention.

15.12.1 Subprimitives for Boxed Memory Referencing.

%p-pointerp (location)
T if the contents of the word at location points to storage. This is similar to (%POINTERP (CONTENTS location)), but the latter may get an error if LOCATION contains a forwarding pointer, a header type, or a void marker. In such cases, %P-POINTERP correctly tells you whether the header or forwarding pointer points to storage. Suitable for use only on boxed locations.

%p-pointerp-offset (location offset)
Similar to %P-POINTERP but operates on the word OFFSET words beyond LOCATION. Suitable for use only on boxed locations.

%p-contents-offset (base-pointer offset)
Returns the contents of the word OFFSET words beyond BASE-POINTER. This first checks the cell pointed to by BASE-POINTER for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds OFFSET to the resulting forwarded BASE-POINTER and returns the contents of that location. Suitable for use only on boxed locations.

There is no %P-CONTENTS, since CAR and CONTENTS perform that operation.

%p-contents-safe-p (location)
T if the contents of word LOCATION are a valid Lisp object, at least as far as data type is concerned. It is NIL if the word contains a header type, a forwarding pointer, or a void marker. If the value of this function is T, you will not get an error from (CONTENTS location). Suitable for use only on boxed locations.

%p-contents-as-locative (pointer)
Given a POINTER to a memory location containing an EVCP or One-Q-Forward, returns the contents of the location as a DTP-LOCATIVE. It changes the disallowed data type to DTP-LOCATIVE so that you can safely look at it and see what it points to. Suitable for use only on boxed locations.

%p-contents-as-locative-offset (base-pointer offset)
Extracts the contents of a word like %P-CONTENTS-OFFSET, but changes it into a locative like %P-CONTENTS-AS-LOCATIVE. This can be used, for example, to analyze the DTP-EXTERNAL-VALUE-CELL-POINTER pointers in a FEF, which are used by the compiled code to reference value cells and function cells of symbols. Suitable for use only on boxed locations.

%p-safe-contents-offset (location offset)
Returns the contents of the word offset words beyond location as accurately as possible without getting an error. Suitable for use only on boxed locations. Forwarding pointers are checked as in %P-CONTENTS-OFFSET.

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

%p-store-contents (pointer value)
Stores value into the data-type and pointer fields of the location addressed by pointer, and returns value. The cdr-code field of the location remains unchanged.

%p-store-contents-offset (value base-pointer offset)
Stores value in the location offset beyond words beyond base-pointer, then returns value. The cdr-code field remains unchanged. Forwarding pointers in the location at base-pointer are handled as they are in %p-contents-offset.

%p-pointer (pointer)
Extracts the pointer field of the contents of the location addressed by POINTER and returns it as a FIXNUM. Use only on boxed locations.

%p-data-type (pointer)
Extracts the data-type field of the contents of the location addressed by POINTER and returns it as a FIXNUM. Use only on boxed locations.

%p-cdr-code (pointer)
Extracts the cdr-code field of the contents of the location addressed by POINTER and returns it as a FIXNUM. Use only on boxed locations.

%p-store-pointer (pointer value)
Stores value in the pointer field of the location addressed by POINTER, and returns VALUE. Use only on boxed locations.

%p-store-data-type (pointer value)
Stores value in the data-type field of the location addressed by POINTER, and returns VALUE. Use only on boxed locations.

%p-cdr-code (pointer value)
Stores value in the cdr-code field of the location addressed by POINTER, and returns VALUE. Use only on boxed locations.

%p-store-tag-and-pointer (pointer miscfields pointerfield)
Stores MISCFIELDS and POINTERFIELD into the location addressed by POINTER. 25 bits are taken from POINTERFIELD to fill the pointer field of the location, and the low 7 bits of MISCFIELDS are used to fill both the data-type and cdr-code fields of the location. The low 5 bits of MISCFIELDS become the data-type, and the top two bits become the cdr-code. Use only on boxed locations.

This is a suitable primitive to use when both the data type and the pointer field of a memory word are to be changed since it does so atomically; that is, the resulting word is only subjected to the Write Barrier after both fields have been changed. Applying the Write Barrier when only one of the two has changed can cause problems.

%p-store-data-type-and-pointer (pointer data-type-to-store ptr-to-store)
Similar to %P-STORE-TAG-AND-POINTER except that the cdr code of the location being stored into is preserved. Stores data type DATA-TYPE-TO-STORE and pointer field PTR-TO-STORE into the corresponding fields at location POINTER, preserving the cdr code at POINTER. Use only on boxed locations.

This is a suitable primitive to use when both the data type and the pointer field of a memory word are to be changed since it does so atomically; that is, the resulting word is only subjected to the Write Barrier after both fields have been changed. Applying the Write Barrier when only one of the two has changed can cause problems.

15.12.2 Subprimitives for Unboxed Memory Referencing.

%p-ldb (byte-spec pointer)
Extracts a byte according to BYTE-SPEC from the contents of the location addressed by POINTER, in effect regarding the contents as a 32-bit number and using LDB. The result is always a FIXNUM. Use only on unboxed locations.

%p-ldb-offset (byte-spec base-pointer offset)
Extracts a byte according to BYTE-SPEC from the contents of the location OFFSET words beyond BASE-POINTER, after handling forwarding pointers. Use only on unboxed locations.

%p-dpb (value byte-spec pointer)
Stores VALUE, a FIXNUM, into the byte selected by BYTE-SPEC in the word addressed by POINTER. NIL is returned. Use only on unboxed locations.

%p-dpb-offset (value byte-spec base-pointer offset)
Stores value into the specified byte of the location OFFSET words beyond that addressed by BASE-POINTER, after first handling forwarding pointers. NIL is returned. Use only on unboxed locations.

%p-deposit-field (value byte-spec pointer)
Like %P-DPB, except that the selected byte is stored from the corresponding bits of VALUE rather than the right-aligned bits. See the note above %P-DPB for restrictions. Use only on unboxed locations.

%p-deposit-field-offset (value byte-spec base-pointer offset)
Like %P-DPB-OFFSET, except that the selected byte is stored from the corresponding bits of VALUE rather than the right-aligned bits. See the note above %P-DPB for restrictions. Use only on unboxed locations.

%p-mask-field (byte-spec pointer)
Like %P-LDB, except that the selected byte is returned in its original position within the word instead of right-aligned. Use only on unboxed locations.

%p-mask-field-offset (byte-spec base-pointer offset)
Like %P-LDB-OFFSET, except that the selected byte is returned in its original position within the word instead of right-aligned. Use only on unboxed locations.

15.13 TGC SUBPRIMITIVE SEMANTICS

With the advent of Temporal Garbage Collection, the semantics (and hence the proper use) of most of the special memory referencing subprimitives has changed. This change was required in order to handle properly the new "super-invisible" forwarding pointer used by TGC, the DTP-GC-Young-Pointer (GCYP). All uses of these primitives in Explorer kernel code had to be checked for compliance with these rules. Any user code which employs them must likewise be checked for compliance. This subsection gives a summary of the rules and the reasons behind them.

NOTE

Failure to follow the TGC-imposed rules for subprimitive usage can cause the system to crash even if the garbage collector is not active.

15.13.1 GC Young Pointer Usage.

With TGC, any time an attempt is made to store a (pointer to a) young object into an older object a trap is taken in the microcode, and instead a DTP-GC-Young-Pointer (GCYP) is stored in the old object. This young pointer points at an indirection cell in the new INDIRECTION-CELL-AREA area. The indirection cell actually contains the young object (or a pointer to it). Isolating these indirection cells allows scavenging of generations to be very fast, since we know that all pointers to a generation can be found using just a few indirection cell regions as the root of the scavenge space.

The GCYP is handled much like other forwarding pointer types, and probably most closely resembles the single-word-forwarding type DTP-ONE-Q-FORWARD. On any reference to a cell that can contain a Lisp object a check for a GCYP must be made. If one is found the reference is indirected to what the GCYP points to instead. This is the familiar transporting process.

As with most other forwarding pointer types, the GCYP is not allowed "in the machine"; that is, it is always "snapped out" right when it is read from memory. It is illegal to construct a pointer with type GCYP (with, for example %MAKE-POINTER), and it is further illegal to have any pointer or locative to an indirection cell except via GCYPs.

15.13.2 Dividing the Subprimitives.

In nearly all cases, the presence of GCYPs is completely invisible to the Lisp world. All Lisp objects are checked for the presence of forwarding pointers. The process of reading an object "into the machine" (pushing it on the PDL as a function argument, for example), performs this check automatically.

The introduction of GCYPs forces us to rethink our use of certain subprimitives, however. The %P- class of subprimitives require special care since they may be used to address arbitrary memory locations, not just ones known to be valid Lisp objects.

When using %P- subprimitives to reference memory locations, it has always been required that the user know if the referenced location is boxed or unboxed. Obviously it does not make sense to pick up the pointer field of an unboxed cell and attempt to use it as an address. However the user has formerly had a choice of functions to use in referencing word fields. This is no longer the case.

Consider the sets of statements below. Previously, if you wanted to read the pointer field of a memory word, you could use either of the following:

(%P-POINTER ptr) (%P-LDB %%Q-Pointer ptr)

You might know you are looking at a boxed word, and expect to get a valid address back. Or you might be looking inside unboxed data, but for some reason just "happen" to want to look at the field that corresponds to the pointer field of a boxed word. In either case, it was formerly allowable to use either call.

With the advent of TGC, however, we make a hard distinction between subprimitives which must be used to reference Lisp objects only, and those that should be used to reference unboxed data. If you are looking at or storing into Lisp object cells, there must be a check performed for the presence of a GCYP, since what you really want to reference is the corresponding field in the indirection cell the GCYP points to. If you know you are looking at unboxed data, however, you do not want to check for GCYP forwarding, since the cell referenced cannot be correctly interpreted as a type and pointer.

The tables below summarize the members of the two subprimitive groups. The bottom line is that the first group can only be meaningfully used on storage words that contain legitimate tag-and-pointer fields (boxed data). It is safe to use these functions if you know the word has a valid tag field (as long as you also follow the other storage rules of the particular function, of course). The second group should be used on unboxed memory words and may be used (for ease or efficiency) in certain very special cases on typed words (called "SAFE" boxed locations). The meaning of "SAFE" boxed locations is described later.

Table 15-1 GROUP 1. Use on BOXED cells ONLY

%P-POINTER
* %P-POINTER-OFFSET
%P-STORE-POINTER
* %P-STORE-POINTER-OFFSET
%P-DATA-TYPE
* %P-DATA-TYPE-OFFSET
%P-STORE-DATA-TYPE
* %P-STORE-DATA-TYPE-OFFSET
%P-CDR-CODE
* %P-CDR-CODE-OFFSET
%P-STORE-CDR-CODE
* %P-STORE-CDR-CODE-OFFSET
%P-STORE-TAG-AND-POINTER
* %P-STORE-DATA-TYPE-AND-POINTER
%P-STORE-CONTENTS
%P-STORE-CONTENTS-OFFSET
%BLT-TYPED

* Indicates a function new with TQC.

Table 15-2 GROUP 2. Use on UNBOXED or "SAFE" BOXED locations

%P-LDB
%P-LDB-OFFSET
%P-DPB
%P-DPB-OFFSET
%P-DEPOSIT-FIELD
%P-DEPOSIT-FIELD-OFFSET
%P-MASK-FIELD
%P-MASK-FIELD-OFFSET
%BLT
%BLT-TO-PHYSICAL
%BLT-FROM-PHYSICAL

Note that the members of Group 1 all either explicitly indicate a typed-data field (pointer, cdr-code, data-type), or otherwise imply by their names that they handle typed data (%BLT-TYPED, %P-STORE-CONTENTS). The members of Group 2 have more generic load

or store names.

To summarize the necessary changes:

1. If %P-LDB is being used to reference boxed data use %P-POINTER, %P-DATA-TYPE, or %P-CDR-CODE instead.

2. If %P-LDB-OFFSET is being used to reference a boxed word inside a structure, use the new %P-POINTER-OFFSET, %P-DATA-TYPE-OFFSET, or %P-CDR-CODE-OFFSET instead.

3. If %P-DPB or %P-DEPOSIT-FIELD is being used to reference boxed data use %P-STORE-POINTER, %P-STORE-DATA-TYPE, OR %P-STORE-CDR-CODE instead.

4. If %P-DPB-OFFSET is being used to reference a boxed word inside a structure, use %P-STORE-POINTER-OFFSET, %P-STORE-DATA-TYPE-OFFSET, or %P-STORE-CDR-CODE-OFFSET instead. These are also new.

5. Always use %BLT-TYPED to move data FROM completely boxed locations TO completely boxed locations. Use %BLT only to move data FROM unboxed storage TO unboxed storage. Special handling is required for the rare cases where the source and destination storage locations in a block transfer do not have the same typing characteristic (are not both boxed or both unboxed). This is discussed later.

15.13.3 Structure Forwarding Considerations.

Whenever an address offset into a structure is referenced, the -OFFSET version of a subprimitive should be used since these follow structure- forwarding. In the examples below, (1) and (2) are <u>not</u> equivalent, but (2) and (3) <u>are</u> equivalent as long as ADDRESS points to a header.

1. (%P-LDB (BYTE n n) (%POINTER-PLUS address offset))

2. (%P-LDB-OFFSET (BYTE n n) address offset)

3. (%P-LDB (BYTE n n)
   (%POINTER-PLUS
   (FOLLOW-STRUCTURE-FORWARDING address)
   offset))

Whether you use a Group 1 or Group 2 -OFFSET function further depends on whether you are referencing the boxed or unboxed part of a structure. For example, you would use %P-LDB-OFFSET when looking at FEF instructions, but use the new %P-DATA-TYPE-OFFSET

to check the data-type of a FEF-relative argument which is stored in the boxed section of the FEF.

15.13.4 Use of Group 2 Subprimitives on "SAFE" Boxed Storage.

It is actually all right to use Group 2 (unboxed) subprimitives to access boxed data under certain conditions. Most of these these conditions are listed below. There is actually quite a bit of system code that uses Group 2 subprimitives on boxed data. The array code, for example, uses %P-LDB to access array-header fields even though array headers are technically boxed. This is all right because an array header's pointer field does not contain a pointer and never can.

In short "SAFE" boxed storage is storage that does not now contain a pointer type Lisp object and never has contained one in the past. It can, however, contain immediate data types such as FIXNUMs now or in the past. Such storage locations are safe because they cannot contain GCYPs. In addition, certain areas will never contain GCYPs for simplicity and efficiency reasons.

Remember, if you cannot easily determine whether a boxed storage location is "SAFE", just change to a Group 1 subprimitive. This will always work, but may not be strictly necessary. SAFE, after all, is a relative term. It can be said, for example, that it is much SAFER to swim in shark-infested waters if you carry a harpoon.

Some examples of SAFE BOXED locations follow.

- The FIXED areas that appear before WORKING-STORAGE will never contain GCYPs.

- Regular PDLs, binding PDLs, and stack groups will never contain GCYPs. This is for efficiency because these structures are so volatile. Hence Group 2 subprimitives can generally be used on addresses in the areas shown below. The error handler takes advantage of this rule.

Area	Contents
LINEAR-PDL-AREA	Regular PDL of initial stack group
LINEAR-BIND-PDL-AREA	Bind PDL of initial stack group
PDL-AREA	All other regular PDLs
SG-AND-BIND-PDL-AREA	All other bind PDLs and all stack groups

NOTE

Locations beyond the current top-of-stack (push-pointer) in any PDL may contain unboxed data. These locations should not be referenced unless it is well-known will be found there (the error handler sometimes does this).

- Header types that have immediate pointer fields are safe. These are DTP-HEADER, DTP-ARRAY-HEADER, DTP-FEF-HEADER. The -OFFSET versions of group 2 subprimitive may be used on these. Symbol headers and instance headers are not safe, however, because they are actually pointer fields.

- Storage that has been freshly allocated by one of the following is safe because it has not yet been stored into at all:

%ALLOCATE-AND-INITIALIZE
%ALLOCATE-AND-INITIALIZE-ARRAY
%ALLOCATE-AND-INITIALIZE-INSTANCE
MAKE-LIST
%MAKE-STACK-LIST

- Boxed storage that has always always (and you can guarantee this) contained only immediate types is safe. Immediate types include DTP-FIX, DTP-SHORT-FLOAT, DTP-CHARACTER. By special dispensation the symbols NIL and T are included in this group. They are not immediate types but a GCYP can never be created which points to them because by definition no object is older than T or NIL.

Note that it is not sufficient to know that a location currently contains an immediate type, since if it has ever contained a pointer-type Lisp object, a GCYP may be there. Such a GCYP will not get snapped out until a collection of the old object's generation occurs. For example, just because a symbol value cell now holds a FIXNUM does not guarantee that no GCYP exists there. If a young BIGNUM were stored there previously, and it has since been overwritten by a FIXNUM, there will still be a GCYP there pointing to an indirection cell which contains the FIXNUM.

15.13.5 Special Cases of %BLT-TYPED and %BLT.

With TGC, %BLT-TYPED can only safely be used to transfer data FROM completely boxed storage TO storage that is completely boxed

before the transfer. The requirement that the destination be boxed is new with TGC, since an attempt will be made to follow GCYPs at the destination prior to the write. To transfer typed data TO a location that has formerly contained untyped data, the destination must be "faked" to look typed before %BLT-TYPED is used. An example:

```
(LET ((typed-array (MAKE-ARRAY 5.))
      (untyped-array
         (MAKE-ARRAY 5.
                     :element-type '(unsigned-byte 32.)))
     .... other code ....
     ;; Transfer contents of TYPED-ARRAY to UNTYPED-ARRAY
     ;; First make inside of UNTYPED-ARRAY look boxed.
     (%P-DPB-OFFSET dtp-fix %%G-DATA-TYPE untyped-array 1)
     (%P-DPB-OFFSET 0       %%G-POINTER   untyped-array 1)
     (%BLT
       ;; FROM: The 1st data word where we've
       ;; just put a fixnum 0...
       (%POINTER-PLUS
          (FOLLOW-STRUCTURE-FORWARDING untyped-array) 1)
       ;; TO: The 2nd data word ...
       (%POINTER-PLUS
          (FOLLOW-STRUCTURE-FORWARDING untyped-array) 2)
       ;; Then from the 2nd to the 3rd, and so forth.
       (1- (ARRAY-LENGTH untyped-array))
       1)
     ;; Now can do %BLT-TYPED
     (%BLT-TYPED
       (%POINTER-PLUS
         (FOLLOW-STRUCTURE-FORWARDING typed-array) 1)
       (%POINTER-PLUS
         (FOLLOW-STRUCTURE-FORWARDING untyped-array) 1)
       (ARRAY-LENGTH typed-array)
       1))
```

%BLT should only be used to transfer FROM completely unboxed storage TO completely unboxed storage. If the destination is boxed and you want to replace it with unboxed data, the destination should first be "faked" to be SAFE BOXED before the %BLT. This ensures that all objects pointed to by the current boxed storage will be made garbage. An example:

```
(LET ((typed-array (MAKE-ARRAY 5.))
      (untyped-array
         (MAKE-ARRAY 5.
                     :element-type '(unsigned-byte 32.)))
     .... other code ....
     ;; Transfer contents of UNTYPED-ARRAY to TYPED-ARRAY
     ;; First zap inside of TYPED-ARRAY with fixnum zeros
     ;; so the garbage collector will know all the old
     ;; stuff is trashed.
```

```
(%P-STORE-CONTENTS-OFFSET 0 typed-array 1)
(%BLT-TYPED
  ;; FROM: The 1st data word where we've
  ;; just put a fixnum 0...
  (%POINTER-PLUS
     (FOLLOW-STRUCTURE-FORWARDING typed-array) 1)
  ;; TO: The 2nd data word ...
  (%POINTER-PLUS
     (FOLLOW-STRUCTURE-FORWARDING typed-array) 2)
  ;; Then from the 2nd to the 3rd, and so forth.
  (1- (ARRAY-LENGTH typed-array))
  1)
;; Now can do %BLT.
(%BLT
   (%POINTER-PLUS
     (FOLLOW-STRUCTURE-FORWARDING untyped-array) 1)
   (%POINTER-PLUS
     (FOLLOW-STRUCTURE-FORWARDING typed-array) 1)
   (ARRAY-LENGTH untyped-array)
   1))
```

SECTION 16

Other Subprimitives, Variables, and Counters

16.1 INTRODUCTION

NOTE

Unless indicated otherwise, all the subprimitive and variable names listed in this section are in the SYSTEM package.

This section covers the Explorer subprimitives that are non-storage-related. They, like other subprimitives should be used with caution, since they may ruin the Lisp environment if used incorrectly.

NOTE

Because they are used at the lowest level of the Explorer virtual machine implementation, the subprimitives and variables described here may change in the way they function or even be removed entirely from the system without notice. Most programs should generally not refer to them directly.

16.2 ARRAY SUBPRIMITIVES

The subprimitives described below are special-purpose array manipulating functions.

%string-equal (string1 start1 string2 start2 count)
: %STRING-EQUAL is the microcode primitive used by STRING-EQUAL. It returns T if the COUNT characters of STRING1 starting at START1 are char-equal to the count characters of STRING2 starting at START2, or NIL if the characters are not equal or if COUNT runs off the length of either array.

Instead of a FIXNUM, COUNT may also be NIL. In this case, %STRING-EQUAL compares the substring from START1 to (STRING-LENGTH <string1>) against the substring from START2 to (STRING-LENGTH <string2>). If the lengths of these substrings differ, then they are not equal and NIL is returned.

Note that STRING1 and STRING2 must really be strings; the usual coercion of symbols and FIXNUMs to strings is not performed. This function is documented because certain programs which require high efficiency and are willing to pay the price of less generality may want to use %STRING-EQUAL in place of STRING-EQUAL.

Examples:

To compare the two strings foo and bar:
(%STRING-EQUAL foo 0 bar 0 nil)
To see if the string foo starts with the characters "bar":
(%STRING-EQUAL foo 0 "bar" 0 3)

alphabetic-case-affects-string-comparison Variable
If this variable is T, the function %STRING-EQUAL and %STRING-SEARCH-CHAR consider case (and font) significant in comparing characters. Normally this variable is NIL and those primitives ignore differences of case.

This variable may be bound by user programs around calls to %STRING-EQUAL and %STRING-SEARCH-CHAR, but do not set it globally, for that may cause the higher level string comparisons not to function as desired.

%string-search-char (char string from to)
%STRING-SEARCH-CHAR is the microcode primitive called by STRING-SEARCH-CHAR and other functions. STRING must be an array and CHAR, FROM and TO must be FIXNUMs. The arguments are all required. Case-sensitivity is controlled by the value of the variable ALPHABETIC-CASE-AFFECTS-STRING-COMPARISON rather than by an argument. Except for these differences, %STRING-SEARCH-CHAR is the same as STRING-SEARCH-CHAR. This function is documented for the benefit of those who require maximum possible efficiency in string searching.

ar-1 (array i)

ar-2 (array i j)

ar-3 (array i j k)

as-1 (array i)

as-2 (array i j)

as-3 (array i j k)

ap-1 (array i)

ap-2 (array i j)

ap-3 (array i j k)

These are obsolete versions of AREF, ASET and ALOC that only work for one-, two-, or three-dimensional arrays, respectively.

The compiler turns AREF into AR-1, AR-2, etc. according to the number of subscripts specified, turns ASET into AS-1, AS-2, etc., and turns ALOC into AP-1, AP-2, etc. For arrays with more than three dimensions the compiler uses the slightly less efficient form since the special routines only exist for one, two and three dimensions. There is no reason for any program to call AR-1, AS-1, AR-s, etc. explicitly; they are documented because there used to be such a reason, and many old programs use these functions. New programs should use AREF, ASET, and ALOC.

common-lisp-ar-1 (array i)

common-lisp-ar-2 (array i j)

common-lisp-ar-3 (array i j k)

common-lisp-aref (array &rest subscripts)

The first three of these functions are identical to AR-1, AR-2 and AR-3 except that they return a character object rather than an integer when ARRAY is a string. COMMON-LISP-AREF is the general array referencing subprimitive for Common Lisp. It also returns a character object rather than an integer when ARRAY is a string.

array-types Constant

The value of this constant is a list of all of the array type symbols such as ART-Q, ART-4B, ART-STRING and so on. The values of these symbols are internal array type code numbers for the corresponding type.

array-types (array-type-code)

Given an internal numeric array-type code, returns the symbolic name of that type.

array-elements-per-q Constant

This is an association list which associates each array type symbol with the number of array elements stored in one word, for an array of that type. If the value is negative, it is

instead the number of words per array element, for arrays whose elements are more than one word long.

array-elements-per-q (array-type-code)
Given the internal array-type code number, returns the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-bits-per-element Constant
The value of this constant is an association list which associates each array type symbol with the number of bits of unsigned number it can hold, or NIL if it can hold Lisp objects. This can be used to tell whether an array can hold Lisp objects.

array-bits-per-element (array-type-code)
Given the internal array-type code numbers, returns the number of bits per cell for unsigned numeric arrays, or NIL for a type of array that can contain Lisp objects.

array-element-size (array)
Given an array, returns the number of bits that fit in an element of that array. For arrays that can hold general Lisp objects, the result is 25 (decimal), based on the assumption that you will be storing FIXNUMs in the array.

16.3 STACK LIST SUBPRIMITIVES

When you are creating a list that will not be needed any more once the function that creates it is finished, it is possible to create the list on the stack instead of by consing it. This avoids any permanent storage allocation, as the space is reclaimed as part of exiting the function. These lists are called temporary lists or stack lists. You can create them explicitly using the special forms WITH-STACK-LIST and WITH-STACK-LIST*. &REST arguments also sometimes create stack lists. See the Explorer Lisp Reference manual for more information on these forms and there usage. Note that stack lists are automatically copied out of the stack if you store a pointer to them into normal virtual memory outside the stack.

16.4 FUNCTION-CALLING SUBPRIMITIVES

These subprimitives can be used to call a function with the number of arguments variable at run time. Since these subprimitives act as a signal for the compiler to initiate certain operations on the stack, they are only meaningful in compiled code. They are not callable from the interpreted Lisp environment.

NOTE

> The improper use of these functions can irreparably damage the state of the runtime stack (PDL). They should be used only with extreme caution. The preferred higher-level primitive is APPLY.

%push (value)
: Pushes VALUE onto the stack. Use this to push each argument.

%call (function number-of-args &key lexpr self-mapping-table)
: Call FUNCTION, passing arguments that have already been pushed on the stack with %PUSH. This is a sub-primitive that only works in compiled code.

%pop
: Pops the top value off of the stack and returns it as its value.

%assure-pdl-room (n-words)
: Call this before doing a sequence of %PUSH's that will add N-WORDS to the current frame. This subprimitive checks that the frame will not exceed the maximum legal frame size, which is 255 words including all overhead. This limit is dictated by the way stack frames are linked together. If the frame is going to exceed the legal limit, %ASSURE-PDL-ROOM signals an error.

16.5 SPECIAL-BINDING SUBPRIMITIVE

%bind (locative value)

bind (locative value)
Binds the cell pointed to by LOCATIVE to VALUE, in the caller's environment. This function is not defined in the interpreted Lisp environment; it only works from compiled code. Since it turns into an instruction, the "caller's environment" really means "the binding block for the compiled function that executed the %BIND instruction". The preferred higher-level primitives that turn into this are LET, LET-IF, and PROGV.

The binding is in effect for the scope of the innermost binding construct, such as prog or let, even if that construct binds no variables itself.

%BIND is the preferred name; BIND is an older name which will eventually be eliminated.

16.6 CLOSURE SUBPRIMITIVES

These functions are used to implement dynamic closures on the Explorer system. They deal with the distinction between internal and external value cells and control over how these different kinds of value cells interact.

%using-binding-instances (instance-list)
This function is the primitive operation that invocation of closures could use. It takes a list, and for each pair of elements in the list it "adds" a binding to the current stack frame, in the same manner that the %BIND function does. These bindings remain in effect until the frame returns or is unwound.

%USING-BINDING-INSTANCES checks for redundant bindings and ignores them. (A binding is redundant if the symbol is already bound to the desired external value cell.) This check avoids excessive growth of the special PDL in some cases and is also made by the microcode which invokes closures, entities, and instances.

Given a closure, closure-bindings extract its list of binding instances, which you can then pass to %USING-BINDING-INSTANCES.

%external-value-cell (symbol)
Returns a locative to whatever the value cell of SYMBOL points to. If SYMBOL is bound by a closure, this will be a locative to the external value cell. Does not check that the internal value cell contains an external value cell pointer.

16.7 LOCKING SUBPRIMITIVE

%store-conditional (pointer old new)
This is the basic locking primitive. POINTER is a locative to a cell which is read and written without interrupts. If the contents of the cell is EQ to OLD, then it is replaced by NEW and T is returned. Otherwise, NIL is returned and the contents of the cell are not changed. Normally programs should use the store-conditional function instead, since it includes type checking.

16.8 EXPLORER I/O DEVICE SUBPRIMITIVES

%nubus-read (slot byte-address)
Returns the contents of a word read from the NuBus. Addresses on the NuBus are divided into an 8-bit slot number which identifies which physical board is being referenced and a 24-bit address within slot. The address is measured in bytes and therefore should be a multiple of 4. Which addresses are valid depends on the type of board plugged into the specified slot. If, for example, the board is a 2 Megabyte main memory board, then the valid address range from 0 to 4 * (2MB - 1). (Of course, main memory is normally accessed through the virtual memory system.)

%nubus-write (slot byte-address word)
Writes WORD into a word of the NuBus, whose address is specified by SLOT and BYTE-ADDRESS as described above.

16.9 SUBPRIMITIVES TO SHUT DOWN THE LISP ENVIRONMENT

%halt
Stops the machine.

%crash (code object paws-up-p)
Halts the machine after writing a crash record that includes CODE and OBJECT (each stored as one word) which can be

reported by the crash analyzer. If PAWS-UP-P is true, the monitor screen will invert its video characteristic.

%disk-restore (high-16-bits low-16-bits)
Loads virtual memory from the partition named by the concatenation of the two 16-bit arguments, and starts executing it. The name 0 refers to the default load (the one the machine loads when it is started up). This is the primitive used by the DISK-RESTORE function.

16.10 VIRTUAL MEMORY SYSTEM SUBPRIMITIVES

A number of virtual memory system subprimitives are detailed in the last portion of the Paging and Disk Management section.

16.11 GARBAGE COLLECTION SUBPRIMITIVES

A number of subprimitives used by the garbage collector are detailed in the last portion of the section on Garbage Collection.

16.12 MICROCODE VARIABLES

The following variables' values actually reside in the processor's scratchpad memory register blocks. They are forwarded there by DTP-ONE-Q-FORWARD invisible pointers in the value cells of the symbols which name them. These variables are used by the microcode and by the low-level Lisp system to implement the virtual machine. Thus, they are highly internal, and their full meanings are not necessarily documented here. Changing their values can have unpredictable and undesirable results.

Although this variables can be set from Lisp code, since they are actually aliases of processor registers their values are reset to certain default values at boot time. Even a warm boot will alter some of them.

16.12.1 M-Memory Variables. These are the M-Memory variables. A list of all of these variables can be found in the value of the variable M-MEMORY-LOCATION-NAMES. This variable will always be up-to-date even if this documentation is not. Current M-Memory variable contents can be viewed with the function DUMP-M-MEMORY-Q-STORAGE or the function M-MEMORY.

%mode-flags Variable

The MODE-FLAGS currently in effect. Some flags are used as state flags by the transporter and the paging system. Others are general modes honored on a per-stack-group basis, such as the meter-enable field. Byte specifiers for each of the fields of this variable can be found in M-FLAGS-FIELDS.

%sequence-break-source-enable Variable
Enables different sources of sequence breaks. See SB-ON.

%meter-micro-enables Variable
Flags for enabling the different metering events. Byte descriptors can be found in the METER-ENABLES list.

self Variable
The current value of the SELF instance. It is an M-Memory register for efficiency.

self-mapping-table Variable
The current SELF-MAPPING-TABLE array used for instance variable references. It is an M-Memory register for efficiency.

16.12.2 A-Memory Variables. These are the A-Memory variables. A list of all of these variables can be found in the value of the variable M-MEMORY-LOCATION-NAMES. This variable will always be up-to-date even if this documentation is not. Current M-Memory variable contents can be viewed with the function DUMP-A-MEMORY-G-STORAGE or the function A-MEMORY.

%microcode-version-number Variable
This is the version number of the currently-loaded microcode, obtained from the version number stored in the microcode partition loaded.

processor-type-code Variable
Code for the current processor's type. Currently always 3.

microcode-type-code Variable
Code number for the currently running microcode. For legal values and the microcode types they represent, see the variable *MICROCODE-NAME-ALIST*.

microcode-debug-flags Variable
Microcode flags used for determining action to take on trap, sequence break, and so forth. See the function SET-MICROCODE-DEBUG-FLAGS for information on how to set them from Lisp. The list MICROCODE-DEBUG-FLAGS-BITS contains byte specifiers for all the bits.

amem-evcp-vector Variable
A-Memory address of location where A-Memory dynamic variable closure bindings may be stored.

%number-of-micro-entries Variable
The actual number of words used in the MICRO-CODE-ENTRY-AREA and the MICRO-CODE-ENTRY-DEBUG-INFO-AREA.

%counter-block-a-mem-address Variable
The offset in A-Memory of the start of the Microcode Counter block. See the following discussion of Microcode Counters.

%mar-low Variable
When the MAR is enabled, contains the lowest virtual address being monitored.

%mar-high Variable
When the MAR is enabled, contains the highest virtual address being monitored.

%method-search-pointer Variable

%method-subroutine-pointer Variable
Used by the flavor-support microcode.

%current-stack-group-state Variable
The sg-state of the currently-running stack group.

%current-stack-group-calling-args-pointer Variable
The argument list of the currently-running stack group.

%trap-micro-pc Variable
The microcode address of the most recent error trap.

%resumed-trap-ap-level Variable

%resumed-regular-pdl-pointer Variable

%resumed-special-pdl-pointer Variable

%resumed-saved-m-flags Variable
Used by the error handler.

%initial-fef Variable
The function that is called when the machine starts up. Normally this is the definition of LISP-TOP-LEVEL.

%initial-stack-group Variable
The stack group in which the machine starts up. Normally this is the initial Lisp Listener window's process's stack group.

%error-handler-stack-group Constant
The stack group that receives control when a microcode-detected error occurs. It is THE error handler stack group; that is when invoked, it cleans up, signals the appropriate condition, then assigns a second-level error handler stack

group which actually runs the debugger on the erring stack group.

%scheduler-stack-group Constant
The stack group that receives control when a sequence break occurs.

inhibit-scheduling-flag Variable
When true, all sequence breaks are disabled. This is used by the WITHOUT-INTERRUPTS macro.

%inhibit-read-only Variable
If true, you can write into read-only areas. This is used by FASLOAD and the garbage collector.

inhibit-scavenging-flag Variable
If true, the scavenger is turned off. The scavenger is the the quasi-asynchronous portion of the garbage collector which is invoked after consing operations if any generation is in a flipped state (undergoing collection).

default-cons-area Variable
The area number the default area in which to create objects. The default is WORKING-STORAGE-AREA.

background-cons-area Variable
The area number the default area in which to create objects that must not be created in a temporary area (hence this must never be the area-number of a temporary area). Used by the Extra-PDL dumper as the area for copying out numbers. The default is WORKING-STORAGE-AREA.

number-cons-area Variable
The area number of the area where BIGNUMs, ratios, full-size floats and complexnums are consed. Normally this variable contains the area number of the EXTRA-PDL-AREA. This enables number consing, the low-overhead garbage collection of extended numbers. To disable number consing, set this variable to the number of another area.

%region-cons-alarm Variable
Increments whenever a new region is allocated.

%page-cons-alarm Variable
Increments whenever a new page is allocated.

%gc-flip-ready Variable
NIL while a garbage collection is in progress. T when all scavenging is done (that is, when collection is complete and there are no more pointers to Oldspace).

%scavenger-ws-enable Variable
When a generation collection begins scavenging is

temporarily disabled to allow the mutator to move objects dynamically. When this variable is true the scavenger is on hold. When NIL, scavenging can occur (when INHIBIT-SCAVENGING FLAG is also NIL).

%gc-switches Variable
Currently unused.

%gc-generation-number Variable
A FIXNUM which is incremented whenever the garbage collector flips, converting one or more regions from Newspace to Oldspace. If this number has changed, the address of an object may have changed. Comparing this number with a hash table's internal GC generation number is used to cause EQ hash tables to rehash after a GC. The value cell is actually forwarded to a slot in the System Communication Area so that the changes to its value can live across a DISK-SAVE.

load-unit Variable
The physical disk unit from which Lisp was loaded.

%disk-switches Variable
Controls various paging system parameters. Documented in the paragraph on paging subprimitives.

address-of-page-device-table Variable
Controls various paging system parameters. Documented in the paragraph on paging subprimitives.

number-of-page-devices Variable
The current number of logical paging devices in the system. Includes all PAGE bands and the LOD band.

%free-cluster-count Variable
A FIXNUM which holds the current number of PAGE band clusters available for allocation.

%disk-run-light Constant
A FIXNUM. This is the virtual address of the TV buffer location of the run-light which lights up when the disk is active. This plus 2 is the address of the run-light for the processor. This minus 2 is the address of the run-light for the scavenger.

%disk-blocks-per-track Variable

%disk-blocks-per-cylinder Variable
Configuration of the disk.

%clipping-rectangle-left-edge Variable

%clipping-rectangle-right-edge Variable

%clipping-rectangle-top-edge Variable

%clipping-rectangle-bottom-edge Variable

%current-sheet Variable

%currently-prepared-sheet Variable
Used for communication between the window system and the microcoded graphics primitives.

%mouse-cursor-state Variable

%mouse-x Variable

%mouse-y Variable

%mouse-cursor-x-offset Variable

%mouse-cursor-y-offset Variable

%mouse-cursor-width Variable

%mouse-cursor-height Variable

%mouse-x-speed Variable

%mouse-y-speed Variable

%mouse-buttons-buffer-in-index Variable

%mouse-buttons-buffer-out-index Variable

%mouse-wakeup Variable

%mouse-h3 Variable
Used by the mouse tracking microcode.

%meter-global-enable Variable
T if the metering system is turned on for all stack-groups.

%meter-buffer-pointer Variable
A I/O buffer (RGB data buffer) used by the metering system.

%meter-disk-address Variable
Where the metering system writes its next block of results on the disk.

%meter-disk-count Variable
The number of disk blocks remaining for recording of metering information.

meter-unit Variable
The disk unit being used for metering

lexical-environment Variable
This is the static chain used in the implementation of lexical scoping of variable bindings in compiled code.

%mc-code-exit-vector Variable
Unused.

%inhibit-stack-list-copy Variable
Unused.

alphabetic-case-affects-string-comparison Variable
If this variable is T, the function %string-equal and %string-search-char consider case (and font) significant in comparing characters. Normally this variable is NIL and those primitives ignore differences of case.

zunderflow Variable
When this variable is true floating point underflow or divide-by-zero has occurred.

array-index-order Variable
When true (the default) arrays are stored in row-major order. This should not be changed, since no other ordering is supported by some of the array primitives.

ar-1-array-pointer-1 Variable

ar-1-array-pointer-2 Variable
Used by the array microcode as a cache of information on the last array decoded.

16.13 MICROCODE COUNTERS

The Microcode Counters (also called Meters are locations in the processor's scratchpad A-Memory register block which contain 32-bit numbers (they have nothing to do with the Lisp metering system per se).

Most of the meters are used to count events of various sorts. Others exist simply so that the value of certain microcode registers can be read and written from Lisp. They are accessible through the functions READ-METER and WRITE-METER. Their current values are displayed by the PEEK COUNTERS display and by invoking the function DUMP-A-MEMORY-COUNTERS or A-COUNTERS.

All the counters are described below. They are grouped by general use, although they do not appear in that order in the PEEK display

16.13.1 Accessing Counters.

read-meter (name)
Returns the contents of the microcode meter named NAME. NAME name must be one of the symbols in the counter descriptions below. Warning: since the meters can be 32-bit values, either a FIXNUM or a BIGNUM may be returned.

write-meter (name value)
Writes value, a FIXNUM or a BIGNUM, into the microcode meter named name. Name must be one of the symbols listed below.

a-memory-counter-block-names Constant
A list of all the symbols listed below. The value of each symbol is the counter's offset from the start of the counter block.

16.13.2 Page Exception Handling Counters.

%count-first-level-map-reloads Meter
The number of times the second-level map handling had to take place in order to set up a first level map.

%count-second-level-map-reloads Meter
The number of times the second-level virtual-memory map was invalid and had to be reloaded from the Page Hash Table.

%count-meta-bits-map-reloads Meter
The number of times the memory maps have been set up to contain only "meta bits" for the garbage collector without physical address mapping information.

%count-pdl-buffer-read-faults Meter
The number of read references to the PDL buffer that were virtual memory references that trapped.

%count-pdl-buffer-write-faults Meter
The number of write references to the PDL buffer that were virtual memory references that trapped.

%count-pdl-buffer-memory-faults Meter
The number of virtual memory references that trapped in case they might have been in the PDL buffer, but turned out to be real virtual memory references after all (and therefore were needlessly slowed down).

16.13.3 Page Fault Handling Counters.

%count-disk-page-reads Meter
The total number of disk page read operations performed. This is equal to the total number of disk pages read if

prepaging is disabled. If prepaging is enabled, the total number of disk pages read is this number plus the value of the %COUNT-PRE-PAGES-READ counter.

This number can also be considered a count of the number of times the page-fault logic was invoked because of a hard fault on a demanded page.

%count-fresh-pages Meter
The number of fresh (newly-consed) pages created in core. Since a fresh page request is considered a page fault (because they cause another page to be evicted), this number plus the %COUNT-DISK-PAGE-READS is a count of the total number of times the hard page-fault routine has been called.

%count-disk-page-writes Meter
The total number of pages written to the disk.

%count-disk-page-write-operations Meter
The total number of paging write operations. This is usually smaller than the %COUNT-DISK-PAGE-WRITES value because every attempt is made to swap out more than one page at a time.

%count-disk-page-write-appends Meter
The number of pages appended in multi-swapout operations. This number plus %COUNT-DISK-PAGE-WRITE-OPERATIONS should add up to %COUNT-DISK-PAGE-WRITES.

%count-pre-pages-read Meter
The number of page was read in due to pre-paging.

%count-pre-pages-used Meter
The number of read pre-pages that have actually been used. The ratio of this number to %COUNT-PRE-PAGES-READ gives a measure of pre-paging efficiency.

%disk-wait-time Meter
The total time spent waiting for the disk, in microseconds. Since swapouts are overlapped with processing and with swapins whenever possible, this number cannot be used in conjunction with the total number of disk operations in order to measure the disk hardware's processing time per page. However, it could be used to measure the paging system's average disk wait time per read operation, for example, if divided by the %COUNT-PAGE-READS number.

%total-page-fault-time Meter
Contains the total amount of time spent in page fault processing (when tracking this time is enabled by the %DISK-SWITCHES flag). This time includes all the time spent waiting on the disk (%DISK-WAIT-TIME) plus any microcode processing time not done during disk-I/O. Subtract %DISK-

WAIT-TIME from this number to get the amount of time take by the page fault processing routine. This does NOT include time spent handling page exceptions which did not turn out to be hard faults. Such page exception processing time is very small in any case, and can be considered part of the cost of computation.

%count-disk-page-write-waits Meter
The number of times the page write routine had to wait for a page to finish being written out before being able to start its I/O.

%count-disk-page-write-busys Meter
The total number of times any paging routine had to wait for a page to finish being written out in order to do something else with the disk.

%aborted-swapouts Meter
The number of pages referenced while a swapout to disk operation for them was still in progress.

%count-swapout-page-count-reached Meter
The number of multipage swapouts stopped because of the DISK-SWITCHES limit.

%swapout-sized-by-rqb-or-page-count Meter
The number of page writes limited by the size of the swapout RQB scatter list.

%count-disk-page-read-resubmissions Meter
The number of read operations restarted due to disk error. Since page read resubmissions are currently not supported, this number should always be 0.

%count-sb-from-swapper Meter
Unused.

%swapout-sequence-breaks-in-progress Meter
Unused.

16.13.4 Page Replacement Algorithm Counters.

%count-clean-page-requests Meter
The number of times FINDCORE (the page replacement routine) has been requested to return a clean memory page rather than just the least recently used page.

%count-clean-page-request-failed Meter
The number of times FINDCORE could not find a clean page.

%count-findcore-clear-pages Meter
The number of already flushed pages found by FINDCORE.

%count-findcore-steps Meter
The number of pages inspected by the page replacement algorithm.

%count-findcore-emergencies Meter
The number of times no evictable page was found and extra aging had to be done.

%least-used-page Meter
The PHT byte index of the least recently used physical memory page.

%most-recently-referenced-page Meter
The PHT byte index page frame number of the most recently referenced physical memory page.

16.13.5 PHT and PPD Information Counters.

%page-hash-table-address Meter
The starting physical address of the PHT.

%physical-page-data-addess Meter
The starting physical address of the PPD.

%pht-index-limit Meter
Maximum byte offset in the PHT

%physical-page-data-end Meter
Maximum byte offset in the PPD.

%pht-index-size Meter
Number of bits of the virtual page number to use in computing an address' PHT hash value.

%pht-search-depth Meter
Longest collision bucket so far in the Page Hash Table.

%page-table-search-count Meter
Unused.

16.13.6 GC Counters.

%count-cons-work Meter
The number of words consed times 16. This is used to drive scavenging.

%count-scavenger-work Meter
Scavenger work done. Value is usually less than or equal to %COUNT-CONS-WORK, but can be slightly greater if the scavenger gets a bit ahead of consing.

%tgc-counter Meter
General purpose TGC counter. Currently unused.

%max-generation-0-object-size Meter
Size of the largest object (in words) allowed to be created in generation 0. Larger objects will be created in generation 1.

16.13.7 Hardware Fault Detection Counters.

%count-nubus-gacbl-retries Meter
The number of times the system had to retry a NuBus operation due to receiving a Go Away Come Back Later (GACBL) response from the NuBus.

%count-nubus-parity-retries Meter
The number of times the system had to retry a NuBus operation due to receiving a Parity Error response from the NuBus.

16.13.8 Miscellaneous Counters.

%lowest-direct-virtual-address Meter
The virtual address corresponding to the start of the processor's internal A-Memory. Same address as the constant A-MEMORY-VIRTUAL-ADDRESS, except that READ-METER on the meter will give you a BIGNUM.

%io-space-virtual-address Meter
Lowest virtual address for TV screen memory.

%crash-record-physical-address Meter
Physical address in NVRAM of the currently allocated crash record, or 0 if none allocated.

%tv-clock-rate Meter
The number of TV frames per clock sequence break. The default value is 67 which causes clock sequence breaks to happen about once per second.

%meter-wait-time Meter
The amount of time spent in metering.

%slots-i-own Meter
Bit map for ownership of the hardware boards. Used in multiprocessing environment to determine which boards belong to the Explorer processor.

%cnfg-partition-slot-unit Meter
Slot and unit for reading the configuration partition, or all ones if none used.

%cnfg-partition-name Meter
ASCII name of the configuration partition.

%count-tail-recursions-ignored Meter
The number of times we have to ignore the D-TAIL-REC return type (tail recursion) generated by the compiler and just do a D-RETURN instead.

SECTION 17

Error Handling

17.1 INTRODUCTION

This section explains the signalling of errors. It explains how the microcode signals an error condition to a program or to the user.

This section is organized from the abstract to the concrete. First, the error conditions are described. Then progressively lower levels of the implementation are discussed.

17.2 MICROCODE ERROR CONDITIONS

There are a number of conditions that are signalled by the system microcode. Listed below are the condition flavors, all of which are built directly or indirectly upon ERROR. A later section covers the names of the microcode traps themselves. All of the symbols listed are in the ERROR-HANDLER (EH:) package unless another package name is present.

ARG-TYPE-ERROR: an argument to an operation is not of an acceptable type.

ARRAY-NUMBER-DIMENSIONS-ERROR: attempt to access an array with greater or fewer dimensions than the array has. Built on BAD-ARRAY-ERROR.

BAD-ARRAY-ERROR: generic array problems that lack their own condition.

CALL-TRAP-ERROR: break on entry to a function or on normal exit from a CATCH.

CANT-INITIATE-ON-THIS-DEVICE-TYPE: tried to initiate I/O on an unknown device type.

CELL-CONTENTS-ERROR: the transporter found something bad (but not fatal) in memory. Unbound variables and undefined functions signal other conditions.

DANGEROUS-ERROR: virtual-memory overflow, region-table overflow.

SYS:DIVIDE-BY-ZERO: division by zero.

EXIT-TRAP-ERROR: break on exit from a function.

FLOATING-EXPONENT-OVERFLOW-ERROR: result is too large in magnitude to be represented as a floating point number.

FLOATING-EXPONENT-UNDERFLOW-ERROR: result is too small in magnitude to be represented as a floating point number.

FUNCTION-ENTRY-ERROR: a problem was encountered in entering a function, like too many or too few arguments, or an argument of a wrong data type.

INTERNAL-MEMORY-LOCATION-OOB: out-of-bounds reference to an internal processor memory.

INVALID-FUNCTION: some non-functional object was called as a function.

MAR-BREAK: the Memory Address Register comparator caused a break on a read or write.

PDL-OVERFLOW-ERROR: stack overflow on the regular or special variable PDL.

STEP-BREAK-ERROR: signalled by breakpoints, single-step breaks, and trace breaks.

SUBSCRIPT-ERROR: a subscript for an array access is out-of-bounds, negative, or otherwise in error.

THROW-EXIT-TRAP-ERROR: break on THROW through a marked catch.

THROW-TAG-NOT-FOUND: a THROW was done to a tag for which there is no pending CATCH.

UNBOUND-VARIABLE: an access to a symbol's value cell or to a closure variable found it unbound (DTP-NULL). Built on CELL-CONTENTS-ERROR.

UNDEFINED-FUNCTION: an access to a symbol's function cell or to a method of a select-method found it unbound (DTP-NULL). Built on CELL-CONTENTS-ERROR.

UNIMPLEMENTED-HARDWARE: tried some operation on XBUS or UNIBUS, neither of which exists in the Explorer System.

USER-NUBUS-ERROR: error in user NuBus operation.

SYS:ZERO-TO-NEGATIVE-POWER: attempt was made to raise zero to a negative power.

17.2.1 Microcode Error Table.

This paragraph is a description of microcode error table entries. The microcode error table is read from SYS:UBIN;EXP1-UCODE.TBL#nnn (where nnn is the microcode version) and stored as the value of the variable MICROCODE-ERROR-TABLE. The error table provides information about error trapping locations in the microcode. Each entry in the MICROCODE ERROR TABLE is a list which associates a microcode PC to a symbolic description of the error called an Error Table Entry (ETE). The CAR of the ETE list contains information about how to construct the appropriate condition instance. This information is fully defined in SYS:EH;ERROR-CONDITION-DEFINITIONS and is described below.

Each location in the microcode where TRAP can be called is followed by an ERROR-TABLE pseudo-op. This pseudo-op appears at the PC which is going to be TRAP's return address. An example is:

```
(ERROR-TABLE ARGTYP FIXNUM M-T 0)
```

The CDR of this list is the ETE. The ETE's first element is the name of the error and the second is the first "argument" to that error's associated routines. Note that all ETEs are lists whose CARs are symbols. The symbol is defined by a DEF-UCODE-ERROR form in SYS:EH;ERROR-CONDITION-DEFINITIONS. DEF-UCODE-ERROR tells what flavor of condition instance to build and what to put in it for every microcode trap, keyed by the name of the error.

Error-table entries are listed below alphabetically by type, with the args as a lambda-list. Types not used in the error-handler are marked *. These types seem to be historical remnants. Some of these types are used in CADR or Lambda microcode; they aren't used on the Explorer.

In the argument lists below, an argument of the form "FOO-location" will have the value of one of the following registers when the trap occurs: M-A, M-B, M-C, M-D, M-E, M-F, M-G, M-H, M-I, M-J, M-K, M-L, M-Q, M-R, M-S, M-T, M-1, M-2, M-3, M-4, A-SG-PREVIOUS-STACK-GROUP, PP (the top of the PDL), VMA, MD, or (PP number) where number is a negative index into the PDL. These locations are saved in stack groups and are accessed by EH:SG-CONTENTS for the purposes of examination, replacement, and restarting from errors.

Since these registers are the ones saved in stack groups, they will be pushed on the PDL, written to memory, GC-WRITE-TESTed, and so forth. Therefore, they must be fully tagged or have all ones or zeros in the data type field (M-1, M-2, M-3 and M-4 are exempt from this restriction).

An argument of the form "FOO-tag" is a pseudo-label defined by a RESTART entry. See section on restart below.

17.2.1.1 Special Error Table Entries.

The error table entries in this section are those not directly related to trap messages. They provide useful ancillary information for the processing of traps.

* ARG-POPPED arg1 arg2 arg3 arg4.... Saves (arg1 arg2...) as info on where to find popped args if trap after pop. The error handler collects these but never uses them.

CALLS-SUB subroutine-tag. The subroutine-tag will appear on the >>TRAP line after "->", as an indication of what function called the trapping routine.

* DEFAULT-ARG-LOCATIONS arg1 arg2 arg3.... Saves (arg1 arg2...) as info on where args may be found if no other info is available. The error handler collects these but never uses them.

RESTART restart-tag. Defines restart-tag to be this micro-pc, as a pseudo-lable for other error table entries. A restart-tag marks the micro-pc at which to resume execution, for proceedable traps. Note: RESTART is the only way to define restart-tags. It is not important whether or not the restart-tag is defined as a label in the microcode (they tend to be similar, just for mnemonic value), but it must be unique among restart-tags.

17.2.1.2 Normal Error Table Entries.

The error table entries in this section are used to signal the actual traps. Their names are used in DEF-UCODE-ERRORs to generate condition instances.

AREA-OVERFLOW area-number-location. Signalled during region consing when the area has a maximum size "Allocation in the ~A area exceeded the maximum of ~D. words."

ARGTYP description arg-location &optional arg-number restart-tag function-name. Description is what was expected - see EH:DATA-TYPE-NAMES. It can be a list of allowable types.

Arg-location contains the failing arg. M-1, M-2, M-3 and M-4 are currently not allowed, because the error handler wants a locative to the slot in the erring stack group, and the high bits of M-1, M-2, M-3 and M-4 are stored separately from the pointer field (thats how all 32 bits are preserved). VMA is also not allowed.

Arg-number is the bad argument's number, zero origin.

Restart-tag is the label to restart from, if other than the current pc.

Function-name is the erring function, if that is not the

obvious one.

ARRAY-HAS-NO-LEADER array-location restart-tag. Some array leader operation was attempted on an array with no leader.

ARRAY-NUMBER-DIMENSIONS ignore number-of-dimensions array-location restart-tag. Number-of-dimensions is a constant (array-decode-*), or nil if variable (most cases).

Array-location is where to find the array.

Restart-tag is the label to restart from, if other than the current pc.

BAD-ARRAY-DIMENSION-NUMBER array-location dimension-number-location. "The dimension number ^S is out of range for ^S."

BAD-ARRAY-TYPE array-header-location. The array type of this array was not one of the legal types.

BAD-CDR-CODE address-location. "A bad cdr-code was found in memory (at address ^O)".

BAD-INTERNAL-MEMORY-SELECTOR-ARG object-location. "^S is not valid as the first argument to %WRITE-INTERNAL-PROCESSOR-MEMORIES." Currently, only 1, 2, 4, and 5 are legal.

BIGNUM-NOT-BIG-ENOUGH-DPB. "There is an internal error in bignums; please report this bug."

BITBLT-ARRAY-FRACTIONAL-WORD-WIDTH. "An array passed to BITBLT has an invalid width. The width, times the number of bits per pixel, must be a multiple of 32."

BITBLT-DESTINATION-TOO-SMALL. "The destination of a BITBLT was too small."

BREAKPOINT. Caused by executing a BPT instruction. This is the misc entry smashed in for breakpoints in FEFs.

CALL-TRAP. Microcode support for things like BREAKON. This is the entry half.

CANT-INITIATE-ON-THIS-DEVICE-TYPE device-type-location. "Can't initiate on device type ^S."

CONS-ZERO-SIZE location. Attempt to allocate zero storage.

DATA-TYPE-SCREWUP name. This happens when some internal data structure contains the wrong data type. Name is the type of structure.

DIVIDE-BY-ZERO &optional dividend-location.

EXIT-TRAP. Microcode support for things like breakon. This is the exit half.

FILL-POINTER-NOT-FIXNUM array-location restart-tag "The fill-pointer of the array given to ~S, ~S, is not a fixnum."

FLOATING-EXPONENT-OVERFLOW arg "~S produced a result too large in magnitude to be a :[;small] flonum." "Result is to be placed in M-T and pushed on the pdl. Arg is SFL or FLO. In the case of SFL the PDL has already been pushed."

FLOATING-EXPONENT-UNDERFLOW arg. "~S produced a result too small in magnitude to be a :[;small] flonum." "Arg is SFL or FLO."

FLONUM-NO-GOOD. A subset of argtyp. ARGTYP is not usable.

TOO-FEW-ARGS "Function ~S called with only ~D argument ~1@*~P."

TOO-MANY-ARGS "Function ~S called with too many arguments (~D)."

ILLEGAL-AREA "Tried to cons in free, fixed, or unused-code region. Please report this error."

ILLEGAL-INSTRUCTION. Illegal macroinstructions that aren't unimplemented miscops. "There was an attempt to execute an invalid instruction: ~O."

INDIVIDUAL-SUBSCRIPT-OOB array-location dimension-number-location restart-tag. Dimension-number is the location of the offending dimension's index. "We assume that the current frame's args are the array and the subscripts, and find the actual losing subscript that way."

INSTANCE-LACKS-INSTANCE-VARIABLE var-location instance-location. "Signalled by LOCATE-IN-INSTANCE." "There is no instance variable ~S in ~S."

INTERNAL-MEMORY-LOCATION-OOB memory-selector-location index-location. "Internal memory location is out of range."

MAR-BREAK direction. "The MAR has gone off because of an attempt to write ~S into offset ~O in ~S." "The MAR has gone off because of an attempt to read from offset ~O in ~S." Direction is WRITE or READ. This trap is for the MAR feature.

MICRO-CODE-ENTRY-OUT-OF-RANGE misc-number-location. "MISC-instruction ~S is not an implemented instruction."

NO-CURRENTLY-PREPARED-SHEET location. "There was an attempt to draw on the sheet ~S without preparing it first."

NO-MAPPING-TABLE. "Flavor ~S is not a component of SELF's flavor, ~S, on a call to a function which assumes SELF is a ~S."

NO-MAPPING-TABLE-1. "SYS:SELF-MAPPING-TABLE is NIL in a combined method."

NONEXISTENT-INSTANCE-VARIABLE. "Compiled code referred to instance variable ~S, no longer present in flavor ~S."

NUMBER-ARRAY-NOT-ALLOWED array-location restart-tag. "The array ~S, which was given to ~S, is not allowed to be a number array." This one occurs when making a locative to an array element. None of the current uses has a restart-tag.

NUMBER-CALLED-AS-FUNCTION number-location. "The number, ~S, was called as a function."

PDL-OVERFLOW pdl-type. "The ~A push-down list has overflowed." Pdl-type is either REGULAR or SPECIAL.

RASTER-WIDTH-TOO-WIDE. The raster width of a font was too wide.

RCONS-FIXED. "There was an attempt to allocate storage in the fixed area ~S." The area number is in M-S.

REGION-TABLE-OVERFLOW. "Unable to create a new region because the region tables are full."

RPLACD-WRONG-REPRESENTATION-TYPE first-arg-location. "Attempt to RPLACD a list which is embedded in a structure and therefore cannot be RPLACD'ed. The list is ~S." First-arg-location tells where to find the first arg to RPLACD.

SELF-NOT-INSTANCE. "A method is referring to an instance variable, but SELF is ~S, not an instance."

SG-RETURN-UNSAFE. "An unsafe stack group attempted to STACK-GROUP-RETURN." "No args, since the frob is in the previous-stack-group of the current one."

STACK-FRAME-TOO-LARGE. "Attempt to make a stack frame larger than 256. words." Called from %ASSURE-PDL-ROOM.

STEP-BREAK. Interface to microcode support for single-stepping.

SUBSCRIPT-OOB index-location limit-location restart-tag indices-flag. Index-location is where to find the index used, and should always be present.

Limit-location is where to find the legal limit for the subscript, and should always be present.

Restart-tag may be a list, which will be pushed

sequentially. "This is used to get the effect of making the microcode restart by calling a subroutine which will return to the point of the error."

Indices-flag is "T if indices are on the stack, 1 if ar-1-force (etc.) and there is only one index, or absent if array's rank should be used to decide where the args are."

*THROW-TAG-NOT-FOUND.

TRANS-TRAP For the conditions unbound-symbol, unbound-instance-variable, unbound-closure-variable, undefined-function, bad-data-type-in-memory.

"The variable ~S ~A unbound."

"The function ~S ~A undefined."

"The instance variable ~S ~A unbound in ~S."

"The variable ~S ~A unbound (in a closure value-cell)."

"The word #<~S ~S> was read from location ~O ~O[in~A~]."

USER-NUBUS-ERROR high-address-location low-address-location nubus-tms-type-location "User NuBus Error of type ~S, at address #x ~16R 16,6, 48^R. %Error Bits: #x ~16R."

USER-NUBUS-GACBL-LIMIT. "Number of GACBLs exceeded limit in user NuBus operation."

USER-NUBUS-PARITY-LIMIT. "Number of Parity Errors exceeded limit in user NuBus operation."

VIRTUAL-MEMORY-OVERFLOW. "You've used up all available virtual memory!"

WRITE-IN-READ-ONLY address-location. "There was an attempt to write into ~S, which is a read-only address."

WRONG-SG-STATE sg-location. "The state of the stack group, ~S, given to ~S, was invalid." Sg-location is where to find the invalid stack group.

ZERO-ARGS-TO-SELECT-METHOD select-method-location. "~S was applied to no arguments."

SECTION 18

Crash Handling

18.1 ILLEGAL OPERATION

When the microcode detects an irrecoverable or "can't happen" error, it will crash the system. This process is known as ILLOP after the Illegal Instruction Operation microcode routine that performs it.

ILLOP causes the machine to halt. Further operation in the presence of the irrecoverable error may only worsen the situation or complicate it beyond analysis. Instead, ILLOP will make notes about the error in a crash record stored in the nonvolatile NVRAM on the System Interface Board, and then halt the machine.

ILLOP is a very low level routine. It assumes very little about the state of the machine and it will just halt if it detects that any of its assumptions are wrong. It does not require that any of interrupts, device support, virtual memory, storage allocation, garbage collection, Lisp object support, function calling, or instruction execution be intact. It does assume that the processor is functioning properly; that the registers A-Zero, M-Zero, A-Ones, and M-Ones are set up to contain 0 or -1 as appropriate; that the NuBus is available; and the the NVRAM can be read and written.

After the crash RAM has been written, the crash is indicated by complementing the video sense of the screen. The effect is dramatic. This may fail if the memory interface or the screen interface is not functioning properly but a failure of this operation will not affect the proper recording of the crash in the crash record.

The analysis of crash records is performed by the Lisp crash analyzer. The crash analyzer functions are documented fully in the Explorer Tools and Utilities manual, as are many of the higher level details of crash reporting. This section is intended to be a supplement to that discussion directed at systems programmers. For the final word, consult the Lisp code in SYS:NVRAM; CRASH and the other files in the NVRAM system.

NOTE .

Unless otherwise indicated, all symbols and functions named here are in the SYSTEM package.

18.2 CRASH RECORDING

ILLOP stores some of the machine state in the NVRAM so that the next successful startup can explain the cause of the crash. If the system cannot be successfully started, field service can read the crash reason from diagnostic hardware and/or software. This data is called a crash record.

In order that an unsuccessful attempt to restart the system will not lose the original crash data, crash records for the last few system shutdowns are kept in a circular buffer in NVRAM. Each time the system is started a record is allocated from the buffer. When the system halts, the reason is recorded in the crash record.

NOTE

In order for proper recording of crash information to take place, the structure of NVRAM must first be initialized using the SETUP-NVRAM function. Only then can valid crash records be recorded. Since NVRAM is non-volatile memory, the SETUP-NVRAM function should only need to be run when the system is first installed or after service maintenance has been performed on the System Interface Board.

18.2.1 Crash Record Allocation.

Allocation of a crash record for the current system startup is performed by the microcode during the boot process. It occurs as early in the boot as possible so that useful information can be recorded by ILLOP if the machine crashes during the boot.

Allocation of the crash record is controlled by four 16-bit numbers that are stored at a known place within the NVRAM. These are shown in Table 18-1. All of the allocation registers contain 16-bit byte offsets into the NVRAM, which is accessed using physical memory references. Offsets are expressed in hexadecimal.

NVRAM is actually 8-bit memory mapped into the NuBus address space one-byte-per-32-bit-NuBus-word; hence only multiple-of-four byte addresses are used (eg, 0, 4, 8, #xC, #x10, ...). A 16-bit quantity (such as the crash record allocation registers) is stored with its low order bits in the lowest address and its high order bits in the address 4 higher. For example, #xF4B2 would be stored in NVRAM-CRASH-BUFF-POINTER with #xB2 in NVRAM-BASE + #x90 and #xF4 in NVRAM-BASE + #x94.

NVRAM-CRASH-BUFF-POINTER is the offset into the NVRAM (in bytes) of the beginning of the currently selected crash record. The currently selected crash record describes the current system startup. When the system is running, it points to the record that will be filled in when the system next halts. When the system is not running, it points to the crash record for the last system shutdown.

NVRAM-CRASH-BUFF-REC-LEN is the size of a crash record. It is the amount by which to increase NVRAM-CRASH-BUFF-POINTER to reach the next record.

NVRAM-CRASH-BUFF-LAST is the offset to the beginning of the last crash record in the buffer. This is used by allocation and also when scanning the buffer backward to see the history of shutdowns.

NVRAM-CRASH-BUFF-BASE is the offset to the beginning of the first crash record in the buffer. This is used by allocation to "wrap around" the buffer.

The algorithm to allocate a new crash record, then, is: Add NVRAM-CRASH-BUFF-REC-LEN to NVRAM-CRASH-BUFF-POINTER. The result is the new NVRAM-CRASH-BUFF-POINTER. If the new NVRAM-CRASH-BUFF-POINTER is greater than NVRAM-CRASH-BUFF-LAST then it should be reset to NVRAM-CRASH-BUFF-BASE.

The algorithm for finding the previous crash record is: Subtract NVRAM-CRASH-BUFF-REC-LEN from NVRAM-CRASH-BUFF-POINTER. If this is less than NVRAM-CRASH-BUFF-BASE, it should be set to NVRAM-CRASH-BUFF-LAST.

Table 18-1 Crash Record Allocation Registers

NVRAM-BASE plus (hex)	Allocation Register
80	NVRAM-CRASH-BUFF-FORMAT-PROCESSOR
88	NVRAM-CRASH-BUFF-FORMAT-REV
90	NVRAM-CRASH-BUFF-POINTER
98	NVRAM-CRASH-BUFF-REC-LEN
A0	NVRAM-CRASH-BUFF-LAST
A8	NVRAM-CRASH-BUFF-BASE

18.2.2 Crash Record Contents.

The crash record format is shown in Table 18-2. The templates for the crash table, for the rest of NVRAM, and for other information used by crash record support can be found in the file SYS:UCODE;LROY-GDEV. The list CRASH-RECORD-OFFSETS contains a list of symbolic names whose values are these offsets.

Table 18-2 Crash Record Format

NVRAM-CRASH-BUFF-POINTER plus (hex)	Contents
	General information
0	Progress field. Indicates how far into the boot process the system progressed before halting.
	Load information
4	Disk controller slot number
8	Disk device number for microload
C	Disk device number for Load Band
10	Microload name (4)
20	Load Band name (4)
30	Microload version (2)
38	Load Band version (2)
40	Load Band revision (2)
	Date and time information
48	Month of boot
4C	Day of boot
50	Year of boot
54	Hour of boot
58	Minute of boot
5C	Month of crash
60	Day of crash
64	Year of crash
68	Hour of crash
6C	Minute of crash
	Flags field
70	Report Flags
	Shutdown information
74	Halt Location (2)
7C	Halt Kind
	Saved Registers
80	contents of M-1 (4)
90	contents of M-2 (4)
A0	contents of MD (4)
B0	contents of VMA (4)
C0	contents of M-FEF (4)
D0	contents of UPC-1 (2)

D8	contents of UPC-2 (2)
E0	Location counter (4)
F0	contents of M-T (4)
100	Length of crash record

The progress field indicates how far into the boot process the system progressed before crashing. Currently supported values for this field are listed in the variable CRASH-RECORD-PROGRESS-CODES.

The load information fields are initialized during crash record allocation by the microcode. They reflect the load information saved by the boot process.

The boot time is written to the crash record by a function on the WARM-INITIALIZATIONS-LIST after the system time base has been initialized. The crash time fields are updated about every five minutes from Lisp by the TM-Update process so that they will accurately reflect the time if the machine halts.

The report flags are used to store information about which crash records have been logged to a crash-log file by the crash analyzer (see section on crash analyzer below). It is also contains a flag that is set if this boot was a warm boot so that information can be displayed in the crash record. Crashs that occur after a warm boot or warm boot attempt are often caused by problems in the warm booted environment, and hence are not as interesting as crashes that occur in a cold booted environment.

The halt kind field indicates the type of the last shutdown. The field is initialized to the System Boot state by the crash record allocation microcode, then updated during ILLOP to reflect the crash kind. Possible values for halt kind are:

* System Boot. The last shutdown was caused by a cold boot sequence or by a warm boot sequence during normal operation (that is, not a warm boot initiated after the machine crashed). Note that since the crash record allocation routine is called from warm-boot, the system can be warm booted after an abnormal shutdown and still retain all the crash record information from that shutdown. However, if the machine was in a hung state when warm or cold booted, ILLOP processing will not be done, and the shutdown will be reported in this category. META-CONTROL-META-CONTROL-C can be used to force a crash from a hard run, and therefore produces a crash record.

* Microcode Halt. This indicates that the last crash was called from the microcode when it detected an unrecoverable error condition. In this case, ILLOP stores the micro-pc address from which ILLOP was called in the halt address field. This micro-pc is later looked up in the crash table database by the crash analyzer in order to provide the user with a text description of the microcode crash reason. See the section on the crash analyzer, below.

* Hardware Halt. This halt kind is currently unsupported. All detectable hardware halt conditions (such as memory parity errors, illegal page faults, and so forth) currently fall into the microcode halt category.

* Lisp Halt. The last halt was called by Lisp through the si:%crash function. In this case, a Lisp crash code (which is one of the arguments to %CRASH) is stored in the halt address field. The second argument to %CRASH (an object) is stored in the M-1 field. Currently, the only valid Lisp crash code is 0 which indicates a normal system shutdown called from Lisp. This code is seen when the system was halted by a user-initiated call to either SHUTDOWN or SYSTEM-SHUTDOWN.

The saved registers fields contain the values of the indicated processor registers when ILLOP was called. Their values can appear in the microcode crash descriptor text reported by the crash analyzer, and in any case their values are labeled and displayed in the crash analysis report.

18.2.3 The Crash Table.

The crash analysis database consists of the table of crash codes and crash descriptions produced by the micro assembler. This table is kept in the file SYS:UBIN;<ucode-name>.CRASH#nnn where nnn is the microcode revision number and ucode-name is obtained by looking up the microcode name associated with the current value of MICROCODE-TYPE-CODE in the *MICROCODE-NAME-ALIST*. The crash analyzer loads the appropriate version of this file into memory when a microcode crash is being reported. The crash table file contains a complete list of all current crash descriptor texts.

Note that there may be some microcode crashes that will not have a description in the crash database. This fact will be reported by the crash analyzer. The crash micro-pc for such crashes is valid, however, and can be used to determine the path taken to ILLOP.

The crash table file is generated by the microassembler. At points in the microcode where ILLOP may be called a CRASH-TABLE pseudo-op is generated. From this a Crash Table Entry (CTE) is generated and added to the list of CTEs that make up the crash table file. The format of a CRASH-TABLE pseudo-op and the resulting CTE are as follows:

Ucode CRASH-TABLE pseudo-op:

```
General form:
   (CRASH-TABLE <format string> <one or more format args>)
Example:
   (CRASH-TABLE "Nasty data type ~a read from address ~x"
      (Q-DATA-TYPES (LDB %%Q-Data-Type MD))  VMA)
```

Crash Table Entry (in .CRASH file):

```
General form:
   (<Micro PC> <format string> <one or more format args>)
Example:
   (34174 "Nasty data type ~a read from address ~x"
      (Q-DATA-TYPES (LDB %%Q-Data-Type MD))  VMA)
```

The first element of the CTE is the microcode PC address from which the call to ILLOP was made. The second element is usually a string. In this case we essentially do the following to format the crash description.

```
(APPLY #'FORMAT <stream> (CDR <cte>))
```

This works because the saved register values in the crash record are bound to special variables named VMA, MD, M-FEF, etc., at the time this FORMAT is executed.

If the second element of a CTE is a symbol, we look for the REPORT property on its property list. If present, it will be a report function to run.

Note that the file SYS:NVRAM;ANALYSIS-FUNCTIONS contains a number of crash-description formatting routines that have very detailed knowledge of the contents of the saved microcode registers. These routines must be kept in synch with current microcode usage, just as with the error handler files.

SECTION 19

Compiler Notes

This section supplements Section 21 of the Explorer Lisp Reference manual with some additional information that might be of interest to system programmers.

19.1 COMPILE-TIME PROPERTIES OF SYMBOLS

When symbol properties are referred to during macro expansion, properties defined in a file should be in effect for the compilation of the rest of the file. This does not happen if GET and DEFPROP are used, because the DEFPROP is not executed until the file is loaded. Instead, you can use GETDECL and DEFDECL. These are normally the same as GET and DEFPROP, but during file-to-file compilation, they also refer to and create declarations.

SYS:FILE-LOCAL-DECLARATIONS
During file-to-file compilation, the value of this variable is a list of all declarations that are in effect for the rest of the file. Macro definitions, DEFDECL's, PROCLAIM's and special declarations that come from DEFVAR's are all recorded on this list.

GETDECL function-spec property
This function is a version of GET that allows the properties of a function to be overridden by local declarations.

If LOCAL-DECLARATIONS or SYS:FILE-LOCAL-DECLARATIONS contain a declaration of the following form:

(property function-spec value)

GETDECL returns value. Otherwise, GETDECL returns the result of the following form:

(function-spec-get function-spec property)

The GETDECL function is typically used in macro definitions. For example, the SETF macro uses GETDECL to obtain the SETF property of the function in the expression for the field to be set.

PUTDECL function-spec property value
The PUTDECL function causes (GETDECL function-spec property) to return value.

The PUTDECL function makes an entry on SYS:FILE-LOCAL-DECLARATIONS of the following form:

(property function-spec value)

This form stores value where GETDECL can find it; but if PUTDECL is called during compilation, it affects only the rest of that compilation.

DEFDECL symbol property value
When executed, DEFDECL resembles PUTDECL except that the arguments are not evaluated. This special form is usually the same as DEFPROP except for the order of the arguments.

Unlike DEFPROP, when DEFDECL is encountered during file-to-file compilation, it is executed, creating a declaration that remains in effect for the rest of the compilation. (The DEFDECL form also goes into the xld file to be executed when the file is loaded). The DEFPROP special form would have no effect whatever at compile time.

The DEFDECL special form is often useful as a part of the expansion of a macro. It is also useful as a top-level expression in a source file.

Consider the following form:

```
(DEFDECL FOO SETF ((FOO X) . (SET-FOO X SI:VALUE)))
```

The preceding form in a source file allows the following form to be used in functions in that source file; and by anyone, once the file is loaded:

```
(SETF (FOO ARG) VALUE)
```

COMPILER:*LOCAL-DECLARATIONS-SPECIFIERS*
This variable is a list of declaration specifier symbols for which the compiler will push the declaration onto the LOCAL-DECLARATIONS list for access by GETDECL. Users can define new declarations by pushing the name onto this list. Compatibility note: in Explorer Releases 1 and 2, all declarations were pushed on LOCAL-DECLARATIONS, but Release 3 is more selective for the sake of efficiency.

19.2 DECLARATIONS

The following declaration specifiers are used internally in addition to those documented in section 13 of the Explorer Lisp Reference manual.

:EXPR-SXHASH number
At the beginning of the body of a DEFMACRO, DEFSUBST, or inline function, a (DECLARE (:EXPR-SXHASH <number>) may be used to specify the hash code that will be recorded for the macro. This hash code is used for giving warnings when loading a file that uses macros whose definitions have changed; it is normally computed by hashing the definition. The :EXPR-SXHASH declaration can be used to cause a slightly modified version of a macro to have the same hash code as the previous version (obtained from the debug-info) in order to suppress these warnings.

COMPILER:TRY-INLINE function-name ...
This is similar to an INLINE declaration except that the inline expansion will be used only if, after optimization, it is not significantly larger than the original function call. This is useful for functions that can be optimized down to something trivial when some of the arguments are constants or known to be of a particular type, but are too big to be included inline in the general case.

SYS:DOWNWARD-FUNCTION
The declaration (DECLARE SYS:DOWNWARD-FUNCTION) can be used at the beginning of the body of a LAMBDA expression or local function to inform the compiler that the lexical closure which is being created is only being passed downward and there will be no references to it after execution leaves the context in which it was created. This enables the compiler to use the instruction MAKE-EPHEMERAL-LEXICAL-CLOSURE instead of MAKE-LEXICAL-CLOSURE, thereby saving some run-time overhead. Note that in many cases, the compiler is able to figure this out for itself without needing the declaration, and that incorrect use of the declaration could cause severe problems.

19.3 XLD FILES

All xld files are composed of 16-bit bytes. The first two bytes in the file contain fixed values, which are present so that the system can tell a proper xld file. The next byte is the beginning of the first group. A group starts with a byte that specifies an operation. It can be followed by other bytes that are arguments.

Most of the groups in an xld file are present to construct objects when the file is loaded. These objects are recorded in the fasl-table. Each time an object is constructed, it is assigned the next sequential index in the fasl-table. The indices are used by other groups later in the file to refer back to objects already constructed.

To prevent the fasl-table from becoming too large, the xld file can be divided into whacks. The fasl-table is cleared out at the beginning of each whack.

The other groups in the xld file perform operations such as evaluating a list previously constructed or storing an object into a symbol's function cell or value cell.

19.4 OPTIMIZATION

Besides the functions COMPILER:ADD-OPTIMIZER and COMPILER:OPTIMIZE-PATTERN which are described in section 21.8 of the Explorer Lisp Reference manual, the following functions may be used to specify optimizations:

COMPILER:FOLD-CONSTANT-ARGUMENTS function-name
: Tells the compiler that if it sees a call to the designated function in which all of the arguments are constants, then it can call the function at compile-time and replace the function call with a QUOTE form containing the resulting value. This also implies that the function has no side-effects, so calls can be deleted if their value is not used.

COMPILER:DEFCOMPILER-SYNONYM &quote function synonym-function
: Both arguments should be symbols, and are not evaluated. When the compiler sees the first argument used as the name of a function to be called, it will compile code to call the function named by the second argument instead.

COMPILER:OPTIMIZE-STATUS
: This function can be called to find out the current values of the OPTIMIZE switches. The value returned is in the form

of a declaration specifier so that one can do:

```
(SETQ SAVE-OPT (COMPILER:OPTIMIZE-STATUS))
...
(PROCLAIM SAVE-OPT)
```

to save and restore the switches.

19.5 COLD-LOAD ATTRIBUTE

Putting the attribute COLD-LOAD:T in the mode line of a file lets the compiler know that the file is part of the minimal kernel, or in other words, it is loaded by Genasys into the cold band. This knowledge is used in the following ways:

1. The compiler can warn you about using features that Genasys does not support. For example:
   a. Genasys can't handle load-time evaluation of constants [reader macro #,].
   b. (PROCLAIM '(TYPE ...)) doesn't work until after the compiler is loaded.
2. The compiler can give special handling where needed. For example, package commands such as EXPORT that don't supply an explicit package argument are automatically given one by the compiler because defaulting to *PACKAGE* is not appropriate during "crash-list" evaluation.
3. The compiler can warn you about using functions that are not in the cold load. Anyone who has ever had to debug a band that won't boot because it is trying to call an undefined function should appreciate the value of this. More about this below.
4. Since the COLD-LOAD attribute identifies the file as part of the kernel, style warnings for use of internal functions are suppressed. For example, you can use microcoded functions such as MEMQ and FIND-POSITION-IN-LIST without getting a complaint that they are obsolete.
5. Top-level forms which are going to be evaluated at load time will be fully macro-expanded at compile time in order to minimize the amount of work done in evaluating the "crash-list" when the band is booted and to avoid potential problems with macros that are undefined or that use things that aren't initialized yet.

Warnings about use of functions that are not in the cold load are

implemented by checking the source file pathname of each function that is called to see if it has the COLD-LOAD attribute also. Thus, for these warnings to be meaningful, it is essential that all the files that are part of the cold load have the attribute. To avoid getting irrelevant warnings, functions that are not really needed in the cold load should be in separate files from those that are. Currently, these warnings are suppressed in COMPILE-FILE (they appear only when compiling in an editor buffer or with SAFETY>1) to avoid getting a great many warnings during system builds. For cases where you really do need to reference a function that won't really be called in the cold band, you can suppress the warning by binding the INHIBIT-STYLE-WARNINGS-SWITCH flag. For example:

```
(IF (FBOUNDP 'FOO)
    (COMPILER-LET ((INHIBIT-STYLE-WARNINGS-SWITCH T))
      (FOO X)))
```

SECTION 20

Macro-Instructions

20.1 INTRODUCTION

Probably the most confusing thing to new users is determining how the macro-instruction set differs from the "core" Lisp routines (that is, those Lisp routines needed to define a minimum Lisp kernel). This instruction set (the macro-instructions) is a "pseudo" subset of the Lisp instruction set. In other words, these instructions are those that are implemented by microcode and are at the level below the compiler. It is the set available to the compiler in constructing functional content. Some Lisp instructions map directly into macro-instructions (therefore, you can get an ample description of what the instruction does by consulting the Explorer Lisp Reference manual), while others are a bit more complex and require more effort from the compiler. Still other macro-instructions don't correspond to basic Lisp functions at all, but are used in other aspects of the virtual machine implementation (such as in virtual memory or storage management). There are also some instructions here that are only available to the compiler and so are not classified as Lisp instructions (for example, Exchange).

The purpose of this section is to describe in more detail the macro-instruction set so that if you were the compiler or debugging compiler-generated code you could easily determine exactly what arguments are needed for a macro-instruction, what side effects a macro-instruction may have, and finally, what should be returned to you on the stack. Therefore, you will find in this section more detail on datatypes, exact error checking being performed, stack manipulation, and some history behind why some instructions exist and how they are used.

NOTE

The section entitled The Disassembler in the Explorer Lisp Reference manual explains how to use the DISASSEMBLE function, how the macro-instruction set works and how to understand the behavior of code written in this instruction set. That section should be read before proceeding with this section.

NOTE

The instruction set for Release 3 is defined by the file SYS:UCODE;DEFOP which uses several macros that are defined in SYS:COMPILER;TARGET. Byte specifiers for the instruction fields can be found in SYS:UCODE;DEF-ELROY.

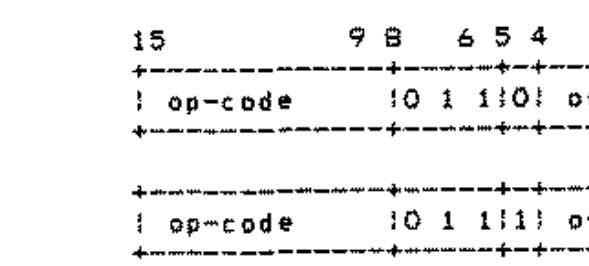
20.2 MAIN OPS

MAIN OP instructions (see Figure 20-1) have an operand address as part of the instruction (called base-arguments in this document) and may take additional operands from the stack. The disposition of the result is implied by the operation. All the base registers are 0-origin; that is, offset 0 is the first element.

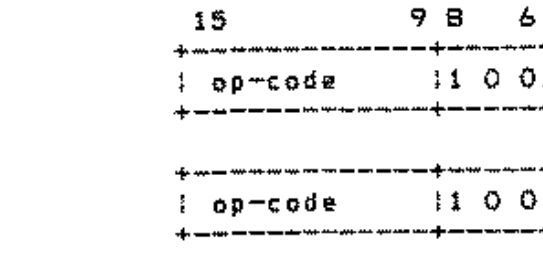
base register: 0 = FEF
1 = FEF+64
2 = FEF+128
3 = higher lexical context
4 = SELF mapping table
5 = local variables
6 = arguments
7 = PDL-POP [offset not used]

Figure 20-1 MAIN OP Instruction Format

For a base register value of 3, the offset field is interpreted as follows:



For a base register value of 4, the offset field is interpreted as follows:



MAIN-OPS are defined by the special form DEFOP.

20.3 SHORT BRANCHES

The Short Branch instruction format is shown in Figure 20-2. The op-code identifies the instruction as a branch and specifies the condition to be tested. The lower 9 bits of the instruction are a signed displacement relative to the address of the next instruction.

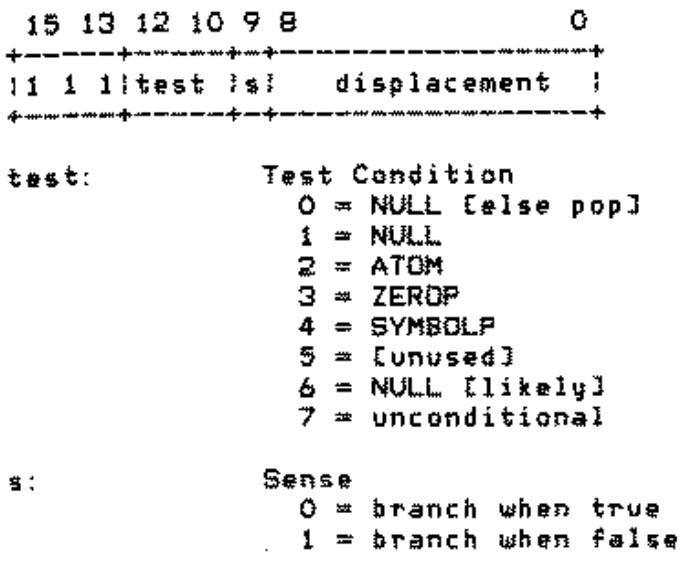


Figure 20-2 Short Branch Instruction Format

If the displacement cannot be encoded within 9 bits, then a Long Branch is used. Long branches are separate op-codes in the AUX-OP group (see below).

Short branch instructions are defined using the special form DEF-BRANCH-OP.

20.4 IMMEDIATE OPERATIONS

Immediate operations (see Figure 20-3) use the lower 9 bits of the instruction word in special ways. PUSH-NUMBER uses it as a FIXNUM operand. Various op-IMMED instructions use it as a signed integer operand. These are defined by using DEFOP with the :NO-REG option of Immed.

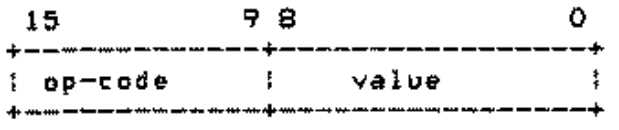
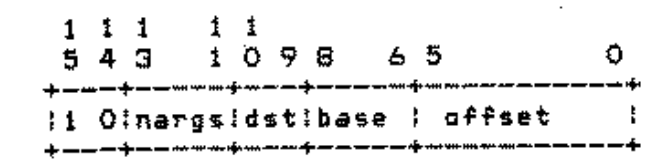


Figure 20-3 Immediate Operation Instruction Format

20.5 CALL INSTRUCTIONS

The instructions CALL-0 through CALL-6 have the format shown in Figure 20-4.



nargs: Number of Arguments

dst: Destination
0 = set indicators only
1 = push result on stack
2 = return from current frame
3 = replace current frame

Figure 20-4 Call Instruction Format

The base and offset fields are the same as for the MAIN-OPS and specify the function to be called. The function arguments are pushed on the stack before executing this instruction. If the number-of-arguments field contains a 7, then this is a CALL-N instruction, and the number of arguments is the last thing that was pushed on the stack.

The various call instructions are defined using the special form DEF-CALLOP.

20.6 MISC-OPS

MISC-OPS (see Figure 20-5) take their arguments from the stack and produce a result value which sets the indicators and is optionally pushed on the stack.

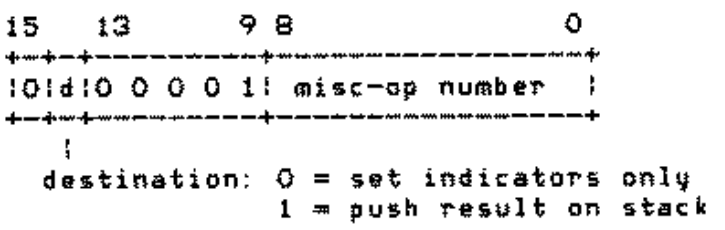


Figure 20-5 MISC-OP Instruction Format

The lower 9 bits of the instruction specify which of the many MISC-OPS is to be performed.

MISC-OPS are defined by using the special form DEF-MISC-OP.

20.7 AUX-OPS

AUX-OPS (see Figure 20-6) are similar to MISC-OPS except that they do not produce any result value.

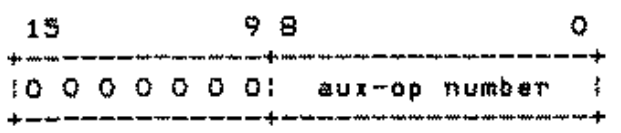
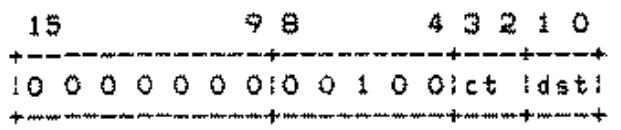


Figure 20-6 AUX-OP Instruction Format

The lower 9 bits of the instruction specify which of the AUX-OPS is to be performed.

AUX-OPS are defined using the special form DEF-AUX-OP. AUX-OPS can be considered to be divided into four groups: simple AUX-OPS, Complex Calls, Long Branches, and AUX-OPS with a count field.

20.7.1 AUX-OP Complex Call.

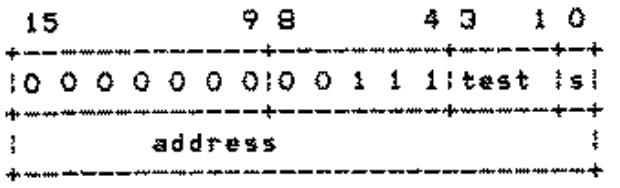


ct: Call Type
- 0 = Use call-info word. Push arguments, mapping table (optional), call-info word and function.
- 1 = APPLY (with one argument). Push argument and function to be called.
- 2 = [unused]
- 3 = LEXPR-FUNCALL-WITH-MAPPING-TABLE (one argument). Push argument, mapping table and function.

dst: Destination
- 0 = set indicators only
- 1 = push result on stack
- 2 = return from current frame
- 3 = replace current frame

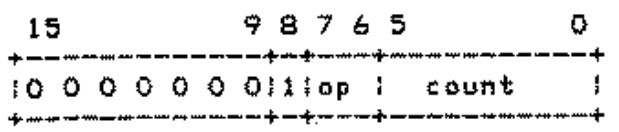
See the section on function calling for a description of the call-info word.

20.7.2 AUX-OP Long Branch.



The Test and Sense fields have the same values as for a short branch. Instead of a signed relative displacement, the second half-word of the instruction contains the new PC offset from the start of the FEF.

20.7.3 AUX-OPS With Count Field.



op: Operation
0 = unbind "count" special variables
1 = pop "count" values off stack
2 = return "count" values from stack
3 = [unused]

20.8 AREFI INSTRUCTIONS

This group of instructions (see Figure 20-7) is used for single-dimension array references having a small constant index. The index is an immediate value in the instruction, and the other operands are taken from the stack.

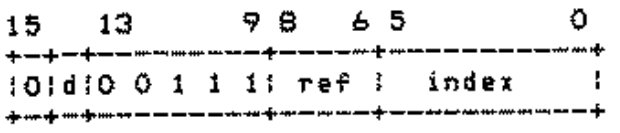


Figure 20-7 AREFI Instruction Format

20.9 MODULE GROUP

This group of instructions (see Figure 20-8) are similar to MISC-OPS, except that they tend to be specific for certain applications or environments, and the microcode that implements a module is not required to be present.

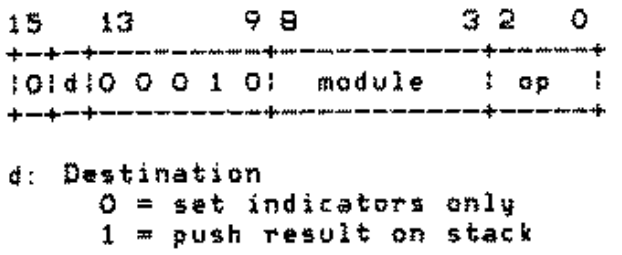


Figure 20-8 Module Group Instruction Format

The operation to be performed is specified by a 6-bit module number and a 3-bit operation code within that module. The currently assigned module numbers are:

0 TV (various %DRAW-<thing> functions)
1 Mouse

with more to be defined later.

20.10 UCODE ENTRIES

A Ucode entry (which stands for microcode entry) is not really a macro-instruction at all. Rather it is a functional object: a word of data type DTP-U-ENTRY. It can be funcalled, applied to arguments, passed as a value, and so forth just like any other Lisp functional object. The use of the pointer field of a U-ENTRY is detailed in the section on Internal Storage Formats.

Ucode entries are listed here because they closely resemble MISC-OPs and for completeness because all such functions must be defined by the DEFOP file. A Ucode entry, like a MISC-OP, is a microcoded function which takes its arguments from the stack. However, the number of arguments is variable, so the function can have an &REST arg.

20.11 MACROCODE INSTRUCTION SET

The following is the syntax line for a macro-instruction:

SOME-MACROINSTRUCTION arg1 arg2 [opcode] FORMAT Level description

- The name of the macro-instruction will be in upper case.

- Arguments are listed; they are to be pushed on the stack, arg1 being the first one pushed on the stack. The last argument listed is the one on top of the stack. If an argument is described with the prefix immed-, then the argument is not pushed on the stack, rather it is included in the instruction itself (see the IMMEDIATE instruction description above). If an argument is described with the prefix base- then the argument is also not pushed on the stack but rather addressed with the MAIN-OP base-offset scheme described above.

- Brackets ([]) are used to enclose the opcode for the macro-instruction (specified in octal).

- The FORMAT of the macro-instruction will be on the far right and in upper case. It will be MAIN-OP, SHORT-BRANCH, IMMEDIATE, CALL, MISC-OP, AUX-OP, AREFI, or MODULE(MODULE-NAME).

- Level is either Lisp-Function or blank. This indicates if there is a corresponding Lisp function for this macro-instruction by the same name. If there is a corresponding Lisp function for this macro-instruction by another name, then the Lisp function will be referred to in description.

- A brief description will then follow.

- All MISC-OP instructions pop arguments that are on the stack and return a value on the top of stack unless otherwise indicated.

- All MAIN-OP instructions use the base-offset scheme to address their first argument (base-argument) and all other arguments are popped off the stack.

- Most instructions set the indicators. Some exceptions are: branches, (AUX) POP-PDL, (AUX)

UNBIND, LEXICAL-UNSHARE, and (AUX) LEXICAL-UNSHARE-ALL.

The following is an alphabetic list of macro-instructions.

ABS num [513] MISC-OP Lisp-Functio
Returns the absolute value of num which can be any type of number.

ADD-IMMED num, immed-y [43] IMMEDIATE
Returns the sum of num and the immediate operand, immed-y, which is part of the ADD-IMMED instruction.

ALOC array, &REST subscripts [2] UCODE ENTRY
Returns a locative to the element of array specified by subscripts.

AP-1 array, index [645] MISC-OP Lisp-Functio
Returns a locative to the element of array specified by index. This is the one-dimensional case of ALOC.

AP-1-FORCE array,index [652] MISC-OP Lisp-Functio
Returns a locative to the element of array specified by index. Array is treated (forced) as one-dimensional. That is, it is indexed with a single subscript regardless of its rank.

AP-2 array, sub1, sub2 [646] MISC-OP Lisp-Functio
Returns a locative to the element of array specified by subscripts sub1 and sub2.

AP-3 array, sub1, sub2, sub3 [647] MISC-OP Lisp-Functio
Returns a locative to the element of array specified by subscripts sub1, sub2 and sub3.

AP-LEADER array, index [644] MISC-OP Lisp-Functio
Returns a locative to the leader element of array specified by the subscript index.

APPLY-TO-INDS fn, args [104] AUX-OP
Apply function fn to the list of arguments args.
Set the indicators with the value returned.

APPLY-TO-PUSH fn, args [105] AUX-OP
Apply function fn to the list of arguments args.
Push the returned value on the stack.

APPLY-TO-RETURN fn, args [106] AUX-OP
Apply function fn to the list of arguments args.
Return the result value from the current function.

APPLY-TO-TAIL-REC fn, args [107] AUX-OP
Apply function fn to the list of arguments args.

Replace the current stack frame and return the result value.

ZLC:AR-1 array, index [641] MISC-OP Lisp-Function
Returns the element of the one-dimensional array array specified by index. Array must be a one-dimensional array and index must be a FIXNUM. If index is less than zero or greater than the largest index permissible, then a SUBSCRIPT-OOB error is signalled. If array is not one-dimensional, then ARRAY-NUMBER-DIMENSIONS is signalled. The type of result depends on the type of array.

AR-1-FORCE array, index [651] MISC-OP Lisp-Function
Returns the element of the array array specified by index. Array is treated (forced) as a one-dimensional array; i.e., it is indexed with a single subscript regardless of its actual rank.

AR-2 array, sub1, sub2 [642] MISC-OP Lisp-Function
Exactly like AR-1 except array must be two-dimensional.

ZLC:AR-2-REVERSE array, sub2, sub1 [650] MISC-OP Lisp-Function
Returns the element of the two-dimensional array array. See the Explorer Lisp Reference manual for a discussion of this instruction.

AR-3 array, sub1, sub2, sub3 [643] MISC-OP Lisp-Function
Exactly like AR-1 except array must be three-dimensional.

AREF array, &REST subscripts [1] UCODE ENTRY
Returns the element of the array array specified by subscripts.

ARRAY-ACTIVE-LENGTH array [662] MISC-OP Lisp-Function
Returns the number of "active" elements in array. If the array has a fill pointer then the fill pointer value is returned, else the number of elements is returned.

ARRAY-DIMENSION array, dimension [665] MISC-OP Lisp-Function
Returns the length of dimension dimension of array. The first dimension is number 0.

ARRAY-HAS-FILL-POINTER-P array [241] MISC-OP Lisp-Function
Returns T if array has a leader and leader element 0 is a FIXNUM. Otherwise, returns the symbol NIL. If array is not an array an ARGTYP error is signalled.

ARRAY-HAS-LEADER-P array [234] MISC-OP Lisp-Function

Returns T if array has a leader. Otherwise, returns the symbol NIL. If array is not an array an ARGTYP error is signalled.

ARRAY-IN-BOUNDS-P array &REST subscripts [5] UCODE ENTRY
Returns T if the indices are in bounds for the dimensions of array. Otherwise, returns the symbol NIL.

ARRAY-LEADER array, index [640] MISC-OP Lisp-Function
Returns the array leader element of array specified by index. If array is not an array an ARGTYP error is signalled. If array does not have a leader the ARRAY-HAS-NO-LEADER error is signalled. If index is not a FIXNUM the ARGTYP error is signalled. Finally, if index is greater than or equal to the length of the leader the SUBSCRIPT-OOB error is signalled.

ARRAY-LEADER-LENGTH array [663] MISC-OP Lisp-Function
Returns the length of the array leader of array. If array is not an array the ARGTYP error is signalled.

ARRAY-LENGTH array [660] MISC-OP Lisp-Function
Returns the length of array. Does not take into account the fill pointer. Compare this macro-instruction with ARRAY-ACTIVE-LENGTH.

ARRAY-PUSH array, value [340] MISC-OP Lisp-Function
Add value as an element at the end of array. The fill pointer (leader element 0) is the index of the next element to be added. Returns NIL and doesn't update the fill pointer if array is full, otherwise returns the index of the element added. Does not automatically increase the size of the array like ARRAY-PUSH-EXTEND.

ARRAY-RANK array [664] MISC-OP Lisp-Function
Returns the rank or number of dimensions of array.

ARRAYP base-arg1 [311] MAIN-OP Lisp-Function
arg1 [600] MISC-OP Lisp-Function
Returns T if base-arg1 (or arg1) is an array (has DTP-ARRAY datatype). Otherwise the symbol NIL is returned.

AS-1 value, array, index [321] MISC-OP Lisp-Function
Stores value into the one-dimensional array array specified by index. Array must be an array and index must be a FIXNUM, otherwise the ARGTYP error is signalled. If index is less than zero or greater than the largest index permissible, then a SUBSCRIPT-OOB error is signalled. If array

is not one-dimensional, then ARRAY-NUMBER-DIMENSIONS is signalled. Returns value.

AS-1-FORCE value, array, index [324] MISC-OP Lisp-Functi
Stores value into the array array specified by index. Array is treated (forced) as one-dimensional; i.e., it it indexed with the single subscript index regardless of its rank. Returns value.

AS-2 value, array, sub1, sub2 [322] MISC-OP Lisp-Functi
Exactly like AS-1 except array is two-dimensional.

AS-2-REVERSE value, array, sub1, sub2 [325] MISC-OP Lisp-Functi
Stores value into the two-dimensional array array. See the Explorer Lisp Reference manual for a discussion of this instruction.

AS-3 value, array, sub1, sub2, sub3 [323] MISC-OP Lisp-Functi
Exactly like AS-1 except array is three-dimensional.

ASET value, array, &REST subscripts [0] UCODE ENTRY
Stores value into the array array specified by subscripts.

ASH n, nbits [5311] MISC-OP Lisp-Functi
Shift n arithmetically by nbits. N may be a DTP-FIXNUM or a DTP-EXTENDED-NUMBER.

ASSQ x, alist [610] MISC-OP Lisp-Functi
Search alist by comparing the CAR of each element for being EQ to x. Returns the CDR of the matching element if a match is found. Otherwise returns the symbol NIL.

ATOM x [564] MISC-OP Lisp-Functi
Returns the symbol NIL if x is a list. Otherwise, returns T.

BIGNUM-TO-ARRAY bignum, base [654] MISC-OP Lisp-Functi
Converts bignum into an array. Bignum is expressed in base and placed into an ART-Q array. The sign of bignum is ignored.

BIND loc, val [200] MISC-OP
Set the value of loc to val and save the old value of loc on the special binding stack.

BIND-CURRENT base-loc [154] MAIN-OP
Save the value of base-loc on the special binding stack. Any other stores into base-loc will not corrupt its previous value.

BIND-NIL base-loc [151] MAIN-OP
Set the value of base-loc to NIL and save the old value

on the special binding stack.

BIND-POP newval, base-loc [153] MAIN-OP
Set the value of base-loc to newval and save the old value on the special binding stack.

BIND-T base-loc [152] MAIN-OP
Set the value of base-loc to T and save the old value on the special binding stack.

BIT-VECTOR-P object [250] MISC-OP Lisp-Functic
Returns T if object is a bit vector, otherwise return NIL. A bit vector is defined to be an array of rank 1 whose elements are restricted to 0 and 1; i.e., a one-dimensional array of array type ART-1b.

BITBLT alu width from-array from-x from-y to-array to-x to-y [345] MISC-OP Lisp-Functic

BOUNDP symbol [240] MISC-OP Lisp-Functic
Returns T if the value cell of symbol is unbound. A cell is unbound if its data type is DTP-NULL.

BREAKPOINT [1] AUX-OP

C**R list [4,6] MISC-OP Lisp-Functic
Returns the C**R of list. Signals the ARGTYP error if list (or any required intermediary list) is not a list. The error check may be overridden if taking the CAR of a symbol or number is allowed. C**R may be CDAR or CAAR.

C***R list [11-17] MISC-OP Lisp-Functio
Returns the C***R of list. Signals the ARGTYP error if list (or any required intermediary list) is not a list. The error check may be overridden if taking the CAR of a symbol or number is allowed. C***R may be CAAAR, CAADR, CADAR, CDAAR, CDADR, CDDAR, or CDDDR.

C****R list [20-37] MISC-OP Lisp-Functio
Exactly like C***R. C****R may be CAAAAR, CAAADR, CAADAR, CAADDR, CADAAR, CADADR, CADDAR, CADDDR, CDAAAR, CDAADR, CDADAR, CDADDR, CDDAAR, CDDADR, CDDDAR, or CDDDDR.

CALL-* base-func [100, 104, 110, 114, 120, 124, 130] CALL-OP
Call the function base-func. * may be from 0 to 6 respectively, and represents the number of arguments for base-func that have already been pushed on the stack. Func may be any functional argument.

CALL-N n, base-func [134] CALL-OP
This is used to call functions with greater than 7 arguments.
Call the function base-func with n arguments.
The n arguments have already been pushed on the stack.
Base-func may be any functional argument.

CAR-SAFE object [620] MISC-OP Lisp-Function

CARCDR list [622] MISC-OP
Returns the cdr of list. Sets the indicators based
on CAR of list.

CDR-SAFE object [621] MISC-OP Lisp-Function

CEILING-1 dividend, divisor [551] MISC-OP
Returns dividend divided by divisor rounded up.
The remainder is not returned. To receive both
the quotient and the remainder see CEILING-2.

CEILING-2 dividend, divisor [555] MISC-OP
Returns dividend divided by divisor, rounded up,
and the remainder. Therefore, returns 2 values.

CHAR-INT character [704] MISC-OP Lisp-Function
Returns a FIXNUM whose value corresponds to
character.

CHARACTERP object [254] MISC-OP Lisp-Function
Returns T if object is a character, otherwise returns
NIL. Object is a character if it is defined with
a DTP-CHARACTER data type.

CLOSURE symbol-list, function [717] MISC-OP Lisp-Function
Returns a dynamic closure (DTP-CLOSURE),
closing function over the special variables
in symbol-list.

COMMON-AR-1 array, index [670] MISC-OP Lisp-Function
Returns the element of the one-dimensional array
array specified by index. Array must be a
one-dimensional array and index must be a FIXNUM.
If index is less than zero or greater than the
largest index permissible, then a SUBSCRIPT-OOB
error is signalled. If array is not
one-dimensional, then ARRAY-NUMBER-DIMENSIONS is signalled.
The type of result depends on the type of array.
This differs from AR-1 for array types string and fat-string.
AR-1 will return FIXNUMs for elements within these arrays
while the COMMON-LISP-AR-n instructions return characters.

COMMON-LISP-AR-1-FORCE array, index [673] MISC-OP Lisp-Function
Returns the element of the array array specified by

index. Array is treated (forced) as a one-dimensional array; i.e., it is indexed with a single subscript regardless of its actual rank. See COMMON-LISP-AR-1 for the difference between this and AR-1-FORCE.

COMMON-LISP-AR-2 array, sub1, sub2 [671] MISC-OP Lisp-Function
Exactly like COMMON-LISP-AR-1 except array must be two-dimensional.

COMMON-LISP-AR-3 array, sub1, sub2, sub3 [672] MISC-OP Lisp-Function
Exactly like COMMON-LISP-AR-1 except array must be three-dimensional.

COMMON-LISP-AREF array, &REST indices UCODE ENTRY
This is no longer a macro-instruction but rather a function.

COMMON-LISP-ELT sequence, index [626] MISC-OP Lisp-Function
Returns the element of sequence specified by index, which must be a positive FIXNUM or the ARGTYP error is signalled. Sequence may be a one-dimensional array (vector) or a list.

COMMON-LISP-LISTP object [243] MISC-OP Lisp-Function
The Common Lisp version of LISTP. Returns T if object is NIL or a cons, otherwise returns NIL. LISTP returns NIL if object is NIL.

COMPLEX-CALL callinfo, function [100] AUX-OP
Calls the function function with the supplied call-info word. If there are arguments for function then they are already on the stack. The complex call instructions basically enable the manipulation of the call-info word. Thus, they are used mainly to achieve Lexpr calls without providing the self-mapping table (there is another macro-instruction for Lexpr calls with self-mapping tables: LEXPR-FUNCALL-WITH-MAPPING-TABLE), and Multiple value return calls. See the section on function calling for more information.

COMPLEX-CALL-TO-* callinfo, function [100-103] AUX-OP
The four forms of COMPLEX-CALL. Calls the function function with the supplied call-info word. If there are arguments for function then they are already on the stack. * may be INDS, PUSH, RETURN, or TAIL-REC. See the section on calling for information about the four call destinations.

COMPLEXP object [260] MISC-OP Lisp-Function
Returns T if object is a complex number, otherwise returns NIL.

CONS car, cdr [412] MISC-OP Lisp-Functi
Constructs a list cell (cons) that has a CAR of car and a CDR of cdr. The storage is allocated in the default consing area specified by the variable DEFAULT-CONS-AREA.

CONS-IN-AREA car, cdr, area [413] MISC-OP Lisp-Functi
Exactly like CONS except the storage is allocated in the argument area. Signals the ARGTYP error if area is not a FIXNUM or a symbol with a FIXNUM in its value cell.

CONSP-OR-POP object [304] MISC-OP

COPY-ARRAY-CONTENTS from, to [342] MISC-OP Lisp-Functi
Copy all the elements from the array FROM to the array TO. If TO is longer than FROM, TO is filled with zeros (if a numeric array) or NILs. If either array is multidimensional, its elements are used in the order they are stored in memory.

COPY-ARRAY-CONTENTS-AND-LEADER from to [343] MISC-OP Lisp-Functi
Exactly like COPY-ARRAY-CONTENTS except the leader slots are also copied.

COPY-ARRAY-PORTION from-array from-start from-end to-array to-start to-end [344] MISC-OP Lisp-Functi
Exactly like COPY-ARRAY-CONTENTS except from-start and from-end are indices in from-array indicating the portion to copy. To-start and to-end are indices in to-array indicating where to place the elements.

DEPOSIT-FIELD value ppss fixnum [526] MISC-OP Lisp-Functi
Returns a number which in the byte ppss matches value and the rest matches fixnum. ppss is a field specifier as in LDB.

DISPATCH index base-disptable [72] MAIN-OP
Allows a multiway transfer of control. The address field is an immediate value used as a 9-bit offset from the beginning of the FEF; this is the beginning of a table of PC values. The table entry at the offset specified by the index argument becomes the new program counter value.

DPB value ppss num [524] MISC-OP Lisp-Functi
The inverse of LDB. The low order ss bits of value replace the field of the same size in num.

Never changes the sign of the quantity DPB'ed into. If pp is above the current size of num, the quantity is sign extended until it is long enough to accommodate the DPB. Returns either a FIXNUM or a BIGNUM.

Value and ppss must be FIXNUMs and ppss specify a field less than 25 bits. Num must be FIXNUM or a BIGNUM.

ENDP object [255] MISC-OP Lisp-Functio
Returns T if object is NIL. Returns NIL if object is a cons (DTP-LIST). Otherwise, signals the ARGTYP error.

EQ x, base-y [23] MAIN-OP Lisp-Functio
EQ x, y [574] MISC-OP
Returns T if x and base-y are the exact same Lisp object, otherwise returns NIL.

EQ-IMMED x, immed-y [3] IMMEDIATE
Exactly like EQ except immed-y is taken from the instruction itself.

EQ-T object [261] MISC-OP
Returns T if object is the Lisp object T, otherwise returns NIL.

EQL x, base-y [24] MAIN-OP Lisp-Functio
EQL x, y [573] MISC-OP Lisp-Functio
See the Explorer Lisp Reference manual.

EQUAL x, base-y [25] MAIN-OP Lisp-Functio
EQUAL x, y [562] MISC-OP
Returns T if x and base-y are EQ. If x and base-y are numbers then they are equal if they have the same value and type. Two conses are equal if the CARs are equal and their CDRs are equal. Two strings are equal if they have the same length, and the characters composing them are the same. All other objects are equal if and only if they are EQ.

EQUALP x, base-y [26] MAIN-OP Lisp-Functio
EQUALP x, y [575] MISC-OP
See the Explorer Lisp Reference manual.

EXCHANGE [10] AUX-OP
Swaps the top two items on the stack.
Accepts and returns no values.

EXPT base, exponent [536] MISC-OP Lisp-Functio
Returns base raised to the power exponent.

FBOUNDP symbol [237] MISC-OP lisp
Returns T if the function cell of symbol does not

contain the unbound marker, otherwise return NIL.
Symbol must be a symbol or the ARGTYP error is
signalled.

FIND-POSITION-IN-LIST element, list [616] MISC-OP Lisp-Function
Returns the numeric index in list at which element
is found (uses EQ), unlike MEMQ which may return
the rest of list beginning with element. Otherwise
returns NIL. The index returned is zero-based.
List must be a list or the ARGTYP error is signalled.

ZLC:FIX number [510] MISC-OP Lisp-Function
See the Explorer Lisp Reference manual.

FIXNUMP base-object [34] MAIN-OP Lisp-Function
object [602] MISC-OP Lisp-Function
Returns T if object is a FIXNUM, otherwise return NIL.

FIXP x [561] MISC-OP
Returns T if x is an integer, otherwise returns
NIL disregarding the datatype of x.

FLOAT-EXPONENT flonum [516] MISC-OP Lisp-Function
Returns the exponent of flonum as a FIXNUM.

FLOAT-FRACTION flonum [517] MISC-OP Lisp-Function
Returns flonum modified to contain 0 as its exponent.
The result is either zero or has absolute value at
least 1/2 and less than one.

FLOATP x [230] MISC-OP Lisp-Function
Returns T if x is a floating point number,
otherwise returns NIL.

FLOOR-1 dividend, divisor [550] MISC-OP
Returns dividend divided by divisor, rounded down.

FLOOR-2 dividend, divisor [551] MISC-OP
Returns dividend divided by divisor, rounded down,
and the remainder. Therefore, this instruction
returns 2 values.

FUNCTION-CELL-LOCATION symbol [633] MISC-OP Lisp-Function
Returns a locative to symbol's function cell.
Symbol must be a symbol or the ARGTYP error is
signalled.

GCD num1 num2 [543] MISC-OP
Returns the GCD (Greatest Common Divisor) of num1
and num2. Therefore, num1 or num2 will be returned.
Both arguments must be numbers or the ARGTYP error is
signalled.

G-L-P array [653] MISC-OP Lisp-Function
Returns a list with the contents of array. Array must be an array of type ART-Q-LIST or the ARGTYP error is signalled.

GET-LEXICAL-VALUE-CELL env-list, symbol-cell-location [722] MISC-OP Lisp-Function

GET-LOCATION-OR-NIL symbol property [72] MISC-OP Lisp-Function
Returns a locative to the plist location containing the value of property. Symbol can be a symbol, instance, or disembodied property list. If property is not found, returns NIL.

GETL symbol indicator-list [71] MISC-OP Lisp-Function
Find any of the properties in indicator-list, on symbol. Whichever of those properties occurs first in the property list is used. The value is a pointer to the cell in the property list that points to the indicator. The CADR of the value is the property's value.

HALT [3] AUX-OP
Halts the processor. This is used by hardware debugging to possibly evaluate the state of the machine. This should not normally be used to shutdown the system, rather %CRASH should be used.

HAULONG integer [515] MISC-OP Lisp-Function
Returns the number of bits in integer as a FIXNUM. For example, the size of #o777 is nine. Integer may be either a FIXNUM or a BIGNUM.

INT-CHAR fixnum [703] MISC-OP Lisp-Function
Returns a character whose value corresponds to fixnum.

INTEGERP base-object [35] MAIN-OP Lisp-Function
Returns T if base-object is an integer, otherwise returns NIL for other numbers and non-numbers.

INTERNAL-CHAR-EQUAL ch1 ch2 [232] MISC-OP
Compares two characters that are either FIXNUMs or DTP-CHARACTER objects. If the character codes are equal then T is returned. Otherwise, if ALPHABETIC-CASE-AFFECTS-STRING-COMPARISON is non-NIL, NIL is returned. If the characters are different and case doesn't matter, T is returned if one is the alphabetic uppercase of the other, otherwise NIL is returned. (NOTE: notice how those macro-instructions with the prefix INTERNAL- are not defined with the Lisp-Function level.)

INTERNAL-FLOAT number [512] MISC-OP
Returns number if number is already a floating-point. Otherwise, number is converted to a single-float number and returned. This is like ZLC:FLOAT.

INTERNAL-GET-2 symbol property [70] MISC-OP
Returns symbol's property property, otherwise NIL is returned. Returns NIL if symbol does not have a property list or is of a type that does not have properties. Symbol may be a symbol, list, locative, or an instance.

INTERNAL-GET-3 symbol property default [73] MISC-OP
Similar to INTERNAL-GET-2 except returns default instead of NIL if property is not found.

LAST list [611] MISC-OP Lisp-Functio
Returns the last cons cell of list. Signals the ARGTYP error if list is not of the type list.

LDB ppss, num [520] MISC-OP Lisp-Functio
Returns the ss number of bits starting at bit pp.

LDB-IMMED num, immed-ppss [44] IMMEDIATE
Exactly like LDB except immed-ppss is taken directly from the instruction opcode. Immed<8:4> is the 5 bit pp (position) and Immed<3:0> is the 4 bit ss (length).

LENGTH list-or-array [612] MISC-OP Lisp-Functio
Returns the number of elements in list-or-array if list-or-array is a list. Returns the array active length if list-or-array is an array. The array's active length is the value of its fill-pointer if it exists or the number of elements in the array.

LENGTH-GREATERP list-or-array value [231] MISC-OP Lisp-Functio
Tests whether list-or-array has more than the number of elements indicated by value without using LENGTH, and thus without going any farther down the list than necessary. Returns T if the number of elements is greater than value, otherwise returns NIL.

LEXICAL-UNSHARE [76] MAIN-OP
LEXICAL-UNSHARE-ALL [31] AUX-OP
See the Closures section.

LEXPR-FUNCALL-WITH-MAPPING-TABLE arglist, mapping-table, fn
[114] AUX-OP
Lexpr-funcalls the function fn with the supplied mapping-table and one argument, arglist.

See the section on function calling for more information concerning Lexpr-funcalls.

LEXPR-FUNCALL-WITH-MAPPING-TABLE-TO-*
arglist mapping-table fn [114-117] AUX-OP
The four forms of LEXPR-FUNCALL-WITH-MAPPING-TABLE. * may be INDS, PUSH, RETURN, or TAIL-REC, respectively. LEXPR-FUNCALL-WITH-MAPPING-TABLE-TO-INDS is the same as LEXPR-FUNCALL-WITH-MAPPING-TABLE.

LIST &REST elements [10] UCODE ENTRY
See the Explorer Lisp Reference manual.

LIST* first, &REST elements [11] UCODE ENTRY
See the Explorer Lisp Reference manual.

LIST*-IN-AREA area first &REST elements [13] UCODE ENTRY
See the Explorer Lisp Reference manual.

LIST-IN-AREA area, &REST elements [10] UCODE ENTRY
See the Explorer Lisp Reference manual.

LISTP x [32] MAIN-OP Lisp-Functio
LISTP x [576] MISC-OP
Returns T if x is a list or NIL, otherwise returns NIL. Note that this is the Common Lisp LISTP.

LOAD-FROM-HIGHER-CONTEXT contex-desc [720] MISC-OP

LOCATE-IN-HIGHER-CONTEXT contex-desc [721] MISC-OP
See Closures section.

LOCATE-IN-INSTANCE instance, symbol [710] MISC-OP Lisp-Functio
Returns a locative to the instance variable symbol of instance.

LOCATE-LEXICAL-ENVIRONMENT env-num [77] MAIN-OP
See Closures section.

LONG-BR [176] AUX-OP
Unconditional macrocode branch. Instead of a signed relative displacement like short-branches, the second halfword of the instruction contains the new PC offset relative to the start of the FEF (as opposed to relative from the current instruction).

LONG-BR-* [160-175] AUX-OP
Conditional macrocode branch. See LONG-BR for new PC generation scheme. * may be NULL-ELSE-POP, NOT-NULL-ELSE-POP, NULL, NOT-NULL, ATOM, NOT-ATOM, ZEROP, NOT-ZEROP, SYMBOLP, NOT-SYMBOLP, NULL-LIKELY or NOT-NULL-LIKELY.
See section on short branches for a description

of the different conditions.

LONG-PUSHJ [177] AUX-OP
Unconditional subroutine call. The destination of the call is computed like in LONG-BR. The macrocode subroutine will return POPJ.

LSH n, nbits [530] MISC-OP Lisp-Function
Return n logically shifted (zero fill) by nbits. The sign of nbits controls the direction of the shift. If nbits is negative, the shift is to the right. n and nbits must both be FIXNUMs or the ARGTYP error is signalled.

MAKE-EPHEMERAL-LEXICAL-CLOSURE envdesc, function [724] MISC-OP
MAKE-LEXICAL-CLOSURE envdesc, function [723] MISC-OP
See Closures section

MASK-FIELD ppss, fixnum [522] MISC-OP Lisp-Function
Returns fixnum with all but the ppss byte replaced with zeros.

MEMQ x, list [613] MISC-OP Lisp-Function
Returns the sublist of list beginning with the first occurrence of x (using EQ), otherwise return NIL.

MINUS number [514] MISC-OP Lisp-Function
Returns the negative of number. Number must be of type number of the ARGTYP error is signalled.

MINUSP base-x [37] MAIN-OP Lisp-Function
MINUSP number [567] MISC-OP
Returns T if base-x is negative (strictly less than zero), otherwise returns NIL. The ARGTYP error is signalled if number is not of type Number.

MOVEM base-loc [141] MAIN-OP
Copies the value at the top of stack to base-loc. Does not pop the top of stack nor does it return a value.

NAMED-STRUCTURE-P object [603] MISC-OP Lisp-Function
Returns the symbol name of object if it is a named-structure array, otherwise return NIL. See the section on internal storage for a description of named-structures and the location of their name. This instruction will also return NIL if the name of object is not a symbol or a closure.

NCONS car [410] MISC-OP Lisp-Function
Constructs a cons (list cell) that has a CAR of car and a CDR of NIL. The storage is allocated in the

area specified by the variable DEFAULT-CONS-AREA.

NCONS-IN-AREA car, area [411] MISC-OP Lisp-Function
Exactly like NCONS except the storage is allocated in area. area must be a FIXNUM or a symbol that evaluates to a FIXNUM or the ARGTYP error is signalled.

NLISTP x [235] MISC-OP Lisp-Function
Returns T if x is an atom, otherwise returns NIL. This instruction will return NIL for NIL.

NOT x [563] MISC-OP Lisp-Function
Returns T if x is NIL, otherwise returns NIL.

NOT-INDICATORS [77] MISC-OP
Returns T if and only if the indicators are null, otherwise returns NIL.

NSYMBOLP x [236] MISC-OP Lisp-Function
Returns T if x is not a symbol, otherwise returns NIL.

NTH n, list [614] MISC-OP Lisp-Function
Returns the nth element of list (0-origin).

NTHCDR n, list [615] MISC-OP Lisp-Function
Returns list with the first n elements discarded. This is the equivalent of performing the CDR n times.

NUMBERP x [30] MAIN-OP Lisp-Function
NUMBERP x [565] MISC-OP
Returns T if x is of type Number, otherwise returns NIL.

PDL-WORD n [40] MISC-OP
Returns the value on the PDL (cached stack) that is n below the current PDL-pointer. This assumes that the selected word is on the PDL (cached stack) and never fetches from memory.

PLUSP x [36] MAIN-OP Lisp-Function
PLUSP x [566] MISC-OP
Returns T if x is positive (strictly greater than zero), otherwise returns NIL. The ARGTYP error is signalled if x is not of type Number.

POP obj, base-loc [140] MAIN-OP
Removes (pops) obj from the top of stack to base-loc. Nothing is returned.

POP-M-FROM-UNDER-N num-pops, num-to-keep
[13] AUX-OP
While keeping the top num-to-keep values on the stack,

removes the num-pops values from underneath them.

POP-PDL-* [500-517] AUX-OP
Removes the top * elements from the PDL (stack cache).
* may be from 1 to 63.

POPJ return-macropc [14] AUX-OP
Transfer macrocontrol to return-pc. return-pc
is relative to the beginning of the FEF.
Therefore, the next macro-instruction executed will
be at return-pc.

PREDICATE [76] MISC-OP
Returns NIL if and only if the indicators are
null, otherwise return T.

PROPERTY-CELL-LOCATION symbol [634] MISC-OP Lisp-Functio
Returns a locative to symbol's property-list cell.
If symbol is not a symbol then the ARGTYP error
will be signalled.

PUSH base-obj [50] MAIN-OP
Pushes the value at base-obj to the
top of stack.

PUSH-AR-1 index, base-array [67] MAIN-OP
Pushes the value at the specified
index into the array at base-array.

PUSH-AREFI array, immed-index [47] AREFI
Returns the value of array specified by immed-index.
This is used for single-dimension array references
where immed-index is a small constant.
immed-index is an immediate value in the
instruction and the array is popped off the
stack.

PUSH-CADDR base-list [55] MAIN-OP
Pushes the CADDR of the list at base-list.

PUSH-CADR base-list [53] MAIN-OP
Pushes the CADR of the list at base-list.

PUSH-CAR base-list [51] MAIN-OP
Pushes the CAR of the list at base-list.

PUSH-CDDR base-list [54] MAIN-OP
Pushes the CDDR of the list at base-list.

PUSH-CDR base-list [52] MAIN-OP
Pushes the CDR of the list at base-list.

PUSH-CDR-STORE-CAR-IF-CONS x,base-dest [147] MAIN-OP

Returns the CDR of x (popped from stack)
and stores its CAR at base-dest if x is a cons,
otherwise NIL is left in the indicators.
This is used mainly to implement DOLIST.

PUSH-CONS car, base-cdr [56] MAIN-OP
Pushes the cons of car and the cdr at
base-cdr. Car is popped from the stack.

PUSH-GET sym, base-ind [57] MAIN-OP
Pushes sym's base-ind property, otherwise NIL
is returned. Returns NIL if sym does not have a
property list or is of a type that does not have
properties. Sym may be a symbol, list, locative,
or an instance. This instruction corresponds to
the GET function with two arguments.

PUSH-LOC base-loc [150] MAIN-OP
Pushes a locative that points to base-loc.

PUSH-LONG-FEF base-x [70] MAIN-OP
Pushes what is at base-x plus the current
FEF base address. base-x is a 9-bit offset.
The current FEF base address is defined to be the
function object of the currently executing stack
frame. The read of the object at base-x plus FEF
will be transported. This reference, therefore,
can cause a TRANS-TRAP error to be signalled if
the word read contains an invalid object or is
unbound.

PUSH-NEG-NUMBER [46] MAIN-OP
PUSH-NUMBER [47] MAIN-OP
Pushes the specified thing onto the stack. Knowing
what the thing is means greater efficiency because
these things never have to be TRANSPORTed.
These take an immediate operand which is used as
a 9-bit integer.

RATIONALP x [256] MISC-OP Lisp-Function
Returns T if x is a ratio or an integer,
otherwise returns NIL for all other data types.
An error is never signalled.

RATIOP x [257] MISC-OP Lisp-Function
Returns T if x is a ration, otherwise return NIL.
An error is never signalled.

RETURN base-val [17] MAIN-OP
Causes the current call frame to fold and a
return to the previous call frame. The value
at base-val will be left on top of the stack,
which is exactly where the current (function

executing RETURN) call frame started.

RETURN-* val0, val1, ..., val* [600-637] AUX-OP
Causes the current call frame to fold and a return to the previous call frame. *-values will be left on top of the stack, which is exactly where the current (function executing RETURN-*) call frame started. Therefore, this instruction moves the top * values to the new top of stack. Actually no more values are returned than the caller is expecting so the number returned may be less. * may be from 0 to 63 (decimal).

RETURN-LIST list [121] AUX-OP
Causes the current call frame to fold and a return to the previous call frame. Returns the elements of list as multiple values. This instruction will return the exact number of elements that the caller expects, i.e., if the number of elements in list is less than the number the caller expected to be returned, then NILs are used to pad. Conversely, the caller never receives more values than it is expecting.

RETURN-N val1 val2 ... valnumvals numvals [120] AUX-OP
Exactly like RETURN-* except the number of values to return is specified with numvals. Again, the caller will never receive more values than it is expecting so the number returned may be less than numvals.

RETURN-NIL [122] AUX-OP
Causes the current call frame to fold and a return to the previous call frame. The value NIL will be left on top of the stack, which is exactly where the current (function executing RETURN-NIL) call frame started.

RETURN-NOT-INDS [137] AUX-OP
Causes the current call frame to fold and a return to the previous call frame. The value left on top of the stack, which is exactly where the current (function executing RETURN-NOT-INDS) call frame started will be T if the NIL indicator is not set. Otherwise NIL is returned.

RETURN-PRED [136] AUX-OP
Exactly like RETURN-NOT-INDS except returns T if the NIL indicator is set, otherwise returns NIL.

RETURN-T [123] AUX-OP
Causes the current call frame to fold and a return to the previous call frame. The value T will be left on top of the stack, which is exactly where the current (function executing RETURN-T) call frame started.

ROT n, nbits [532] MISC-OP Lisp-Function
Returns n rotated by nbits. The direction of rotation is controlled by the sign of nbits. If nbits is positive then a left shift is performed, otherwise a right shift is performed. n is treated as a 25-bit number and n and nbits must both be FIXNUMs.

ROUND-1 dividend, divisor [553] MISC-OP
Returns dividend divided by divisor, rounded to the nearest integer. The remainder is not returned.

ROUND-2 dividend, divisor [557] MISC-OP
Returns dividend divided by divisor, rounded to the nearest integer and the remainder.

RPLACA cons, newcar [300] MISC-OP Lisp-Function
Returns cons with its CAR replaced with newcar. cons must be a list or a locative or the ARGTYP error is signalled.

RPLACD cons, newcdr [301] MISC-OP Lisp-Function
Returns cons with its CDR replaced with newcdr. cons must be a list or the ARGTYP error is signalled.

SCALE-FLOAT flonum, integer [545] MISC-OP Lisp-Function
Returns flonum with integer added to its exponent.

SELECT x, selectq-table [71] MAIN-OP
Used to implement multi-way branches based on value of x. Branches are defined by selectq-table which has the format shown below. It is a cdr-coded list of objects or EVCPs to objects (cdr coding must be correct).

```
+-----------------------+
| FIX : Offset to       |
|       Dispatch Table  |
|     : <item>          | 1st compare value, CDR-NEXT
|       ...             |
|     : <item>          | Last compare value, CDR-NIL
| FIX : Max item num    |<----- Dispatch table only
| FIX : Else PC         |       from here down
| FIX : PC of item 0    |
| FIX : PC of item 1    |
```

```
        | FIX : PC of item 2    |
                  ...
        | FIX : PC of item n    |
        +-----------------------+
```

SET symbol, value [310] MISC-OP Lisp-Functio
Sets the value cell of symbol to value. Signals the ARGTYP error if symbol is not a symbol or is NIL (since it is illegal to set NIL).

SET-XINSTANCE-REF instance, index, value [354] MISC-OP
Sets the slot at index into the instance data structure to value.

SET-AR-1 array, subscript, value [331] MISC-OP Lisp-Functio
Sets the element of the one-dimensional array array specified by subscript to value. Array must be a one-dimensional array and subscript must be a FIXNUM. If index is less than zero or greater than the largest index permissible then a SUBSCRIPT-OOB error is signalled. If array is not one-dimensional then ARRAY-NUMBER-DIMENSIONS is signalled.

SET-AR-1-FORCE array, subscript, value [335] MISC-OP Lisp-Functio
Sets the element of the array array specified by subscript to value. Array is treated (forced) as a one-dimensional array, i.e., it is indexed with a single subscript regardless of its actual rank.

SET-AR-2 array, subscript1, subscript2, value [332] MISC-OP Lisp-Functio
Exactly like SET-AR-1 except array must be two-dimensional.

SET-AR-3 array subscript1 subscript2 subscript3 value [334] MISC-OP Lisp-Functio
Exactly like SET-AR-1 except array must be three-dimensional.

SET-AREF array, &REST subscripts-and-values UCODE ENTRY

SET-ARRAY-LEADER array index value [330] MISC-OP Lisp-Functio
Sets the array leader element of array specified by index to value. If array is not an array an ARGTYP error is signalled. If array does not have a leader the ARRAY-HAS-NO-LEADER error is signalled. If index is not a FIXNUM the ARGTYP error is signalled. Finally, if index is greater than or equal to the length of the leader the SUBSCRIPT-OOB error is signalled.

SET-NIL base-loc [155] MAIN-OP
Sets the value at base-loc to NIL.

SET-T base-loc [156] MAIN-OP
Sets the value at base-loc to T.

SET-ZERO base-loc [157] MAIN-OP
Sets the value at base-loc to a FIXNUM zero.

SETCAR cons, newcar [302] MISC-OP Lisp-Functio
Replaces the CAR of cons with newcar and returns newcar.
Compare with RPLACA which returns a different value.

SETCDR cons, newcdr [303] MISC-OP Lisp-Functio
Replaces the CDR of cons with newcdr and returns newcdr.
Compare with RPLACD which returns a different value.

SETE-1+ base-loc [144] MAIN-OP
Increments the contents at base-loc and stores the result back at base-loc. This is called a read-modify-write instruction. The contents at base-loc must be of the numeric type or an ARGTYP error is signalled.

SETE-1- base-loc [145] MAIN-OP
Exactly like SETE-1- except the contents at base-loc is decremented.

SETE-CDDR base-loc [143] MAIN-OP
Replaces base-loc with the CDDR of base-loc. This is called a read-modify-write instruction. (setq frob (cddr frob)) translates to this instruction. The contents at base-loc and its CDR must be of the list type or the ARGTYP error is signalled.

SETE-CDR base-loc [142] MAIN-OP
Replaces base-loc with the CDR of base-loc. This is called a read-modify-write instruction. (setq frob (cdr frob)) translates to this instruction. The contents at base-loc must be of the list type or the ARGTYP error is signalled.

SETELT sequence, index, value [307] MISC-OP Lisp-Functio
Sets the element of sequence at index to value. This corresponds to SETF on ELT.

SHRINK-PDL-SAVE-TOP n-slots value-to-move
[41] MISC-OP
Pops n-slots (after popping the 2 arguments above) from the PDL and moves value-to-move to the new top of the stack.

SIMPLE-ARRAY-P object [246] MISC-OP Lisp-Functio
Returns T if object is a simple array, otherwise

returns NIL. A simple array is an array that does not have a fill-pointer and is not displaced or indirect. This is a Common Lisp instruction.

SIMPLE-BIT-VECTOR-P object [251] MISC-OP Lisp-Function
Returns T if object is a simple bit vector, otherwise returns NIL. A simple bit vector is a one-dimensional array of array type ART-1B with no fill-pointer that is also not displaced or indirect. This is a Common Lisp instruction.

SIMPLE-STRING-P object [247] MISC-OP Lisp-Function
Returns T if object is a simple string, otherwise returns NIL. A simple string is a one-dimensional array of array type ART-STRING or ART-FAT-STRING with no fill-pointer that is also not displaced or indirect. This is a Common Lisp instruction.

SIMPLE-VECTOR-P object [245] MISC-OP Lisp-Function
Returns T if object is a simple vector, otherwise returns NIL. A simple vector is a one-dimensional numeric array with no fill-pointer that is also not displaced or indirect. This is a Common Lisp instruction.

SMALL-FLOAT number [511] MISC-OP Lisp-Function
Converts number to a short float.

SMALL-FLOATP object [253] MISC-OP Lisp-Function
Returns T if object is of type SHORT-FLOAT (small-float), otherwise returns NIL. (Small-float is Zetalisp terminology, while short-float is the Common Lisp type name.)

SPECIAL-PDL-INDEX [405] MISC-OP Lisp-Function
Returns a locative pointing to the last slot of the current special PDL that was bound.

STACK-GROUP-RESUME sg, x [47] MISC-OP Lisp-Function
Resumes the stack group sg with the argument x. See the stack groups section.

STACK-GROUP-RETURN x [46] MISC-OP Lisp-Function
Resume the stack group which invoked the current stack group with the argument x. The resumed stack group's resumer does not change.

STORE-ARRAY-LEADER value array index [320] MISC-OP Lisp-Function
Sets the array leader element of array specified by index to value. If array is not an array, an ARGTYP error is signalled. If array does not have a leader the ARRAY-HAS-NO-LEADER error is signalled. If index is not a FIXNUM, the ARGTYP error is signalled. Finally, if index is greater than or

equal to the length of the leader, the SUBSCRIPT-OOB error is signalled.

STORE-IN-HIGHER-CONTEXT value context-desc [30] AUX-OP
Used for storing into lexical variables in a higher lexical context.

STRINGP base-x [33] MAIN-OP Lisp-Function
x [601] MISC-OP Lisp-Function
The STRINGP MAIN-OP sets the indicators if base-x is a one-dimensional array of array-type ART-STRING or ART-FAT-STRING. The STRINGP MISC-OP returns T if x is a one-dimensional array of array-type ART-STRING or ART-FAT-STRING, otherwise return NIL.

SYMBOL-FUNCTION symbol [627] MISC-OP Lisp-Function
This is exactly like FSYMEVAL (same MISC-OP number, also). The difference is that the functionality is known in the Lisp world under this name.

SYMBOL-NAME symbol [631] MISC-OP Lisp-Function
Returns the print name of symbol (an array pointer). Symbol must be a symbol or the ARGTYP error is signalled. This was called GET-PNAME in Zetalisp.

SYMBOL-PACKAGE symbol [635] MISC-OP Lisp-Function
Returns the package object of symbol. Symbol must be a symbol or the ARGTYP error is signalled.

SYMBOL-VALUE symbol [636] MISC-OP Lisp-Function
Returns the current value of symbol. Symbol must be a symbol or the ARGTYP error is signalled. If symbol is unbound then the TRANS-TRAP error is signalled. This was called SYMEVAL in Zetalisp.

SYMBOLP x [577] MISC-OP Lisp-Function
Returns T if x is a symbol, otherwise NIL is returned.

TEST base-obj [10] MAIN-OP
Sets the indicators based on the contents of base-obj.

TEST-AREFI [7] AREF1
Sets the indicators based on the contents of an immediate array element reference.

TEST-CAAR base-list [15] MAIN-OP
Sets the indicators based on the contents of the CAAR of base-list.

TEST-CADR base-list [13] MAIN-OP
Sets the indicators based on the contents of the CADR of base-list.

TEST-CAR base-list [11] MAIN-OP
Sets the indicators based on the contents of the CAR of base-list.

TEST-CDDR base-list [14] MAIN-OP
Sets the indicators based on the contents of the CDDR of base-list.

TEST-CDR base-list [12] MAIN-OP
Sets the indicators based on the contents of the CDR of base-list.

TEST-MEMQ x, base-list [16] MAIN-OP
Sets the indicators based on either the sublist of base-list beginning with the first occurrence of x (using EQ), if found, or NIL.

TIME-IN-60THS [472] MISC-OP Lisp-Functio
Used internally to implement TIME when called with no arguments.

TRUNCATE-1 dividend, divisor [552] MISC-OP
Returns dividend divided by divisor, rounded down to zero (truncated). The remainder is not returned.

TRUNCATE-2 dividend, divisor [556] MISC-OP
Returns dividend divided by divisor, rounded down to zero (truncated) and the remainder.

TYPEP-STRUCTURE-OR-FLAVOR object, type [252] MISC-OP Lisp-Functio
Used for TYPEP when the Compiler knows that the type being tested for is a flavor or named structure.

UNBIND-* [400-417] AUX-OP
Undo * bindings on the special PDL (binding stack). * may be from 1 to 63.

UNBIND-TO-INDEX special-pdl-index [17] AUX-OP
Undo bindings on the special PDL (binding stack) until the special PDL pointer is less than or equal to special-pdl-index.

UNBIND-TO-INDEX-MOVE [406] MISC-OP
special-pdl-index, value-to-move
Exactly like UNBIND-TO-INDEX, except value-to-move is returned.

VALUE-CELL-LOCATION symbol [632] MISC-OP Lisp-Functio
Returns a locative to the internal value cell of

symbol. Symbol must be a symbol or the ARGTYP error is signalled. Note that this instruction will return a pointer to the exact contents of the value cell and will not follow forwarding pointers.

VECTOR-PUSH new-element, vector [341] MISC-OP Lisp-Functio
Returns the new fill-pointer for vector after pushing new-element, otherwise returns NIL if vector is full. Vector must have a leader or the ARRAY-HAS-NO-LEADER error is signalled. The fill-pointer of vector (leader element 0) must be a FIXNUM or the FILL-POINTER-NOT-FIXNUM error is signalled. This instruction does not check that vector is a true vector. This is a Common Lisp instruction.

VECTORP object [244] MISC-OP Lisp-Functio
Returns T if object is a vector, otherwise returns NIL. A vector is a one-dimensional array. This is a Common Lisp instruction.

ZEROP number [560] MISC-OP Lisp-Functio
Returns T if number is equal to the zero of its type, otherwise returns NIL. Number must be of type Number (numeric) or the ARGTYP error is signalled.

%ADD-INTERRUPT device-desc, level [211] MISC-OP
Installs an interrupt for the device described by the array device-desc at the level level. The device encoded within device-desc must have a microcode interrupt handler for it or an error is signalled. This instruction does not perform any device initializations.

%ALLOCATE-AND-INITIALIZE data-type header-type header second-word area size [415] MISC-OP Lisp-Functio
This is the subprimitive for creating most structured-type objects. Area is the area in which it is to be created, as a FIXNUM or a symbol. Size is the number of words to be allocated. The value returned points to the first word allocated and is of type data-type. The words allocated are initialized with interrupts disallowed so that storage conventions are preserved at all times. The first word, the header, is initialized to have header-type in its data-type field and header in its pointer field. The second word is initialized to second-word. The remaining words are initialized to NIL. The cdr-codes of all words except the last are set to cdr-next; the cdr-code

of the last word is set to cdr-nil. Note that programs should not rely on the cdr-code field of non-cons cells being in a known state.

%ALLOCATE-AND-INITIALIZE-ARRAY header index-length leader-length area nqs
[416] MISC-OP Lisp-Function

This is the subprimitive for creating arrays, called only by make-array. It is different from %allocate-and-initialize because arrays have a more complicated header structure.

%ALLOCATE-AND-INITIALIZE-INSTANCE header, area, nqs
[420] MISC-OP Lisp-Function

Allocates storage for an instance, sets header type to DTP-Instance-Header and sets data type to DTP-Instance. Fills allocated space with NIL and places header in word 0.

%AREA-NUMBER x [500] MISC-OP Lisp-Function

Returns the area number of the area the pointer x points into, or NIL.

%ASSURE-PDL-ROOM room [12] AUX-OP Lisp-Function

This instruction will trap if there are not room more words available in this function frame.

%BLT from-address, to-address, count, increment
[346] MISC-OP Lisp-Function

Copy a block of virtual memory, a word at a time, with no decoding, for untyped data. Use %BLT-TYPED for words which contain Lisp data types. The first word is copied from from-address to to-address. Increment is added to each address and then another word is copied till count is exhausted.

%BLT-FROM-PHYSICAL source-address destination-address, number-of-words increment
[224] MISC-OP Lisp-Function

Copy a block of physical memory from source-address to unboxed virtual memory starting at destination-address. Not decoded; use on untyped data only.

%BLT-TO-PHYSICAL source-address destination-address number-of-words increment
[223] MISC-OP Lisp-Function

Copy a block of physical memory from unboxed virtual memory starting at source-address to physical memory beginning at destination-address. Not decoded; use on untyped data only.

%BLT-TYPED [347] MISC-OP Lisp-Function

Copy a block of virtual memory, a word at a time.

The first word is copied from from-address to to-address. Increment is added to each address and then another word is copied till count is exhausted. Each word copied is transported and each word written is checked through the the Write Barrier. Returns NIL.

%CHANGE-PAGE-STATUS virt-addr swap-status access-and-meta [360] MISC-OP Lisp-Function
Changes the page status bits of the page containing virt-addr, if it is paged in, to swap-status and access-and-meta. If either swap-status or access-and-meta are NIL then that parameter is not set. Returns T if the page was found in the page hash table (it was swapped in) or NIL if it was not found (it was not swapped in). This does no error checking.

%CLOSE-CATCH [134] AUX-OP
Close a catch.

%CLOSE-CATCH-UNWIND-PROTECT [135] AUX-OP
Close catch but leave info for %unwind-protect-continue.

%COMPUTE-PAGE-HASH addr [475] MISC-OP Lisp-Function
Computes the page hash table index that corresponds to addr and returns it as a FIXNUM.

%CRASH code, object, paws-up-p [3] AUX-OP Lisp-Function
Causes machine to crash (like ILLOP) indicating crash reason as software with code remembered as the crash code. Object is also remembered for crash analysis. If paws-up-p is not NIL then will display paws-up as ILLOP does. If paws-up-p is NIL then it is presumed that the called has indicated the lossage to the user. %CRASH is not restartable, but you may be able to warm-boot out of it.

%CREATE-PHYSICAL-PAGE pfn [140] AUX-OP Lisp-Function
Adds the physical page specified by pfn to the pool of available page frames. See section on Paging and Disk Management for more details.

%DATA-TYPE x [450] MISC-OP Lisp-Function
Returns the data type of x as a FIXNUM. Its value will be less than 32.

%DELETE-PHYSICAL-PAGE pfn [362] MISC-OP Lisp-Function
Deletes the physical page specified by pfn from the page frame pool. Returns T if successful, otherwise NIL. See section on Paging and Disk Management for more details.

%DISK-RESTORE [156] AUX-OP Lisp-Functio
partition-high-16-bits low-16-bits physical-unit
Restores the load partition whose name is formed by concatenating the first two arguments to form a 32-bit number (4 characters). If the number is zero then the default band is restored from the default unit, else the named partition is restored from physical-unit.

%DIV dividend, divisor [544] MISC-OP Lisp-Functio
Returns the rational number generated by dividing dividend by divisor.

%DRAW-CHAR font-array char-code x-bitpos y-bitpos alu-function sheet
[0] MODULE(TV)
Draws the character char-code of font font-array on sheet using alu-function. alu-function is typically TV:ALU-IOR, TV:ALU-ANDCA, or TV:ALU-XOR. x-bitpos and y-bitpos are the position in sheet for the upper left corner of the character to be drawn.

%DRAW-FILLED-RASTER-LINE x1 x2 y left-edge top-edge right-edge bottom-edge alu draw-last-point fill-color destination
[5] MODULE(TV)

%DRAW-FILLED-TRIANGLE x1 y1 x2 y2 x3 y3 left-edge top-edge right-edge bottom-edge alu draw-third-edge draw-second-edge draw-first-edge fill-color destination
[4] MODULE(TV)

%DRAW-LINE x0 y0 x y alu draw-end-point sheet
[2] MODULE(TV)
Draws a straight line from (x0,y0) to (x,y) on sheet using alu as the ALU function. alu is typically TV:ALU-IOR, TV:ALU-ANDCA, or TV:ALU-XOR.

%DRAW-RECTANGLE width height x-bitpos y-bitpos alu-function sheet
[1] MODULE(TV)
Draws a solid rectangle on sheet using alu-function. alu is typically TV:ALU-IOR, TV:ALU-ANDCA, or TV:ALU-XOR. Height and width are the size of the rectangle, and x-bitpos and y-bitpos are the location of the upper left corner.

%EXTERNAL-VALUE-CELL symbol [637] MISC-OP Lisp-Functio
Returns a locative to whatever the value cell of symbol points to. If symbol is closure bound, this will be a locative to the external value cell. Does not check that the internal value cell contains an external value cell pointer.

%FIND-STRUCTURE-HEADER ptr [502] MISC-OP Lisp-Functio

Returns the object containing the cell addressed by the locative, ptr. Finds the overall structure containing the cell addressed by ptr. Does not follow structure forwarding.

%FINDCORE [476] MISC-OP Lisp-Functio
Returns the page frame number of an available physical page. Makes one available if necessary.

%FIXNUM-MICROSECOND-TIME [471] MISC-OP Lisp-Functio
Returns the 32-bit microsecond time truncated to 25-bits and typed as a FIXNUM.

%FUNCTION-INSIDE-SELF [714] MISC-OP Lisp-Functio
Returns the functional part of SELF. If SELF is an instance, return the contents of the cell referenced by the %INSTANCE-DESCRIPTOR-FUNCTION slot of the instance descriptor. This is usually a funcallable hash array. If SELF is a closure, return the function from the closure, otherwise return NIL.

%GC-CONS-WORK nqs [153] AUX-OP Lisp-Functio
Informs the GC microcode that nqs Q's have been allocated. There is no need to do this if storage is allocated by the microcoded storage allocation routines.

%GC-FLIP region [151] AUX-OP Lisp-Functio
Flips region converting new space to old space. Ensures that nothing else in the machine points to old space. If region is T all new space is converted to old space.

%GC-FREE-REGION region [150] AUX-OP Lisp-Functio
Makes region region free. Used on old space region after scavenging is complete.

%GC-SCAV-RESET region [57] MISC-OP
Returns T if the scavenger was looking at this region, otherwise NIL is returned. This also makes the scavenger not look at region and removes region from the cons cache.

%GC-SCAVENGE work-units [152] AUX-OP Lisp-Functio
Scavenge for work-units of work or until a page fault. Returns NIL if completed work-units of work or ran out of work to do. Returns non-NIL if took a page fault before scavenging was complete. A "work-unit" is the scavenging of one Q.

%GET-SELF-MAPPING-TABLE method-flavor-name
[711] MISC-OP Lisp-Functio
Returns NIL if SELF is not an instance.
Returns the value of SELF-MAPPING-TABLE if the mapping table is already this value.
Otherwise, returns the table located by searching the

mapping table alist of the instance descriptor for SELF
for method-flavor-name and getting its CDDR.
method-flavor-name is a symbol for the flavor of the
method for which to get a self mapping table.

%INSTANCE-LOC instance, index [713] MISC-OP Lisp-Function
Returns a locative to the slot index in instance.
The lowest valid index is 1 (1-origin).

%INSTANCE-REF instance, index [712] MISC-OP Lisp-Function
Returns the contents of the slot index in instance.
The lowest valid index is 1 (1-origin).

%IO rqb, device-desc [210] MISC-OP Lisp-Function
Initiate IO request describe rqb (request-block) on the
device described by the array device-desc. The device
handler will interpret the contents of the rqb. Rqb is
often an array but may be a FIXNUM. device-desc is an
array that serves as the IO-DEVICE-DESCRIPTOR
(see section on devices).

%LOGDPB value, ppss, word [525] MISC-OP Lisp-Function
A FIXNUMs-only form of DPB. The low order ss bits of
value replace the field of word. Always returns a
FIXNUM. Does not complain about loading/clobbering
the sign bit. No error checking is performed on the
arguments.

%LOGLDB ppss, word [521] MISC-OP Lisp-Function
A FIXNUMs-only form of LDB. Returns a FIXNUM obtained
from the 32-bit uninterpreted word. This instruction
only return a field up to 25 bits. The result may be
negative if the field size is 25. Signals the ARGTYP
error if ppss is not a FIXNUM or if it specifies a field
greater than 25 bits wide.

%MAKE-EXPLICIT-STACK-LIST length [401] MISC-OP Lisp-Function
Returns a list pointer to the first element
of the list immediately before the argument length.
The length values prior to length are made into a
list; i.e., the CDR-code of the last one is changed
to CDR-NIL.

%MAKE-EXPLICIT-STACK-LIST* length [402] MISC-OP Lisp-Function
Exactly the same as %MAKE-EXPLICIT-STACK-LIST except
it makes the last element be the cdr of the list.

%MAKE-LIST initial-value, area, length [414] MISC-OP Lisp-Function
Constructs a CDR-coded list of initial-value,
length elements long in area.

%MAKE-POINTER dtp, address [446] MISC-OP Lisp-Function

Returns a Q whose data type is dtp and whose pointer field is address; exercise extreme caution.

%MAKE-POINTER-OFFSET new-dtp, pointer, offset
[447] MISC-OP Lisp-Function
Returns a Q whose data type is dtp and whose pointer field is pointer added to offset; exercise extreme caution.

%MAKE-REGION bits, size [505] MISC-OP
Creates a region whose size is at least size and whose region bits are set to bits.

%MAKE-STACK-LIST n [400] MISC-OP Lisp-Function
Returns a pointer of data type DTP-STACK-LIST to the list constructed by pushing n NILS on the stack with CDR-NEXT on them all except the last one, which receives a CDR-NIL. This does not check for PDL room and therefore the execution of %ASSURE-PDL-ROOM before this is recommended.

%MICROSECOND-TIME [470] MISC-OP Lisp-Function
Returns the 32-bit microsecond time as an integer, either a FIXNUM or a BIGNUM.

%MULTIBUS-READ-* multibus-byte-adr [431-433] MISC-OP Lisp-Function
Signals the UNIMPLEMENTED-HARDWARE error.
* is 8, 16, or 32.

%MULTIBUS-WRITE-* multibus-byte-adr word
[214-216] MISC-OP Lisp-Function
Signals the UNIMPLEMENTED-HARDWARE error.
* is 8, 16, or 32.

%NUBUS-READ NuBus-slot slot-byte-adr [434] MISC-OP Lisp-Function
Returns a signed integer, either a FIXNUM or a BIGNUM, read from the word (32-bits) at the physical address formed by taking the low order 24-bits from slot-byte-adr and concatenating the low 8-bits from NuBus-slot. For NuBus slot space accesses, the 4-bits at bit 4 of NuBus-slot should be ones. If a NuBus error occurs, then the USER-NUBUS-ERROR error is signalled. If a GACBL (Go Away Come Back Later) is encountered, it is retried. After a very large number of retries, an error is assumed and the USER-NUBUS-ERROR is signalled.

%NUBUS-READ-* hi-address low-address
[435-436] MISC-OP Lisp-Function
Returns a FIXNUM read from the byte (when * is 8b) or halfword (when * is 16B) at the physical address formed by taking the low order 24-bits from low-address and concatenating the low 8-bits from hi-address.

For NuBus slot space accesses, the 4-bits at bit 4 of low-address should be ones. If a NuBus error occurs then the USER-NUBUS-ERROR error is signalled. If a GACBL (Go Away Come Back Later) is encountered, it is retried. After a very large number of retries, an error is assumed and the USER-NUBUS-ERROR is signalled.

%NUBUS-READ-8B-CAREFUL hi-address low-address
[440] MISC-OP Lisp-Functio

Returns a FIXNUM read from the byte at the physical address formed by taking the low order 24-bits from low-address and concatenating the low 8-bits from hi-address. For NuBus slot space accesses, the 4-bits at bit 4 of low-address should be ones. If a NuBus error occurs then NIL is returned. If a bus timeout occurs then T is returned. If a GACBL (Go Away Come Back Later) is encountered, it is retried. After a very large number of retries, an error is assumed and NIL is returned.

%NUBUS-WRITE NuBus-slot slot-byte-adr word
[217] MISC-OP Lisp-Functio

Writes word (32-bits) at the physical address formed by taking the low order 24-bits from slot-byte-adr and concatenating the low 8-bits from NuBus-slot. For NuBus slot space accesses, the 4-bits at bit 4 of NuBus-slot should be ones. Word should be a signed FIXNUM or a BIGNUM. If a NuBus error occurs then the USER-NUBUS-ERROR error is signalled. If a GACBL (Go Away Come Back Later) is encountered, it is retried. After a very large number of retries, an error is assumed and the USER-NUBUS-ERROR is signalled.

%NUBUS-WRITE-* hi-address low-address data
[220-221] MISC-OP Lisp-Functio

Writes word, a FIXNUM byte (if * is 8b) or a FIXNUM halfword (when * is 16b), at the physical address formed by taking the low order 24-bits from slot-byte-adr and concatenating the low 8-bits from NuBus-slot. For NuBus slot space accesses, the 4-bits at bit 4 of NuBus-slot should be ones. If a NuBus error occurs then the USER-NUBUS-ERROR error is signalled. If a GACBL (Go Away Come Back Later) is encountered, it is retried. After a very large number of retries, an error is assumed and the USER-NUBUS-ERROR is signalled.

%OPEN-CATCH catch-tag, restart-pc [124] AUX-OP
%OPEN-CATCH-MULTIPLE-VALUE catch-tag restart-pc number-of-values
[125] AUX-OP
%OPEN-CATCH-MV-LIST catch-tag, restart-pc

```
                                                [127]   AUX-OP
%OPEN-CATCH-TAIL-RECURSIVE  catch-tag restart-pc
                                                [126]   AUX-OP
        Opens a catch block data structure on the stack.
        Catch-tag and restart-pc are some of the elements
        of that data structure.  See the discussion on
        catching and throwing.

%OPEN-MOUSE-CURSOR                               [1]  MODULE(MOUSE)

%P-CDR-CODE  pointer                           [451]  MISC-OP  Lisp-Function
%P-CONTENTS-AS-LOCATIVE  pointer               [461]  MISC-OP  Lisp-Function
%P-CONTENTS-AS-LOCATIVE-OFFSET pointer offset
                                               [462]  MISC-OP  Lisp-Function
%P-CONTENTS-OFFSET  pointer offset             [456]  MISC-OP  Lisp-Functior
%P-DATA-TYPE  pointer                          [452]  MISC-OP  Lisp-Functior
%P-DEPOSIT-FIELD value ppss pointer             [51]  MISC-OP  Lisp-Functior
%P-DEPOSIT-FIELD-OFFSET  value ppss pointer offset
                                                [53]  MISC-OP  Lisp-Functior
%P-DPB  value ppss pointer                      [50]  MISC-OP  Lisp-Functior
%P-DPB-OFFSET  value ppss pointer offset        [52]  MISC-OP  Lisp-Functior
%P-LDB  ppss pointer                           [454]  MISC-OP  Lisp-Functior
%P-LDB-OFFSET  ppss pointer offset             [457]  MISC-OP  Lisp-Functior
%P-MASK-FIELD  ppss pointer                    [455]  MISC-OP  Lisp-Functior
%P-MASK-FIELD-OFFSET  ppss pointer offset
                                               [456]  MISC-OP  Lisp-Functior
%P-POINTER  pointer                            [453]  MISC-OP  Lisp-Functior
%P-STORE-CDR-CODE  pointer cdr-code             [64]  MISC-OP  Lisp-Functior
%P-STORE-CONTENTS  pointer value                [62]  MISC-OP  Lisp-Functior
%P-STORE-CONTENTS-OFFSET  value pointer offset
                                                [67]  MISC-OP  Lisp-Function
%P-STORE-DATA-TYPE  pointer data-type           [65]  MISC-OP  Lisp-Function
%P-STORE-POINTER  pointer pointer-to-store
                                                [66]  MISC-OP  Lisp-Functior
%P-STORE-TAG-AND-POINTER  pointer misc-fields pointer-field
                                                [63]  MISC-OP  Lisp-Functior
        These pointer-manipulation miscops and other
        Lisp-coded ones are fully described in the
        section on Storage Subprimitives.

%PAGE-IN  pfn vpn                              [363]  MISC-OP  Lisp-Functior
%PAGE-STATUS  ptr                              [474]  MISC-OP  Lisp-Functior
%PAGE-TRACE  table                             [371]  MISC-OP  Lisp-Functior
%PHYSICAL-ADDRESS ptr                          [430]  MISC-OP  Lisp-Functior
        See the section on Paging and Disk Management
        for description.

%POINTER x                                     [445]  MISC-OP  Lisp-Functior
        Returns the pointer field of x as a FIXNUM.

%POINTER-DIFFERENCE ptr1 ptr2                  [463]  MISC-OP  Lisp-Functior
        Returns the number of words between pointers
        ptr2 and ptr1.  These pointers can be anything,
```

but are usually FIXNUMs or locatives. This number can change due to GC.

%RATIO-CONS numerator denominator [1037] MISC-OP Lisp-Function
Returns a rational number constructed from numerator and denominator.

%RECORD-EVENT data-4 data-3 data-2 data-1 stack-level event must-be-4
[373] MISC-OP Lisp-Function

%REGION-NUMBER x [660] MISC-OP Lisp-Function
Returns the region number object x points into, or NIL.

%SET-MOUSE-SCREEN window [0] MODULE-OP
Sets the current screen for the mouse.

%SET-SELF-MAPPING-TABLE mapping-table [22] AUX-OP

%SPREAD list [11] MISC-OP Lisp-Function
Takes a list and pushes its elements on the stack.

%STACK-FRAME-POINTER [404] MISC-OP Lisp-Function
Returns a locative pointing at the current stack frame. This happens to be the same as a pointer to local 0.

%STORE-CONDITIONAL pointer old new [464] MISC-OP Lisp-Function
Store new into pointer if the old contents of pointer match old. Returns T if the store was done, otherwise NIL. This is a basic interlocking primitive, which can be used to simulate any sort of atomic test-and-modify operation.

%STRING-EQUAL string1 index1 string2 index2 count
[233] MISC-OP Lisp-Function
T if count characters of string1 at index1 match those of string2 at index2. Similar to STRING-EQUAL, but args are slightly different and all required -- and it's faster. The comparison ignores case unless ALPHABETIC-CASE-AFFECTS-STRING-COMPARISON is non-NIL.

%STRING-SEARCH-CHAR char string start end
[701] MISC-OP Lisp-Function
The same as STRING-SEARCH-CHAR, but without coercion and error checking. Also, all the args are required. And it's faster.

%STRING-WIDTH table offset string start end stop-width
[702] MISC-OP Lisp-Function

```
%STRUCTURE-BOXED-SIZE  ptr                [503]  MISC-OP  Lisp-Function
        Returns the number of normal Lisp pointers in
        the object indicated by ptr.  This many words at
        the beginning of the object contain normal Lisp data.
        The remaining words contain just numbers (such as
        the instructions of a compiled function, or the data
        in a numeric array).

%STRUCTURE-TOTAL-SIZE  ptr                [504]  MISC-OP  Lisp-Function
        Returns the number of words in the object
        indicated by ptr.

%SXHASH-STRING  string character-mask     [700]  MISC-OP  Lisp-Function
        Returns a hash code for string computed on a
        character-by-character basis after applying
        character mask to each character.

%TEST&SET-68K  slot offset                [227]  MISC-OP
        Test for 1 (then set it) in high bit of byte
        specified by slot and offset.  Return T if bit
        is not already set, else NIL.

%THROW  tag value                         [130]   AUX-OP
%THROW-N  tag &rest values-and-count      [131]   AUX-OP
        Throw one or more values to tag.  See discussion
        of throwing in the Function Calling section.

%UNWIND-PROTECT-CLEANUP                    [15]   AUX-OP
%UNWIND-PROTECT-CONTINUE                  [132]   AUX-OP
        Maybe continue throwing after unwind-protect
        undo-forms.

%USING-BINDING-INSTANCES  binding-instances
                                           [16]   AUX-OP  Lisp-Function
        See section on subprimitives for description.

%WRITE-INTERNAL-PROCESSOR-MEMORIES  code adr d-hi d-low
                                          [370]  MISC-OP  Lisp-Function

*BOOLE  fn arg1 arg2                      [533]  MISC-OP
-  x y                                     [61]  MAIN-OP
LOGAND  x y                                [63]  MAIN-OP
LOGIOR  x y                               [542]  MISC-OP
LOGXOR  x y                                [64]  MAIN-OP
MAX  num1 num2                            [534]  MISC-OP
MIN  num1 num2                            [535]  MISC-OP
+  x y                                     [60]  MAIN-OP
QUOTIENT  num1 num2                       [541]  MISC-OP
*  x y                                     [62]  MAIN-OP
        Limited-argument instructions and miscop forms
        of common functions.  See descriptions of
        corresponding multi-argument Lisp functions.
```

```
*UNWIND-STACK  tag value frame-count action
                                                [133]   AUX-OP  Lisp-Functio
        Throw completely out of a computation, running
        all unwind protects found along the way.  See
        section on function calling for details.

1+  n                                           [65]  MAIN-OP  Lisp-Functio
1-  n                                           [66]  MAIN-OP  Lisp-Functio
        Pushes on the stack the result of incrementing
        or decrementing the operand.

<-IMMED  x immed-y                               [6]  IMMEDIATE
=-IMMED  x immed-y                               [4]  IMMEDIATE
>-IMMED  x immed-y                               [5]  IMMEDIATE
        Sets the indicators as a result of comparing
        a number popped from the stack to the 9-bit immediate
        operand treated as a signed integer.

<  num1 num2                                    [570]  MISC-OP
=  num1 num2                                    [572]  MISC-OP
>  num1 num2                                    [571]  MISC-OP
        Limited-argument miscop forms of common functions.
        See descriptions of corresponding multi-argument
        Lisp functions.
```

APPENDIX A

Data Structures

A.1 NONVOLATILE RAM (NVRAM)

A.1.1 NVRAM Data Structure Definitions.

The NVRAM data structures may be divided into several independent sections. The first section contains information about the location of resources (monitor, keyboard, and load device) required for system testing and booting. The next section contains information about the last system shut down, including cause of shutdown, when the previous system boot occured, and how long the system was running before it shutdown. Crash record "registers" make up the next section, providing control for a circular buffer of crash records (the crash record buffer is located in the same NVRAM, after additional standard sections). Following the crash record registers are sections which provide for dynamic allocation management of the rest of the NVRAM, including the "typed block" area.

A.1.1.1 Test and Boot Resources.

The NVRAM contains values which the System Test and Boot Master (STBM) references in order to find preselected resources. The NuBus slot and a 24 bit logical unit number are provided for a monitor, a keyboard (to be used for user interaction), and a default load source (to be used if a default system load occurs).

Since the NVRAM contents may be invalid if the NVRAM has not been initialized or if the battery runs low, it is necessary to validate the contents of the NVRAM before they are used. Three mechanisms are provided for this function, although all may not be used in some implementations. First, the NVRAM data structure format generation and revision are stored so that they may be validated. The format generation should never change from >01 if it conforms to this specification, but the revision may change as features are added to the NVRAM such that they are upward compatable. Second, the test and boot resource information and the format generation and revision fields are (optionally) protected by a 16 bit CRC value. The CRC calculation method used is identical to that used in the Configuration ROM and provides protection for the first 14 bytes of the NVRAM. Finally, a configuration checksum (optionally) is stored in the NVRAM so

that the chassis configuration can be verified as being unchanged from when the NVRAM test and boot resources information was last updated.

A.1.1.2 Last Shutdown Information.

Information is (optionally) stored in the NVRAM that can be used following a system boot to determine information about the previous system boot and shutdown. The shutdown information includes date and time of last boot, how long the system was running before the shutdown, and cause of the shutdown. After the operating system reads and logs the information in the shutdown fields from the previous system boot, it should update them to reflect the current boot date and time. Then the time since boot should be updated periodically (at least once per minute) so that it is current if an unexpected shutdown occurs. When a system shutdown occurs the shutdown cause should be logged in the NVRAM and the shutdown information valid character set to allow the next boot operation to determine that the shutdown cause was corractly recorded.

A.1.1.3 Crash Record Registers.

The Crash Record Registers are a set of dedicated NVRAM locations that provide a control structure for a circular buffer (located in the NVRAM) where a system processor stores a predefined set of information when it detects a system crash condition. Since it is desirable for there to be a history of several previous crashes stored in the NVRAM at one time, the buffer must be managed such that at any time a new crash record can be added to the buffer, and so that system software can retrieve the crash history without confusion. The Crash Record Registers provide the data necessary for this management function. They include information about the crash record format and size, buffer location in NVRAM and size (pointers to the beginning of first and last records), and a pointer to the currently active crash record.

A.1.1.4 Typed Blocks.

It is desirable to be able to access, allocate and deallocate areas of the NVRAM as needed without disturbing other allocated data structures. NVRAM Typed Blocks provide a generic mechanism for accessing, allocating, and deallocating variable length blocks of NVRAM. Typed Blocks are used to manage the entire NVRAM, excluding the fixed system parameters in the first 64 bytes (offsets 00-FF). The size of Typed Blocks are determined as they are allocated, so that they can be customized to the needs of particular applications. In order to support the required operations, each Typed Block contains: reserve block

Block Type Number	16-bit, uniquely identifies the use of each Typed Block

Block Length	16-bit, specifies the entire byte length of each Typed Block, inclusive of the Block Type Number, the Block Length, and Block Data
Block Data	Application specific data format and content

The Block Type Number and Block Length are used during searches through NVRAM for any specific kind of Typed Block. The Block Type Number provides for the unique recognition of blocks while the Block Length is used to determine the location of the start of the next block.

Block Type Numbers are unique, assigned numbers that designate the use of each Typed Block. Two Block Type Numbers are reserved for the generic functions of Available and End, a range is reserved for TI use, and the remainder are available for assignment to NuBus license holders upon request. Assigned block type numbers are recorded in TI drawing part number 2549287-0001.

Block Type Number >FFFF indicates that a Typed Block is currently Available to be allocated. Zero or more Available blocks may be present at one time in each NVRAM. If no Available blocks are present then there is no remaining space for expansion (unless some already allocated block is removed and its space recovered). Multiple Available blocks may result from deallocation of previously used blocks.

Block Type Number >FFFE indicates the End Block, which marks the end of the NVRAM area dedicated for Typed Blocks. No allocation of NVRAM space shall extend beyond the End Block. Normally the End Block has a Block Length of four (i.e., zero length data section) and is placed in the last four bytes of an NVRAM.

A Typed Block of any particular Number is found by searching through the NVRAM starting at offset >100. Each block's Type Number found is compared with the desired number and the End number (>FFFE). If the End block is found before the desired block, then none has been allocated. If the current block is neither the desired block or the End block, add the Block Length of the current block to the current offset and check the next block.

Typed Block allocation and deallocation is done by changing the Type Block Number of an already existing block and adjusting the Block Lengths. Allocation is done by "carving" an application block out of an Available block. Deallocation is done by returning any block to the Available mode. Note that the allocation operation must be done in coordination with all other

processors in the system such that a collision does not occur by two processors attempting to allocate the same, previously Available block. This is easily accomplished by each processor during the boot process while each processor is loading itself as either the Primitive or as a Secondary but before reaching the Synchronization Point.

A.1.1.5 NVRAM Format.

The following provides specific information about the NVRAM standard data structures:

System default configuration information:

base +>00=	STBM Monitor unit number LSB byte	Binary
base +>04=	STBM Monitor unit number MID byte	Binary
base +>08=	STBM Monitor unit number MSB byte	Binary
base +>0C=	STBM Monitor slot number (FF= none)	Binary
base +>10=	STBM Keyboard unit number LSB byte	Binary
base +>14=	STBM Keyboard unit number MID byte	Binary
base +>18=	STBM Keyboard unit number MSB byte	Binary
base +>1C=	STBM Keyboard slot number (FF= none)	Binary
base +>20=	Boot source device unit LSB byte	Binary
base +>24=	Boot source device unit MID byte	Binary
base +>28=	Boot source device unit MSB byte	Binary
base +>2C=	Boot source device slot (FF= none)	Binary
base +>30=	NVRAM format generation number. Equal >01 for all NuGeneration devices.	Binary
base +>34=	NVRAM format superset revision number.	Binary
base +>38=	NVRAM CRC LSB byte	Binary
base +>3C=	NVRAM CRC MSB byte CRC calculated same as for Config. ROM except it does not cover the entire NVRAM, rather only the system configuration information in the range base +>00 through base + >34.	Binary

Configuration Checksum:

base +>40=	Config. Checksum LSB byte	Binary
base +>44=	Config. Checksum MSB byte 16 bit sum (overflow ignored) generated by adding together all 16 bytes of the Part Number field of every slot Configuration ROM. If a slot is empty or does not appear to contain a Configuration ROM then it does not affect the checksum. This value may be used to verify that the	Binary

system configuration has not changed.
(note that moving a board to another slot
does not change the checksum).

base +>48= reserved for use as Synchronization Flag by STBM
base +>4C= reserved for use in board tests of NVRAM

last shutdown information:

base +>50=	Abnormal Shutdown Valid character. Set to ASCII "V" (>56) if valid shutdown information was stored	ASCII
base +>54=	shutdown cause: >00= overvoltage shutdown >01= undervoltage shutdown >02= overvoltage after high temperature >03= high temperature shutdown >04-FF= reserved	Binary
base +>58-5C	reserved	
base +>60=	month of BOOT, (Jan=1, Feb=2...)	Binary
base +>64=	day of month of BOOT (1..31)	Binary
base +>68=	hour of day of BOOT (0-23)	Binary
base +>6C=	minute of hour of BOOT (0-59)	Binary
base +>70=	Seconds since BOOT LSB byte	Binary
base +>74=	Seconds since BOOT	Binary
base +>78=	Seconds since BOOT	Binary
base +>7C=	Seconds since BOOT MSB byte	Binary

crash records registers:

base +>80=	Crash Record Format Processor type LSB	Binary
base +>84=	Crash Record Format Processor type MSB	Binary
base +>88=	Crash Record Format Revision	Binary
base +>8C=	reserved	
base +>90=	Currently active crash record LSB	Binary
base +>94=	Currently active crash record MSB	Binary
base +>98=	Crash Record Size LSB byte	Binary
base +>9C=	Crash Record Size MSB byte	Binary
base +>A0=	Crash Record Buffer, last record LSB	Binary
base +>A4=	Crash Record Buffer, last record MSB	Binary
base +>A8=	Crash Record Buffer, first record LSB	Binary
base +>AC=	Crash Record Buffer, first record MSB	Binary
base +>B0-EC=	reserved	

NVRAM Typed Blocks list:

```
base +>100      Start of NVRAM Typed Blocks

 typed block format (starting at NVRAM base + >100 ):

block + >0=     Block Type Identifier LSB byte          Binary
block + >4=     Block Type Identifier MSB byte          Binary
                  controlled to be unique 16 bit
                  number identifiing block type:
                  0000-0007= reserved for TI diagnostics
                  0008-000F= reserved for TI Lisp
                  0010-0017= reserved for TI S1500 (tm)
                             (S1500 is a trademark of Texas
                              Instruments Incorporated.)
                  0018-00FF= reserved for TI
                  0100-FFFD= available for assignment,
                             see TI drawing 2549287-0001
                  FFFE     = End block
                  FFFF     = Available block

block + >8      Block Length LSB byte                  Binary
block + >C      Block Length MSB byte                  Binary

block + >10...  Block Data              (application specific)
```

A.2 DISK PARTITION STRUCTURES

There are several standard partitions (contiguous sections of the allocatable media) defined on disk. These include the Volume Label, Partition Table, Save Partition, Test Zone, and Format Information partition. The following sections briefly describe the function of each.

NOTE

All partition addresses below are expressed in blocks. A block is 1024 bytes.

A.2.1 Volume Label Partition.

Each volume contains a variety of partitions which are accessed through a minimal "directory" whose root is the volume label. The label, in logical block 0 of the volume, is the only partition with a fixed location. All other partition locations are determined indirectly via pointers found in first the label, then in the Partition Table. The label contains information about the physical and logical characteristics of the volume and also provides pointers to the Save and Partition Table partitions.

The second block of the label (logical block 1) is reserved as a fixed location, emergency data "save" area of exactly one block length. Use of this block is system implementation dependent. It is not an actual partition (as is the Save partition) but rather just a spare block with a known address.

A.2.2 Partition Table Partition.

The Partition Table provides name, characteristics, pointer and size information for each partition on the volume. The first section of the Partition Table provides information about the table format, such as number of partitions in the table, size of each entry, and offset in each entry to comments.

Following the format information table is the table of individual entries describing each partition. Each entry in the table has a set of partition "key" characteristics. These include:

* Name of the partition

* Partition Type
* Partition User Type
* Partition attribute bits

Any combination of the characteristics may be used to uniquely identify a particular partition. For example, a partition with a particular Name (MCR1), Partition Type (>01=microload), and User Type (>0000=Explorer) may be found, even though other partitions may have the same Name or Partition or User Type. Likewise, the default partition, identified by the default bit (Default attribute=1), Partition Type (>01=microload), and User Type (>0000=Explorer) can be located regardless of what Name the partition may have (this is the mechanism used in the booting process to locate default partitions of specific types).

Several comments about partitions and key values are in order:

1. Name + Partition Type + User Type combinations should be unique, however, a single Name may well be used again for a different Partition Type and/or User Type, and obviously both Partition and User Type shall be used many times.
2. Multiple entries in the table may point to the same physical partition with different key characteristics being used to identify the partition.
3. All searches through the partition table are assumed to start at the beginning of the table and continue until either a match of the desired key characteristics occurs or the end of the table is reached.
4. The order of the entries in the partition table does not imply information about the physical order of partitions on the volume.
5. Zero length partitions are allowed, and indeed are sometimes used as "markers".
6. There may be space on the volume that is not allocated to any partition, and therefore does not appear in the table.

A.2.3 Test Zone Partition.

The Test Zone Partition (TZON) is provided to accommodate testing the hardware. It is used to access the volume without disturbing system or user data. It includes fixed data patterns that can be used to test data read channels and blocks that are reserved for

write-then-read tests to check write channels. This partition is vital for providing device maintenance; it should never be removed from any volume.

A.2.4 Format Information Partition.

The Format Information Partition (FMT) contains details about the way the volume media was tested and formatted. Several sections of data are recorded, including physical sector and track data formats, surface analysis methods used, and detail lists of defects found when the media was initialized. This partition also should never be removed or destroyed. Also, it should not be copied from one disk to another - it is device specific.

A.2.5 Partition Descriptions.

For the following partition layouts, these notes apply.

1. All "text" is in the form of ASCII strings, with the characters read left to right stored in ascending byte address order.

2. It is important that any "text" contain only standard characters (i.e., no special escape or control ASCII characters) in order to provide for portability of the disk between the expected variety of Nu Generation (tm) systems. (Nu Generation is a trademark of Texas Instruments Incorporated.)

3. All four character "text" fields must contain valid ASCII characters in all positions.

4. Comment fields contain "text" of variable length within the maximum field length, with a byte of >00 after the last character of "text". Any remaining characters following the >00 to the end of the field are garbage.

5. In the following description the following abbreviations are used:

   a. LSB = least significant byte

   b. DML = middle low byte

   c. MDH = middle high byte

   d. MSB = most significant byte

   e. "X" = an ASCII character

   f. xxxx xxxN = indicates bit position within a byte

g. >0 = all bits set to zero

6. All numbers are unsigned binary values (unless otherwise defined).

Volume Label

Label description (block #0):	offset	value
1. "LABL" (4 ASCII character)	+000	"L"
	+001	"A"
	+002	"B"
	+003	"L"
2. Revision in four bytes: (number)	+004	LSB byte
	+005	MDL byte
	+006	MDH byte
	+007	MSB byte
3. Reserved (0's, as for all reserved unless stated otherwise)	+008-00F	>0

Type storage:

4. Flag word:	bit: 210		
4a. Bit 0-2 typecode:	000=disk 001=tape 010=Wrt-Once-Rd-Many (WORM) 011-111=reserved	+010	xxxx xNNN 7654 3210
4b. Bit 3 fixed:	0=removable; 1=fixed	+010	xxxx Nxxx 7654 3210
4c. Bit 4 logical:	0=phys addr; 1=logical addr	+010	xxxN xxxx 7654 3210
----------tape----------			
	bit: 765		
4d. Bit 5-7 tape opt:	000=stream only 001=start/stop 010=stream w/track select 011=start/stop w/track select 100-111=reserved	+010	NNNx xxxx 7654 3210
----------disk----------			
4e. Bit 5-7 reserved		+010	NNNx xxxx 7654 3210
----------disk and tape----------			
4f. bit 8-31 reserved		+011-013	>0
5. Device name text (e.g., MAX-0140; CDC-0030) OR reserved (if reserved = >0)		+014 +015 : +01F	"M" "A" : " "

Addressing:

6.	# of bytes/block (required for partition table)	+020 +021	LSB byte MSB byte
	----------disk----------		
7.	# of bytes/sector	+022 +023	LSB byte MSB byte
8.	Reserved	+024-025	>0
9.	# of sectors/track	+026	byte
10.	# of heads (tracks/cyl)	+027	byte
11.	# of cylinders	+028 +029	LSB byte MSB byte
12.	# of reserved sectors for defects*	+02A +02B	LSB byte MSB byte

* = note that the above parameters are used to calculate device maximum capacity by the following equation:

total 1k blocks =
(((#cyls * tracks/cyl * sectors/track)
- (#reserved sectors)) * bytes/sector) / 1024

13.	Reserved	+02C-02F	>0
	----------tape----------		
14.	# of bytes/physical block	+022 +023	LSB byte MSB byte
15.	Reserved	+024-025	>0
16.	# of tracks (if supported)	+026 +027	LSB byte MSB byte
17.	# of blocks (nominal)	+028 +029 +02A +02B	LSB byte MDL byte MDH byte MSB byte
18.	Reserved	+02C-02F	>0
19.	Volume name (16 characters of text)	+030 +031 +032 +033	"N" "A" "M" "E"

		+03F	"X"
20.	Reserved	+040-04F	>0
21.	Partition table name = "PTBL"	+050	"P"
		+051	"T"
		+052	"B"
		+053	"L"
22.	Starting block address of partition table	+054	LSB byte
		+055	MDL byte
		+056	MDH byte
		+057	MSB byte
23.	Length of partition table in blocks	+058	LSB byte
		+059	MDL byte
		+05A	MDH byte
		+05B	MSB byte
24.	Reserved for additional partition table information	+05C-06F	>0
25.	Secondary SAVE store partition name = "SAVE"	+070	"S"
		+071	"A"
		+072	"V"
		+073	"E"
26.	Starting block address of secondary SAVE partition	+074	LSB byte
		+075	MDL byte
		+076	MDH byte
		+077	MSB byte
27.	Length of secondary SAVE partition in blocks	+078	LSB byte
		+079	MDL byte
		+07A	MDH byte
		+07B	MSB byte
28.	Reserved for additional secondary SAVE store information	+07C-08F	>0
29.	Reserved for future label information	+090-0FF	>0
30.	Volume comments, text	+100-3FF	"X"

Label Save Store (Block #1)

NOTE: There is only a single block #1 save store. Therefore, its use must be very carefully controlled in multiprocessor system to prevent problems.

31. Internal SAVE contents are processor specific.

PARTITION TABLE (PTBL)	offset	value
1. "PRTN" (identifier reserved for partition table) Note: This is not the partition table name, but rather a value used to identify a valid partition table.	+000 +001 +002 +003	"P" "R" "T" "N"
2. Partition table revision number	+004 +005 +006 +007	LSB byte MDL byte MDH byte MSB byte
3. Number of partitions in partition table.	+008 +009 +00A +00B	LSB byte MDL byte MDH byte MSB byte
4. Size of each partition table entry in long words (32 bit words) (for current revision = >00000010)	+00C +00D +00E +00F	LSB byte MDL byte MDH byte MSB byte
5. Offset in long words (32 bit) to comment (for current revision = >00000004)	+010 +011 +012 +013	LSB byte MDL byte MDH byte MSB byte
6. Reserved for additional partition table information	+014-03F	>0

Partition table entries: (length may vary with revision)

7.	Partition name (ASCII string)	entry +000	"N"
		entry +001	"A"
		entry +002	"M"
		entry +003	"E"
8.	Partition start address in blocks	entry +004	LSB byte
		entry +005	MDL byte
		entry +006	MDH byte
		entry +007	MSB byte
9.	Partition length in blocks	entry +008	LSB byte
		entry +009	MDL byte
		entry +00A	MDH byte
		entry +00B	MSB byte

Partition attributes:

10.	Byte 0: generic function type code:	entry +00C	byte

* 00=	load band	(processor specific)
* 01=	microload band	(processor specific)
02=	page band	(processor specific)
03=	file band	(operating system specific)
04=	meter band	(processor specific)
05=	test zone band	(generic)
06=	format parameter band	(generic)
* 07=	volume label	(generic)
08=	save band	(generic)
* 09=	partition band	(generic)
* 0A=	configuration band	(generic)
0B=	user defined type #1	(processor/O.S. dependent)
0C=	user defined type #2	(processor/O.S. dependent)
0D=	user defined type #3	(processor/O.S. dependent)
0E=	user defined type #4	(processor/O.S. dependent)
0F=	user defined type #5	(processor/O.S. dependent)
10=	user defined type #6	(processor/O.S. dependent)
11=	user defined type #7	(processor/O.S. dependent)
12=	user defined type #8	(processor/O.S. dependent)
13=	user defined type #9	(processor/O.S. dependent)
14=	user defined type #10	(processor/O.S. dependent)
15-FE=	reserved	
FF=	empty	(generic)

* = One partition of this (function type)/(CPU type) is expected to have the "Default" bit set.

```
11. Byte 1,2: USER type:                          entry +00D  LSB byte
      Unique CPU numbers assigned to each         entry +00E  MSB byte
      processor or unique operating
      sys./class values:

        >0000-EFFF:  range for CPU identifiers. Same as "type
                     of processor" in "board type" field of
                     configuration ROM.

        >F000-FBFF = reserved for additional special blocks

        >FC00-FEFF:  range for use as operating system
                     identifiers (unique numbers to be
                     controlled; single members to be
                     assigned upon request as needed).
                     See TI drawing part number 2549237-0001
                     for a complete list.

        >FC00      = TI Lisp
        >FC01      = TI GDOS Files for S1500
        >FC02      = TI S1500

        >FF00-FFFF:  reserved for use as magic values
                     (unique numbers to be controlled;
                     single numbers to be assigned upon
                     request as needed).
                     See TI drawing part number 2549287-0001.

        >FFFF      = processor and O.S. independent
                     (i.e., all "generic" bands)

12. Bytes 3 property bits (position encoded):  entry +00F  byte
                                                bit:31     24
12a. Bit 31=expandable (1=expandable)           Nxxx xxxx
12b. Bit 30=contractable (1=contractable)       xNxx xxxx
12c. Bit 29=delete protected  (1=protected)      xxNx xxxx
12d. Bit 28=logical partition (1=part of
       logical part)                            xxxN xxxx
12e. Bit 27=copy protected    (1=protected)      xxxx Nxxx
12f. Bit 26=default indicator (1=default)       xxxx xNxx
12g. Bit 25=diagnostic use (1=diagnostic use)   xxxx xxNx
12h. Bit 24=reserved                            xxxx xxxN

NOTE: THESE BITS MAY ONLY BE CHANGED
      BY THE OWNER OF THE PARTITION

   ----- revisions may change what's below this line -----

13. Partition comment text                      entry +010-03F "X"
(always in multiples of 32-bit words)
```

"TZON" (Test Zone) band

1. TZON block #0:

Read only test pattern of fixed values:
(a sequence of values, incremented by 1, beginning at "00". The sequence increments to 255 (FF), and then restarts at 00.)

offset	value
+000	>00
+001	>01
+002	>02
:	:
+0FE	>FE
+0FF	>FF
+100	>00
+101	>01
:	:
+3FF	>FF

2. TZON block #1:

Read only test pattern of fixed values:
The following pattern repeated 4 times (to fill the block):

	+00	+01	. . . . .	+0E	+0F
+000 :	>AA	>AA	. . . . .	>AA	>AA
+010 :	>FF	>FF	. . . . .	>FF	>FF
+020 :	>00	>04	. . . . .	>00	>04
+030 :	>BF	>FF	. . . . .	>BF	>FF
+040 :	>2A	>E3	. . . . .	>2A	>E3
+050 :	>6A	>DB	. . . . .	>6A	>DB
+060 :	>B6	>6D	. . . . .	>B6	>6D
+070 :	>96	>A5	. . . . .	>96	>A5
+080 :	>24	>49	. . . . .	>24	>49
+090 :	>33	>33	. . . . .	>33	>33
+0A0 :	>98	>31	. . . . .	>98	>31
+0B0 :	>1A	>49	. . . . .	>1A	>49
+0C0 :	>CF	>23	. . . . .	>CF	>23
+0D0 :	>8F	>46	. . . . .	>8F	>46
+0E0 :	>63	>53	. . . . .	>63	>53
+0F0 :	>0A	>CC	. . . . .	>0A	>CC

3. TZON blocks #2 & 3 — Reserved for future read-only patterns.

4. TZON blocks #4 - N — Write/read test blocks. Number of blocks reserved for write/read

computed as ((blocks_per_track * heads_per_cylinder) + 2)

"FMT " (Format Information) band (9K, fixed length)

FMT block #0: (read-only)	offset	value
1. Sector skew from head to head:	+000	LSB byte
	+001	MSB byte
2. Sector interlace value:	+002	LSB byte
	+003	MSB byte
3. Sector blocking within interlace factor:	+004	LSB byte
	+005	MSB byte
Sector format information:		
4. Gap 1 size in bits:	+006	LSB byte
	+007	MSB byte
5. Gap 2 size in bits:	+008	LSB byte
	+009	MSB byte
6. Gap 3 size in bits:	+00A	LSB byte
	+00B	MSB byte
7. Header size in bytes:	+00C	LSB byte
	+00D	MSB byte
8. Data field size in bytes:	+00E	LSB byte
	+00F	MSB byte
9. Header sync character (LSB justified):	+010	LSB byte
	+011	MSB byte
10. Data sync character (LSB justified):	+012	LSB byte
	+013	MSB byte
11. Header ECC or CRC polynomial (64 bit, LSB justified):	+014	LSB byte
	:	
	+01B	MSB byte
12. Data ECC or CRC polynomial (64 bit, LSB justified):	+01C	LSB byte
	:	
	+023	MSB byte
13. Encoding method, TEXT string (8 characters): (examples: "MFM ", "RL27")	+024	"M"
	+025	"F"
	+026	"M"
	027-02B	>0
14. Reserved:	+02C-03B	>0

Surface analysis parameters (based on media characteristics):

15. Number of reads per read type:	+03C	LSB byte
	+03D	MSB byte
16. Allowable soft errors as percent of total reads per read type:	+03E	LSB byte
	+03F	MSB byte
17. Allowable soft errors as percent of total reads per data pattern:	+040	LSB byte
	+041	MSB byte
18. Allowable soft errors as percent of total reads:	+042	LSB byte
	+043	MSB byte

19. Effective surface analysis margining combinations. Each two byte set of flags specifies a test read combination (if no bits are set then no read is performed).

LSB byte:

		bit: 7654 3210
19a.	Bit 0: Nominal read	xxxx xxxN
19b.	Bit 1: Offset forward 1	xxxx xxNx
19c.	Bit 2: Offset forward 2	xxxx xNxx
19d.	Bit 3: Offset reverse 1	xxxx Nxxx
19e.	Bit 4: Offset reverse 2	xxxN xxxx
19f.	Bit 5: Strobe early 1	xxNx xxxx
19g.	Bit 6: Strobe early 2	xNxx xxxx
19h.	Bit 7: Strobe late 1	Nxxx xxxx

MSB byte:

		bit: 7654 3210
19i.	Bit 0: Strobe late 2	xxxx xxxN
19j.	Bit 1-7: Reserved	NNNN NNNx

19k.	Combination #0	+044	LSB byte
		+045	LSB byte
19l.	Combination #1	+046	LSB byte
		+047	LSB byte
19m.	Combination #2	+048	LSB byte
		+049	LSB byte
19n.	Combination #3	+04A	LSB byte
		+04B	LSB byte
19o.	Combination #4	+04C	LSB byte
		+04D	LSB byte
19p.	Combination #5	+04E	LSB byte
		+04F	LSB byte
19q.	Combination #6	+050	LSB byte
		+051	LSB byte
19r.	Combination #7	+052	LSB byte
		+053	LSB byte

20. Worst case patterns for surface analysis:

20a. Pattern #0	+054	LSB byte
	+055	MSB byte
20b. Pattern #1	+056	LSB byte
	+057	MSB byte
20c. Pattern #2	+058	LSB byte
	+059	MSB byte
20d. Pattern #3	+05A	LSB byte
	+05B	MSB byte
20e. Pattern #4	+05C	LSB byte
	+05D	MSB byte
20f. Pattern #5	+05E	LSB byte
	+05F	MSB byte
20g. Pattern #6	+060	LSB byte
	+061	MSB byte
20h. Pattern #7	+062	LSB byte
	+063	MSB byte

21. Reserved: +064-3FF >0

FMT block #1~8: ("MAP" area, lists all media defects)

	offset	value
22. Total number of defects in the map:	+400	LSB byte
	+401	MSB byte
23. Drive serial number (12 character text string):	+402	"X"
	+403	"X"
	:	
	+40D	"X"
24. Original date of formatting: (8 char. text string; format= "MmDdYyyy")	+40E	"M"
	+40F	"m"
	+410	"D"
	+411	"d"
	+412	"Y"
	+413	"y"
	+414	"y"
	+415	"y"
25. Latest date of formatting: (8 char. text string; format= "MmDdYyyy")	+416	"M"
	+417	"m"
	+418	"D"
	+419	"d"
	+41A	"Y"
	+41B	"y"
	+41C	"y"
	+41D	"y"

26. Reserved: +41E-41F >0

27. Defect entries, 16 bytes each, in ascending order:
+420-end (ascending order of cylinder/head/byte_
from_index)
Up to 511 entries are accomodated.
Each entry contains the following:

27a. Defect head address (1 byte number)	entry +00	byte
27b. Defect cylinder address (3 byte number)	entry +01	MDL byte
	entry +02	MDH byte
	entry +03	MSB byte
27c. Defect head address (1 byte number)	entry +03	byte
27d. Defect displacement from index (number of bytes):	entry +04	LSB byte
	entry +05	MDL byte
	entry +06	MDH byte
	entry +07	MSB byte
27e. Defect length (number of bits; >FF = 255 bits or longer)	entry +08	MSB byte
27f. Reserved:	entry +09-0F	>0

A.3 MICRO-LOAD PARTITION FORMAT SPECIFICATION FOR THE EXPLORER II PROCESSOR

A.3.1 Introduction.

A.3.1.1 Scope.

This specification defines the format for microcode files and partitions for the Explorer II and other processors which use the TI Lisp Microprocessor (tm). (Lisp Microprocessor is a trademark of Texas Instruments Incorporated.) This is the source document for information necessary to produce system utilities such as assemblers, MCR file-to-partition functions, and any ROM or processor down-loading programs. This standardized format is to be used since ROM resident system booting operations can not be changed, and it is expensive to support variations on utilities (assembler, spyport, mcr-loader, etc.).

NOTE

The maximum length of a microcode partition is limited to one megabyte due to limitations on memory space available during booting.

This format for microcode partitions is slightly different than that of the Explorer I processor. This is due to differences in the processor architecture and in order to improve the robustness of booting for newer products.

The term microload is used to indicate the microcode object file or the microcode disk partition format.

A.3.1.2 Reference Documents.

The following documents provide reference material necessary to understand the implications and applications of the information presented in this document.

1. NuBus System Architecture Specification 2536702-0001
2. Explorer II Processor Specification 2540834-0001

3. Explorer Lisp Microprocessor Specification

A.3.2 Overview. A microcode partition contains a number of types of data required to initialize the Explorer Lisp Microprocessor class of processors. The first section of the partition contains a Processor ID value (unique per processor type, and matches the Processor Type Value in the target processor's Configuration ROM), the assembly version number, a partition length, and checksum values. Additional sections contain a variety of information to initialize various memories and registers in the Lisp Microprocessor such as the local I/O address space on the processor board and virtual memory.

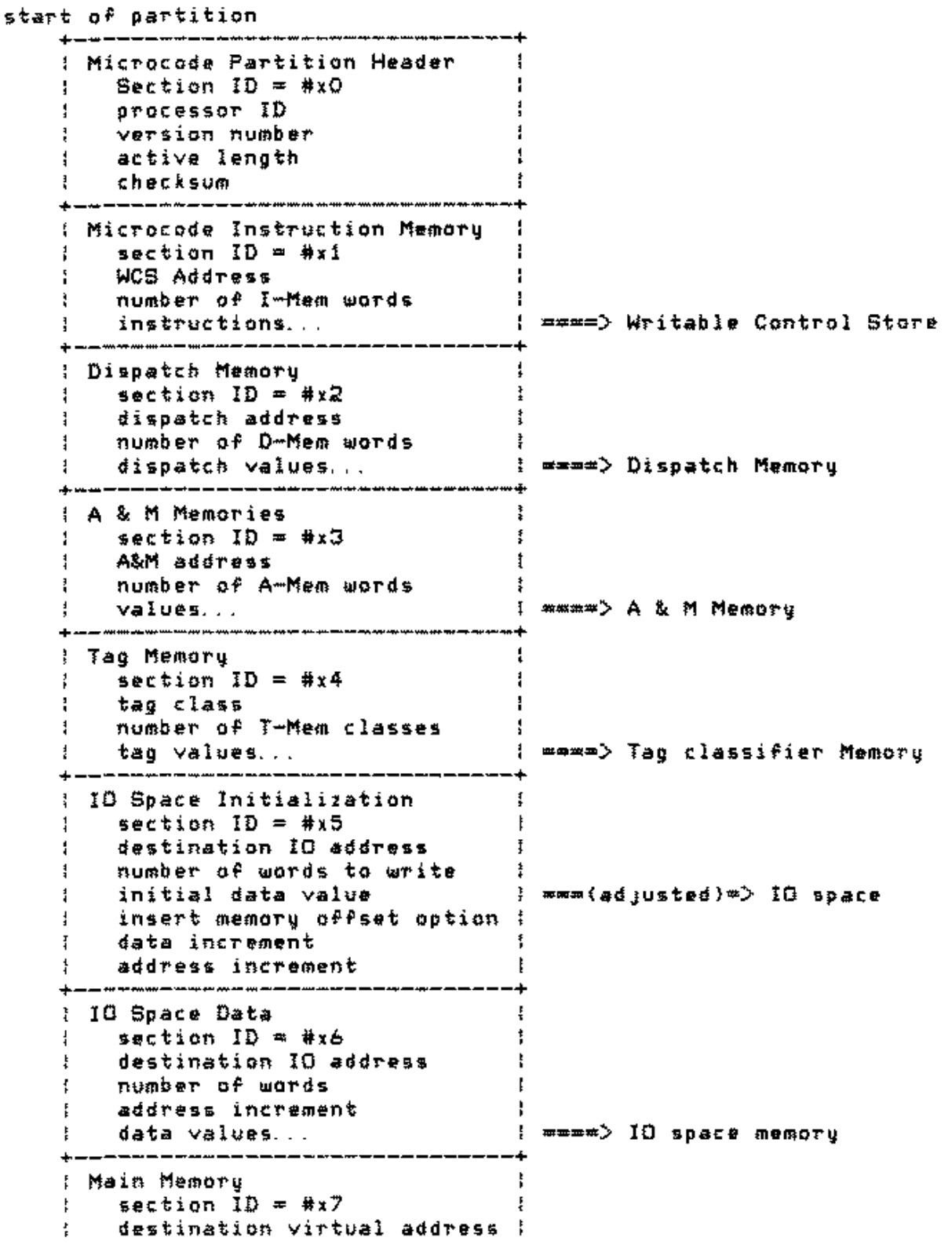
ROM and spyport loaders use the microcode partition sections to perform specific Lisp Microprocessor, board, and system memory initializations. The following sections may be contained within the microload partition:

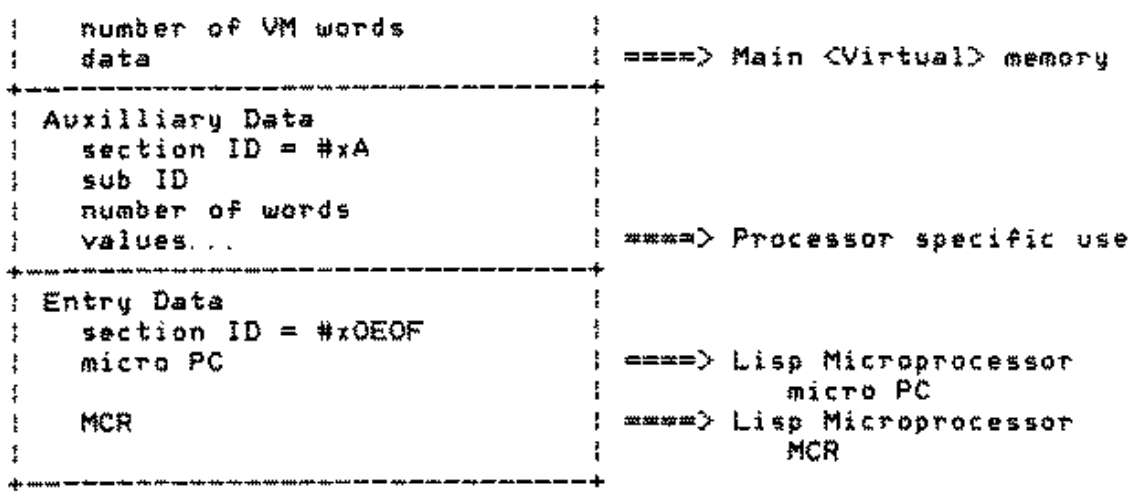
Microcode Partition Header	(required)
Microcode Instruction Memory	(required)
Dispatch Memory	(optional)
M & A Memories	(optional)
Tag Memory	(optional)
Main <virtual> Memory	(optional)
Input/Output Space memory	(optional)
Input/Output Space Initialization	(optional)
Auxiliary Data	(optional)
Entry Data	(required)

A microcode partition is formed of contiguous sections with no top level or embedded pointers to the individual section. Each section is processed in order.

Optional sections may be repeated any number of times within a given microload and are located anywhere between the header and entry data sections. Since multiple sections of any given type may coexist in same the microload, data of later processed sections may overwrite any previously processed section. The processing of the microload will continue sequentially until the Entry Data section is read.

The following diagram delineates sections within a microcode partition. Note that this example does not demonstrate that most section types can appear more that once or in any order within the microload. The ROM and spyport loaders are responsible for processing sections sequentially and to insure that the resultant data resides in the appropriate memories. The assembler is responsible for producing a file of the format described by this specification. All values in the format are stored in pure Little Endian form (i.e. ordered from least significant to most significant).


start of partition
Microcode Partition Header
Section ID = #x0
processor ID
version number
active length
checksum
Microcode Instruction Memory
section ID = #x1
WCS Address
number of I-Mem words
instructions...
====> Writable Control Store
Dispatch Memory
section ID = #x2
dispatch address
number of D-Mem words
dispatch values...
====> Dispatch Memory
A & M Memories
section ID = #x3
A&M address
number of A-Mem words
values...
====> A & M Memory
Tag Memory
section ID = #x4
tag class
number of T-Mem classes
tag values...
====> Tag classifier Memory
IO Space Initialization
section ID = #x5
destination IO address
number of words to write
initial data value
insert memory offset option
data increment
address increment
====(adjusted)=> IO space
IO Space Data
section ID = #x6
destination IO address
number of words
address increment
data values...
====> IO space memory
Main Memory
section ID = #x7
destination virtual address



A.3.3 Detail Section Definitions. The following paragraphs describe the format and uses of each section of the microcode partition. This section is intended to be required for all Explorer Lisp Microprocessor class processors.

A.3.3.1 Microload Header Section.

The Header section is used identify the microload as a microcode partition or object file. Information such as the target processor-ID, microcode assembly version number, and active length are all stored here. A checksum is provided to verify the data integrity of a microcode partition at boot time. The microload Header section must appear as the first section in the Partition. Only one microload Header section is allowed in any given microload. A Header section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x0	Identifies the Header section.
#x4	Processor ID	32 bit	Bits <15:00> contain value
#x8	Version number	32 bit	Microcode assy. version no.
#xC	Active Word Length	32 bit	Total length of active data in 32 bit words (a value of -1 indicates that the checksum is not supported)

#x10	Checksum	32 bit	Checksum of all active data (Zero if not supported)

The Processor ID is used to verify that a given microload is targeted for the booting processor. The Processor ID is a 16 bit value and resides in bits <15:00>. The Processor ID must exactly match the Processor Type field in the target processor's Configuration ROM.

The Version number is a binary value equal to the current microload assembly version. This is determined by the assembler from the last assembly for a given control file. This count begins with 1.

The Active Word Length specifies how many words (in 32 bit values) beyond the Checksum are to be processed for the load-time Checksum calculation (initial value = 0, add each word, ignoring overflow). The match must be exact.

Note that data stored in the partition beyond the end of the Active Word Length is not processed or validated by ROM boot mechanisms and must not cause the total length of the microcode partition to exceed one megabyte. An active length of -1 indicates that the checksum calculation is disabled. In this case, a value of 0 is provided for the Checksum.

A.3.3.2 Instruction Memory Section.

The Instruction Memory section contains the microcode words to be stored in Writable Control Store [WCS]. Each 64 bit Instruction Memory [I-mem] word is stored in two consecutive 32 bit words in the partition, and four consecutive 16 bit words in the MCR file (least significant bits first). The WCS Address specifies the I-Mem destination address into which the first microcode word is written. Successive I-Mem instructions are written into ascending addresses until Number of Words has been transferred. Parity for each I-Mem word is generated by the assembler and is not checked during loading.

One or more instruction memory sections must exist within a microload. However, only the last section of this type before the Entry Data section shall be loaded by the ROM load mechanism. A Microcode section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x1	Identifies the microcode section
#x4	WCS Address	32 bit	Starting microcode address
#x8	Number of Words	32 bit	Total number of 64-bit microcode words
#xC+8N	LSB of instruction	32 bit	Bits <31:00> of an instruction
#x10+8N	MSB of instruction	32 bit	Bits <63:32> of an instruction

(N ranges 0 to Number of Words)

A.3.3.3 Dispatch Section.

The Dispatch section contains data for the dispatch instructions to be stored in the internal Lisp Microprocessor Dispatch RAM. Each 18 bit dispatch word is stored in a 32 bit word in the partition. The Dispatch Address specifies the D-Mem destination address into which the first dispatch word is written. Successive D-Mem words are written into ascending addresses until Number of Words have been transferred. The upper bits of each 32-bit word in the partition, bits <31:18>, are ignored during the booting.

Zero or more Dispatch memory sections may be present in any given microload. Later sections may overwrite earlier sections. Dispatch memory sections must appear before the Entry Data section and after the Microload header. A Dispatch section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x2	Identifies a Dispatch section
#x4	Dispatch Address	32 bit	Initial dispatch address
#x8	Number of Words	32 bit	Number of dispatch words
#xC+4N	Dispatch value	32 bit	Bits <17:00> are dispatch value

(N ranges 0 to Number of Words)

A.3.3.4 A&M Memory Section.

The A&M Memory section contains the initial A&M Memory values to be stored in the internal Lisp Microprocessor A&M Memory RAM. The A&M Memory Address specifies the address for first A&M Memory word. Successive A&M Memory words are written into ascending addresses until Number of Words has been transferred. Data values with addresses 0 to #x3F are also written into M-Memory. Data values above #x3F are written into A-Memory only.

Zero or more A&M Memory sections may be present in any given microload. Later sections may overwrite earlier sections. A&M memory sections must appear before the Entry Data section and after the Microload header. An A&M memory section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x3	Identifies a A&M Memory section
#x4	A&M Memory Adress	32 bit	Starting A&M Memory address
#x8	Number of Words	32 bit	Number of A&M Memory values
#xC+4N	A&M Memory value	32 bit	Initial A&M Memory value

(N ranges 0 to Number of Words)

NOTE

On the Explorer II processor, the highest 8 A-Memory locations (#x3F8 through #x3FF) are reserved for passing system boot information to the downloaded code and are therefore not loaded from the A&M Memory section even if the A&M memory section contains values for these locations. Likewise, because their use is required during booting, locations 0-3 are also not actually initialized from the MCR. Location 0's initial value is indeterminate.

A.3.3.5 Tag Classifier Section.

The Tag Classifier section contains the Tag Classifier data values that are stored in the Lisp Microprocessor's internal Tag Classifier RAM. Each Tag Class specifies 64 bits of data (2-bits for each of 32 data types) to be stored in the Tag Classifier RAM. Tag Class data is divided into four quarters, each of which occupies the most significant 16-bits (<31:16>) of four consecutive 32 bit words in a Tag Classifier section.

The Tag Class Address specifies the Tag Classifier memory destination address for first Tag Class. Successive Tag Classes are then written into memory until Number of Classes have been written. Zero or more Tag Classifier Memory sections may be present in any given microload. Later sections may overwrite earlier sections. Tag Classifier memory sections must appear before the Entry Data section and after the Microload header. A Tag Classifier memory section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x4	Identifies a Tag Classifier Memory section
#x4	Tag Class	32 bit	Starting Tag Class
#x8	Number of Classes	32 bit	Number of Tag Classes
#x0C+10N	Tag Classifier Value	32 bit	Quarter-0 T-mem<15:00>
#x10+10N	Tag Classifier Value	32 bit	Quarter-1 T-Mem<31:16>
#x14+10N	Tag Classifier Value	32 bit	Quarter-2 T-Mem<47:32>
#x18+10N	Tag Classifier Value	32 bit	Quarter-3 T-Mem<63:48>
(N ranges 0 to Number of Classes)			(Qn = Tag Class Quad)

A.3.3.6 I/O Space Initialization Section.

The I/O Space Initialization section contains parameters to specify patterns that are to be written into the local address [I/O] space.

If bit <31> of the Insert Memory Offset word is set (=1) then at boot time the Initial Data value is adjusted by an LDB of the NuBus address of the current system memory into the Initial Data value. Bits <09:00> of the Insert Memory Offset word are used to generate the rotation length and count for the LDB. This feature, in conjunction with the Data and Address Increment values, supports initialization of the Virtual Memory Maps by the ROM mechanisms.

The (potentially adjusted) data value is written into the Starting Address. After each write the data value is adjusted by adding the Data Increment value to allow initializations with values that include zero or more incrementing fields. Also, the destination address is adjusted after each write by adding the Address Increment value. This is necessary due to the manner in which most of the arrays on processor boards are addressed in IO space. Writing and adjusting of data and addresses continues until Number of Words have been written.

Zero or more I/O Space Initialization sections may be present in any given microload. Later sections may overwrite earlier sections. I/O Space Initialization sections must appear before the Entry Data section and after the Microload header. An I/O Space Initialization section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x5	Identifies an I/O Space Initialization section
#x4	Starting Address	32 bit	I/O Space destination address
#x8	Number of words	32 bit	Number of 32-bit words to write
#xC	Initial Data	32 bit	Initial data value for writes
#x10	Insert Memory Offset	32 bit	Bit <31> : i= enable insertion Bit <09:05> : rotation length Bit <04:00> : rotation count
#x14	Data Increment	32 bit	Add to data value after each write
#x18	Address Increment	32 bit	Add to address value after each write

A.3.3.7 I/O Space Data Section.

The I/O Space Data section contains data that are used to load the various memories in I/O space. This can be used to initialize the Transport RAM and other IO space features. The Starting Address is an absolute Lisp Microprocessor IO space address. Data Values are copied from the section into IO space. After each write, the destination address is adjusted by adding the Address Increment value. This is necessary due to the manner in which most of the arrays on processor boards are addressed in IO space. Writing and adjusting of addresses continues until Number of Words have been written.

Zero or more I/O Space Data sections may be present in any given microload. Later sections may overwrite earlier sections. I/O Space Data sections must appear before the Entry Data section and after the Microload header. An I/O Space Data section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x6	Identifies an I/O Space Data section
#x4	Starting Address	32 bit	Destination I/O address
#x8	Number of words	32 bit	Number of 32 bit words to write
#xC	Address Increment	32 bit	Add to destination address value after each write
#x10+4N	Data Value	32 bit	Data values to be written

(N ranges 0 to Number of Words)

A.3.3.8 Main Memory Section.

The Main memory section contains data values to be stored in this processor's main memory. The Source Data Values are stored as 32-bit values. The Destination Address is adjusted from a virtual (word) to a physical (byte) value and then added to the NuBus address of the system memory determined by the booting mechanism (the System Test and Boot Master or utilities such as Spyport software which also load MCR's) to belong to this processor. Successive words of Source Data Values are transferred sequentially until Number of words have been written.

Zero or more Main Memory sections may be present in any given microload. Later sections may overwrite earlier sections. Main Memory sections must appear before the Entry Data section and after the Microload header. A Main Memory section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#x7	Identifies a Main Memory section
#x4	Destination Address	32 bit	Virtual (word) address
#x8	Number of 32-bit words	32 bit	Number of words to write
#xC+4N	Data Value	32 bit	Data

(N ranges 0 to Number of Words)

A.3.3.9 Auxiliary Data Section.

The Auxiliary Data section contains processor specific data. The use of this data will be defined by individual processor boot mechanism and system specifications. The Sub ID code allows a processor to have more than one type of Auxiliary section. The section contains Number of Words of Data Values whose format and use are processor dependent.

The Explorer II processor supports a Required Floating Point Processor Auxiliary Data section (sub-id type = 1). If this section is present in a partition, then the first data word of this section must exactly match the type floating point option present. Floating point option boards identify themselves via a three bit field in the Explorer II External Control Register. The Required Floating Point Processor Auxiliary Data section must be used to prevent microcode which requires a specific floating point option board from being loaded, only to crash when a different option board (or more likely none at all) is present. Additional Auxiliary Data values beyond the first word are ignored, as are Auxiliary Data sections of other sub-id types. They may therefore transport various types of data specific to a MCR partition (such as crash table or build information) but not needed during booting.

Zero or more Auxiliary Data sections may be present in any given microload. Auxiliary Data sections must appear before the Entry Data section and after the Microload header. An Auxiliary Data section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#xA	Identifies a Auxiliary Data section
#x4	Sub ID	32 bit	Differentiates sub sections (processor dependent)
#x8	Number of words	32 bit	Number of 32 bit values in remainder of section
#xC+4N	Data Values	32 bit	Data values

(N ranges 0 to Number of Words)

A.3.3.10 Entry Data Section.

The Entry Data section serves as a marker to indicate the end of the microload. It contains the processor's initial Micro PC and initial Machine Control Register values. Note that both these values are required, as they shall be set as the starting state of the Lisp Microprocessor as the last act of the booting mechanisms.

Exactly one Entry Data section must be present in a partition and it must appear after all other sections which require loading at boot time. Additional sections may be present following the Entry Data section to carry auxiliary information about each particular microload. The Entry Data section contains the following fields:

byte offset	function	value or format	description
#x0	Section ID	#xEOF	Identifies the Entry Data section
#x4	Micro PC	32 bit	Initial Micro Program Counter
#x8	MCR	32 bit	Initial Machine Control Register value

APPENDIX B

Acronyms

B.1 ACRONYMS USED

Acronyms for system structures and routine names are introduced at various points throughout the manual. If you read a section from the manual without reading all preceding sections, an acronym may be encountered without an explanation of its meaning. Table B-1 lists most of the acronyms used in the manual. Refer to this list for a complete description of the term.

Table B-1 Description of Acronyms

Acronym	Meaning
ADL	Argument Descriptor List
DDR	Device Driver
DPMT	Disk Page Map Table
ETE	Error Table Entry
EVCP	External Value Call Pointer
FEF	Function Entry Frame
FMT	Format Information Partition
LAN	Local Area Network
MCR	Machine Control Register
NUPI	NuBus Peripheral Interface
NVRAM	Non-Volatile Random Access Memory
PHT	Page Hash Table
PPD	Physical Page Data
RQB	Request Block
SIB	System Interface Board
UPCS	~~Processor Micro-Stack~~
STBM	System Test and Boot Master
TZON	Test Zone Partition
VMA	Virtual Memory Address
WCS	Writable Control Store