

EXPLORER™ SYSTEM SOFTWARE DESIGN NOTES

MANUAL REVISION HISTORY

Explorer™ System Software Design Notes (2243208-0001)

Original IssueJune 1985

© 1985, 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 computers, as well as the programs that TI has created to use with them, are tools that can help people better manage the information used in their business; but tools—including TI computers—cannot replace sound judgment nor make the manager’s business decisions.

Consequently, TI cannot warrant that its systems are suitable for any specific customer application. The manager must rely on judgment of what is best for his or her business.

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
Data Systems Group
P.O. Box 2909 M/S 2151
Austin, Texas 78769

Printed in U.S.A.

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, followed by a description of elements of the Explorer Lisp machine virtual architecture, followed by several sets of material useful to systems programmers. The information in this version is sketchy in some places; it will be better detailed in future versions.

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 and Paging - Describes how the mapping and paging schemes work on the Explorer.

Section 6 -- Internal Storage Formats - Details the formats used for representing Lisp data items.

Section 7 -- Storage Management - Explains how memory is organized into areas and regions, and describes several specific areas used by the system software.

Section 8 -- Garbage Collection - Describes the techniques and structures used by the Explorer for handling garbage collection.

Section 9 -- Function Calling - Details the mechanisms used for function calling and the internal formats of compiled code.

Section 10 -- Flavors - Discusses the data structures used for handling flavors.

Section 11 -- Stack Groups - Describes the stack group data structure and the special push down list.

Section 12 -- Subprimitives - Explains the functionality and lists the arguments for low level system functions.

Section 13 -- Low-Level Lisp Data Structures - Details how to use encapsulation and some features of defstruct.

Section 14 -- Error Handling - Describes microcode error handling and explains the various error conditions that can arise.

Section 15 -- Crash Handling - Explains how crash records are organized and how you can examine them from Lisp.

Section 16 -- Compiler Notes - Discusses cross-compilation and compile-time properties of symbols.

Appendix A -- Data Structures - Details the content of the non-volatile RAM and the disk partition structures.

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

Contents

Paragraph	Title	Page
-----------	-------	------

1 -- Hardware Overview

1.1	MAJOR COMPONENTS	1-1
1.1.1	System Enclosure	1-1
1.1.2	Brief Descriptions of Each Board	1-1
1.1.3	Monitor	1-2
1.1.4	Mass Storage Subsystem	1-2
1.2	INTRASYSTEM COMMUNICATION	1-3
1.2.1	NuBus	1-4
1.2.1.1	Block Moves	1-4
1.2.1.2	Time-Out Period	1-4
1.2.1.3	Parity	1-4
1.2.1.4	Reset Operation	1-5
1.2.1.5	Power Fail	1-5
1.2.1.6	System Addressing	1-5
1.2.1.7	Configuration (ID) ROM	1-5
1.2.1.8	Bus Locking	1-5
1.2.1.9	NuBus Clock	1-6
1.2.1.10	Unimplemented Memory	1-6
1.2.1.11	Byte, Halfword, and Word Transfer Protocols	1-6
1.2.2	Explorer Bus (Local Bus)	1-6
1.2.3	I/O Connectors	1-7
1.3	INTERSYSTEM COMMUNICATION	1-7
1.4	EXPLORER SYSTEM DEFINED ADDRESSES	1-7

2 -- Bootstrap Loading

2.1	INTRODUCTION	2-1
2.2	TYPES OF LOADS	2-1
2.3	POWER ON / SYSTEM RESET	2-2
2.4	CONFIGURATION ROM	2-8
2.5	THE STBM AND DEVICE INDEPENDENCE	2-10
2.6	NVRAM FORMAT	2-11
2.7	ERROR CODES	2-11
2.7.1	STBM ROMS - Error Codes	2-11
2.7.2	MENU BOOT - Error Codes	2-13
2.8	LIGHT CODE TABLE	2-13
2.8.1	STBM Light Codes	2-14

Paragraph	Title	Page
2.8.2	MENUBOOT Light Codes	2-14
2.9	MICROLOADS	2-15
2.10	INTERFACE BETWEEN BOOT PROM AND MICROLOADS	2-15

3 -- Interrupts

3.1	INTRODUCTION	3-1
3.2	INTERRUPTS ON THE EXPLORER PROCESSOR	3-2

4 -- Device Handling

4.1	INTRODUCTION	4-1
4.2	DEVICE DECODING	4-1
4.3	DEVICE DETAILS	4-2
4.3.1	The NuBus Peripheral Interface Board	4-2
4.3.2	The Keyboard	4-3
4.3.3	TV Vertical Retrace (Interrupt)	4-4
4.4	SERIAL AND PARALLEL IO PORTS	4-4
4.4.1	RS232 Serial Port	4-4
4.4.2	Parallel Port	4-5
4.5	DISK AND TAPE STRUCTURES	4-5
4.5.1	Volume Label Partition	4-6
4.5.2	Partition Table Partition	4-6
4.5.3	Test Zone Partition	4-7
4.5.4	Format Information Partition	4-7

5 -- Virtual Memory and Paging

5.1	VIRTUAL MEMORY	5-1
5.2	PHYSICAL ADDRESSES	5-2
5.3	VIRTUAL ADDRESSES	5-2
5.4	PHYSICAL MEMORY USE	5-3
5.5	MEMORY MAP HARDWARE	5-3
5.6	VIRTUAL MEMORY SYSTEM TABLES	5-4
5.6.1	Page Hash Table	5-4
5.6.2	Physical Page Data Table	5-4
5.6.3	Reverse First Level Map Table	5-5
5.7	SECOND LEVEL MEMORY MAP BLOCK ALLOCATION	5-5
5.8	PDL BUFFER HANDLING	5-6
5.9	VIRTUAL PAGE MANAGEMENT	5-6

Paragraph	Title	Page
5.9.1	Page Aging Process	5-7
5.9.2	Page Hash Table	5-7
5.9.3	Disk Page Mapping Scheme	5-10
5.9.4	Page Swapping	5-12
5.9.5	Logical Paging Devices	5-12

6 -- Internal Storage Formats

6.1	INTRODUCTION	6-1
6.2	Q FORMAT	6-2
6.2.1	CDR Code	6-2
6.2.2	Data Type	6-3
6.2.2.1	Data Type 0 - DTP-TRAP	6-3
6.2.2.2	Data Type 1 - DTP-NULL	6-3
6.2.2.3	Data Type 2 - DTP-FREE	6-4
6.2.2.4	Data Type 3 - DTP-SYMBOL	6-4
6.2.2.5	Data Type 4 - DTP-SYMBOL-HEADER	6-4
6.2.2.6	Data Type 5 - DTP-FIX	6-4
6.2.2.7	Data Type 6 DTP-EXTENDED-NUMBER	6-4
6.2.2.8	Data Type 7 DTP-HEADER	6-4
6.2.2.9	Data Type 8 DTP-GC-FORWARD	6-4
6.2.2.10	Data Type 9 DTP-EXTERNAL-VALUE-CELL-POINTER	6-4
6.2.2.11	Data Type 10 DTP-ONE-Q-FORWARD	6-5
6.2.2.12	Data Type 11 DTP-HEADER-FORWARD	6-5
6.2.2.13	Data Type 12 DTP-BODY-FORWARD	6-5
6.2.2.14	Data Type 13 DTP-LOCATIVE	6-5
6.2.2.15	Data Type 14 DTP-LIST	6-5
6.2.2.16	Data Type 15 DTP-U-ENTRY	6-5
6.2.2.17	Data Type 16 DTP-FEF-POINTER	6-6
6.2.2.18	Data Type 17 DTP-ARRAY-POINTER	6-6
6.2.2.19	Data Type 18 DTP-ARRAY-HEADER	6-6
6.2.2.20	Data Type 19 DTP-STACK-GROUP	6-6
6.2.2.21	Data Type 20 DTP-CLOSURE	6-6
6.2.2.22	Data Type 21 DTP-SMALL-FLONUM	6-6
6.2.2.23	Data Type 22 DTP-SELECT-METHOD	6-6
6.2.2.24	Data Type 23 DTP-INSTANCE	6-6
6.2.2.25	Data Type 24 DTP-INSTANCE-HEADER	6-6
6.2.2.26	Data Type 25 DTP-ENTITY	6-7
6.2.2.27	Data Type 26 DTP-STACK-CLOSURE	6-7
6.2.2.28	Data Type 27 DTP-SELF-REF-POINTER	6-7
6.2.2.29	Data Type 28 DTP-CHARACTER	6-7

Paragraph	Title	Page
6.2.2.30	Data Type 29 - DTP-FEF-HEADER	6-7
6.2.2.31	Data Type 30 THROUGH 31	6-7
6.2.3	Pointer	6-7
6.3	STRUCTURE HEADERS	6-7
6.4	INVISIBLE FORWARDING POINTERS	6-8
6.5	IN THE MACHINE	6-9
6.6	SYMBOLS	6-9
6.7	ARRAYS	6-10
6.7.1	Has Leader	6-11
6.7.2	Displaced	6-12
6.7.3	Number of Dims	6-13
6.7.4	Long Length Flag	6-13
6.7.5	Named Flag	6-13
6.7.6	Index Length	6-14
6.7.7	Array Type	6-14
6.8	DTP-SELF-REF-POINTER FORMAT	6-14
6.9	PDL FORMAT	6-15
6.10	FEF FORMATS	6-16
6.11	FLOATING POINT FORMATS	6-16
6.12	BIGNUM FORMAT	6-18
6.13	CHARACTER FORMAT	6-19
6.14	CLOSURE FORMATS	6-19

7 -- Storage Management

7.1	INTRODUCTION	7-1
7.2	AREAS	7-1
7.3	REGIONS	7-2
7.4	STANDARD AREAS	7-5
7.5	SYSTEM COMMUNICATION AREA	7-9

8 -- Garbage Collection

8.1	INTRODUCTION	8-1
8.2	IN THE MACHINE	8-1
8.3	THE READ BARRIER	8-2
8.4	INCREMENTAL GC	8-3
8.4.1	Scavenging	8-4
8.4.2	Shared Objects	8-5
8.4.3	Areas	8-5
8.4.4	Flipping	8-6
8.5	THE WRITE BARRIER	8-6

Paragraph	Title	Page
-----------	-------	------

9 -- Function Calling

9.1	FUNCTIONAL OBJECTS	9-1
9.1.1	DTP-LIST Functions	9-2
9.1.2	DTP-SYMBOL Functions	9-2
9.1.3	DTP-FEF-POINTER Functions	9-2
9.1.4	DTP-U-ENTRY Functions	9-2
9.1.5	DTP-ARRAY-POINTER Functions	9-2
9.1.6	DTP-STACK-GROUP Functions	9-3
9.1.7	DTP-INSTANCE Functions	9-3
9.1.8	DTP-CLOSURE Functions	9-3
9.1.9	DTP-ENTITY Functions	9-3
9.1.10	DTP-STACK-CLOSURE Functions	9-3
9.1.11	DTP-LOCATIVE Functions	9-3
9.2	PDL LAYOUT	9-4
9.3	FUNCTION CALLING	9-4
9.4	FEF LAYOUT	9-5
9.4.1	FEF Header	9-6
9.4.2	FEF Storage Length	9-6
9.4.3	FEF Name	9-6
9.4.4	Numeric Argument Descriptor	9-6
9.4.5	Special Variable Bit Map	9-7
9.4.6	FEF Miscellaneous	9-7
9.4.7	Special Value Cell Pointers	9-8
9.4.8	Optional FEF Header Words	9-8
9.4.9	Function Entry	9-9
9.4.10	ARG Descriptor List	9-9
9.4.11	FEF Specialness	9-11
9.4.12	Desired Data Type	9-11
9.4.13	Quote Status	9-11
9.4.14	Argument Syntax	9-12
9.4.15	Init Option	9-12
9.4.16	FEF Constants	9-13

10 -- Flavors

10.1	INTRODUCTION	10-1
10.2	INSTANCE DATA STRUCTURE	10-1
10.3	INSTANCE DESCRIPTOR DATA STRUCTURE	10-1
10.4	SELF MAPPING TABLE	10-3
10.5	METHOD DECODE TABLE	10-4
10.6	CALLING AN INSTANCE	10-4
10.7	INSTANCE VARIABLE ACCESSING	10-5

Paragraph	Title	Page
11 -- Stack Groups		
11.1	INTRODUCTION	11-1
11.2	THE STACK GROUP DATA STRUCTURE	11-1
11.3	SPECIAL PDL	11-2
12 -- Subprimitives		
12.1	SUBPRIMITIVES	12-1
12.2	DATA TYPES	12-2
12.3	FORWARDING	12-3
12.4	POINTER MANIPULATION	12-5
12.5	SPECIAL MEMORY REFERENCING	12-6
12.6	ARRAY SUBPRIMITIVES	12-10
12.7	STORAGE LAYOUT DEFINITIONS	12-12
12.8	ANALYZING STRUCTURES	12-13
12.9	CREATING OBJECTS	12-15
12.10	STACK LIST SUBPRIMITIVES	12-15
12.11	COPYING DATA	12-16
12.12	RETURNING STORAGE	12-18
12.13	LOCKING SUBPRIMITIVE	12-18
12.14	EXPLORER I/O DEVICE SUBPRIMITIVES	12-19
12.15	FUNCTION-CALLING SUBPRIMITIVES	12-19
12.16	SPECIAL-BINDING SUBPRIMITIVE	12-21
12.17	CLOSURE SUBPRIMITIVES	12-21
12.18	PAGING SYSTEM SUBPRIMITIVES	12-22
12.19	SUBPRIMITIVES TO SHUT DOWN THE LISP ENVIRONMENT	12-26
12.20	DISTINGUISHING PROCESSOR TYPES	12-26
12.21	MICROCODE VARIABLES	12-27
12.22	MICROCODE METERS	12-30
13 -- Low-Level Lisp Data Structures		
13.1	INTRODUCTION	13-1
13.2	ENCAPSULATION FUNCTIONS	13-2
13.3	DEFSTRUCT DESCRIPTION	13-7
13.4	MORE EXTENSIONS TO DEFSTRUCT	13-8

Paragraph	Title	Page
-----------	-------	------

14 -- Error Handling

14.1	INTRODUCTION	14-1
14.2	MICROCODE ERROR CONDITIONS	14-1
14.2.1	Microcode Error Table	14-3
14.2.1.1	Special Error Table Entries	14-4
14.2.1.2	Normal Error Table Entries	14-4
14.3	ILLEGAL OPERATION	14-9

15 -- Crash Handling

15.1	CRASH RECORDING	15-1
15.1.1	Crash Record Allocation	15-1
15.1.2	Crash Record Contents	15-3
15.1.3	Crash Analyzer	15-6

16 -- Compiler Notes

16.1	CROSS-COMPILATIONS	16-1
16.2	COMPILE-TIME PROPERTIES OF SYMBOLS	16-3
16.3	XFASL FILES	16-5

Appendixes

Appendix	Title	Page
A	DATA STRUCTURES	A-1
B	ACRONYMS	B-1

Illustrations

Figure	Title	Page
1-1	Backplane Configuration	1-3
1-2	Layout of Words, Halfwords, and Bytes	1-4
1-3	System-Defined Addresses	1-9
3-1	Explorer Control Space	3-2
3-2	Device Descriptor Block	3-3
3-3	Interrupt Vector Table	3-3
4-1	NUPI Device Descriptor Block	4-3
5-1	PDL Buffer Wrap Around	5-6
5-2	Page Hash Table Sizes	5-8
5-3	Page Hash Table Entry Format	5-8
5-4	Disk Page Mapping Table	5-11
5-5	Device Status Codes	5-11
5-6	Logical Page Device Information Block	5-12
6-1	Q Format	6-2
6-2	Array Header Word	6-11
6-3	Array with Leader	6-12
6-4	Small Flonum Format	6-16
6-5	Flonum Structure Format	6-17
6-6	Flonum Header Format	6-17
6-7	Flonum Mantissa Format	6-17
6-8	Bignum Structure	6-18
6-9	Bignum Header Format	6-18
6-10	Bignum Data Format	6-19
6-11	Character Format	6-19
7-1	Region Bits Area Entry Description	7-4
7-2	Map of Systems Communications Area	7-9
9-1	Call Block	9-4
9-2	FEF Header Fields	9-6
9-3	FEF Fast Argument Option Fields	9-7
9-4	FEF Misc Word	9-8
9-5	ADL First Q	9-10

Tables

Table	Title	Page
1-1	System-Defined Addresses	1-8
5-1	Physical Page Data Area Word Format	5-5
5-2	Page Hash Table: Swap Status Codes	5-9
6-1	CDR Codes	6-3
6-2	Header Types	6-8
6-3	Qs of Symbol	6-9
6-4	Array Types	6-14
6-5	SELF-REF-POINTER Format	6-15
7-1	Space Type Codes	7-4
9-1	FEF Specialness	9-11
9-2	Desired Data Type	9-11
9-3	Quote Status	9-12
9-4	Argument Syntax	9-12
9-5	Quote Status	9-13
10-1	Instance Descriptor Offsets	10-2
11-1	Special PDL Block Type	11-3
15-1	Crash Record Allocation Registers	15-3
15-2	Crash Record Format	15-4
B-1	Description of Acronyms	B-1

SECTION 1

Hardware Overview

1.1 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.

1.1.1 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.

1.1.2 Brief Descriptions of Each Board.

Explorer processor

- 500-IC microcoded Lisp engine
- two-level, demand-paged virtual memory map
- includes self test ability
- private bus (local bus) to Explorer memory board

Explorer memory board

- 2mb, 4mb, 8mb versions with byte parity
- two ports - NuBus and the Explorer private bus
- interfaces Explorer processor to NuBus
- 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 2 Kbyte 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.
- controls monitor, keyboard and mouse
- provides RS-232C port and an eight-bit parallel port
- ROM contains 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 contains board identification information, diagnostics, and system load device driver.

1.1.3 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 curser movement and positioning using a 200 dots per inch optical pad.

1.1.4 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.

1.2 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.

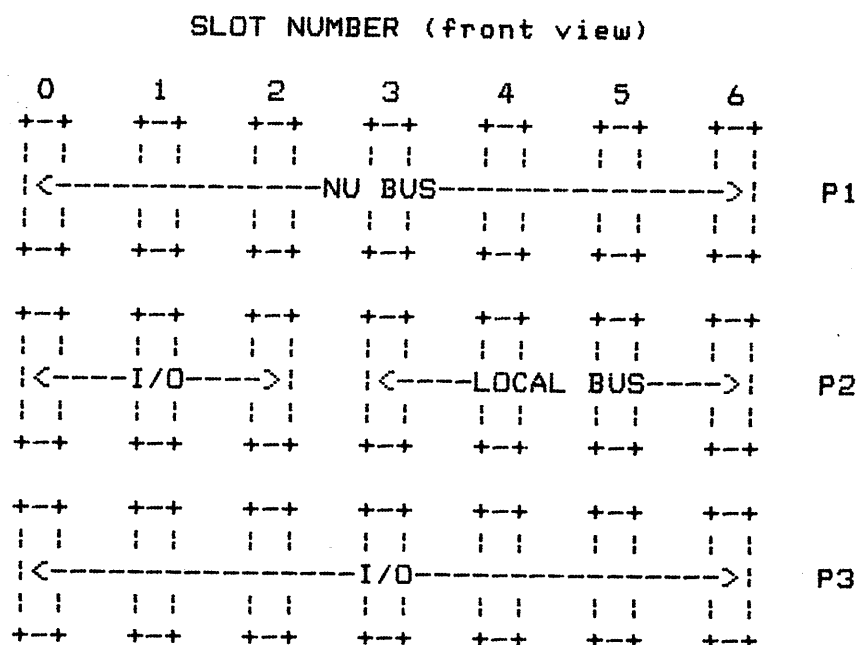


Figure 1-1 Backplane Configuration

1.2.1 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.

Bytes, halfwords, and words are organized as shown in Figure 1-2.

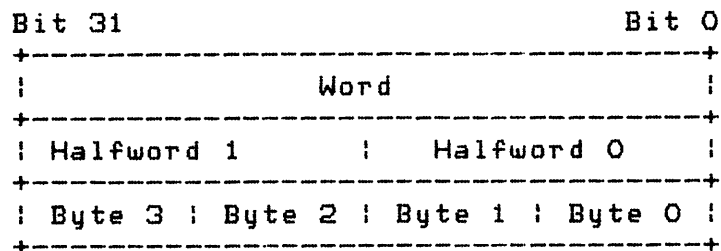


Figure 1-2 Layout of Words, Halfwords, and Bytes

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.

1.2.1.1 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.

1.2.1.2 Time-Out Period.

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

1.2.1.3 Parity.

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

1.2.1.4 Reset Operation.

On power-up, the source that drives the RESET/ signal (the System Interface Board) 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-test 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 selftest and the system will reboot. RESET/ must be low for two or more clock cycles to force the system restart.

1.2.1.5 Power Fail.

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

1.2.1.6 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.

1.2.1.7 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.

1.2.1.8 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.

1.2.1.9 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.

1.2.1.10 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

1.2.1.11 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.

1.2.2 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.

1.2.3 I/O Connectors.

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

1.3 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.

1.4 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	All hardware-decoded interrupt events, except power fail, will fall within these addresses.
!!	
F6F0003C	The byte written to any interrupt even address must contain a 1 in bit 1. 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	Configuration ROM, first 64 bytes. Larger configuration ROMs will occupy the contiguous space below these addresses.
!!	
FSFFFFFF	
F(SIB)Fxxxxx	Bus timeout address. Nothing shall respond to this address, so a read or write at this address tests timeout generation.

The address spaces defined in the configuration ROM are illustrated in Figure 1-3.

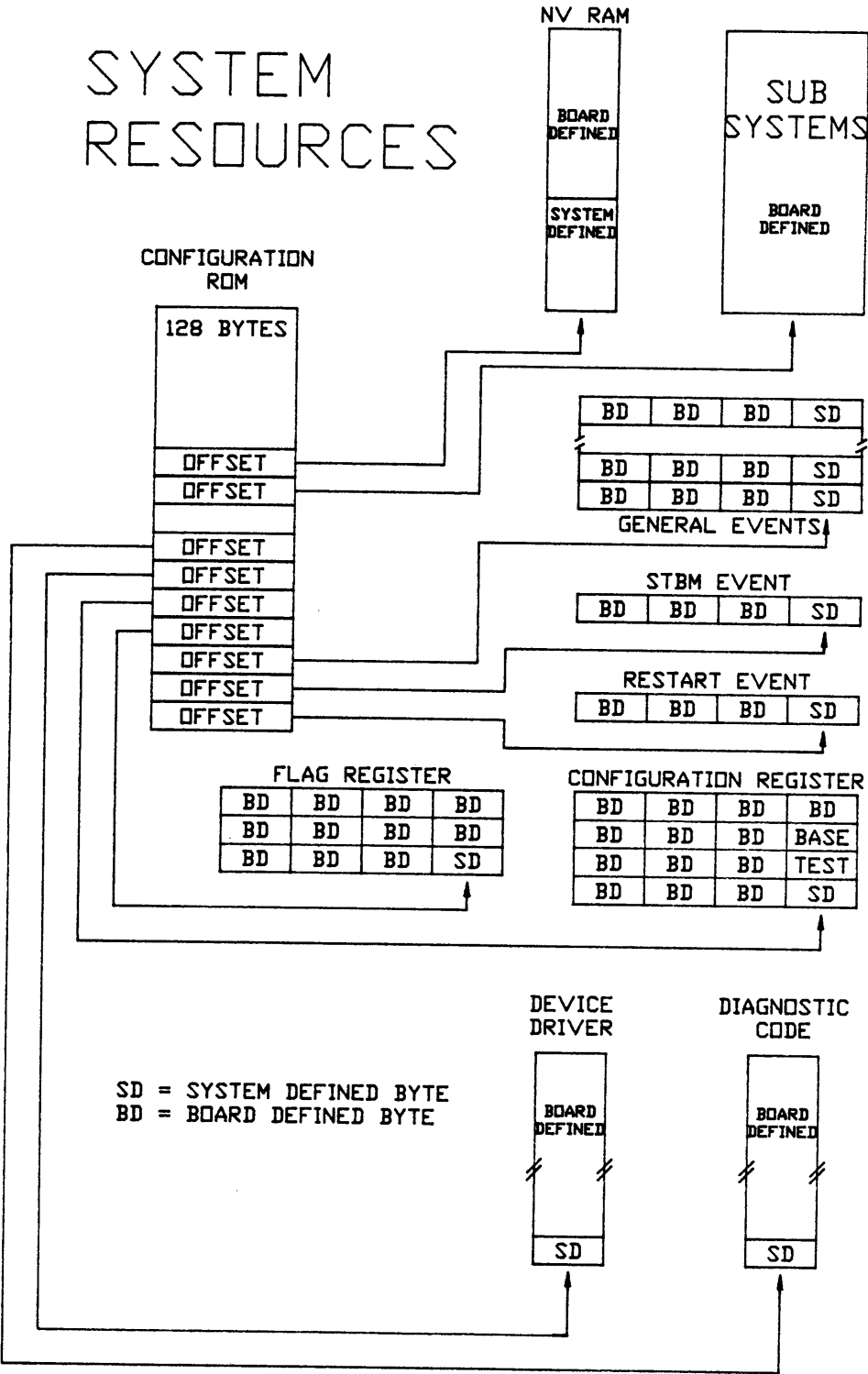


Figure 1-3 System-Defined Addresses

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:

1. 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 chasis. A system reset is performed by issuing the boot 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 entities that get loaded are the system microcode and the system load. The system 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 load (the LISP system software) from a load band on secondary storage.
2. Cold boot. This is performed by issuing the boot 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.
3. Warm boot. This is performed by issuing the boot 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.

4. Menu boot. This is performed by issuing the boot 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 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. 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 PROM at a fixed location. The first action it takes is to perform the processor self-test.

The Explorer processor self-test tests the following:

1. A & M memories
2. ALU operations
3. Arithmetic operations
4. G-register shifting
5. Internal registers
6. Microprogram stack
7. Test dispatch memory
8. Level 1 and level 2 memory maps
9. I-memory (read/write, parity errors)
10. Tag classifier RAM
11. PDL control logic
12. Internal traps

13. Microsequencer functions

14. Events

Then all internal memories and registers are cleared.

If the processor fails self-test, it will crash with a light code #x89. See light codes table.

System Test - if the processor passes its self-test, it proceeds to the next phase of the booting process, System Test.

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 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:

(Unless stated otherwise, 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).

- a. First check 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.
- b. Search for Non-volatile RAM (NVRAM). If the NVRAM bit of the Resource Type field is set, then the board contains NVRAM. The Explorer does not check the CRC on the contents of the NVRAM, but it does check the format generation number to verify it contains a value of #x01. See the paragraph on NVRAM for further details. This is a partial validation of NVRAM contents.

- c. 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 crashes with a light code = #x8E (see light codes table).
- d. Display the message "Slot S TESTING SYSTEM", where S is the slot number of the processor board, on the monitor.
- e. 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 one megabyte of memory. If no valid memory board can be found, the message "ERROR: 00000004" is displayed on the monitor, and the processor crashes with a light code = #x8A.
- f. 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 #x0F. If any test fails, the rest of the tests for that board are skipped.
 - Test if 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.
 - Display "Slot S" on the monitor, where S is the slot number of the board under test.

- Test if the Configuration ROM contents are valid: ID character = #xC3 and CRC verifies. If not, display "ROM" ("TESTS FAILED" will be 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".
 - 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".
 - If the Configuration ROM Diagnostic Offset field is not = #FFFFFFF, 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".
- g. 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 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, but not the LAN bit in the Resource type field set, and then using the lowest numbered unit at that slot. The microcode and load band specified as defaults 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 20 seconds and no boards have failed Chassis Test, the processor does a default load of the banks marked as default load and MCR. 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 load sequence, described below.
4. At this point there are three "hidden options" in addition to the four already mentioned. They are hidden because they are intended for 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 "Q" 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.

Menu Load - 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 microload from the device you selected. If you reached the device selection menu by entering a "hidden option" at the initial menu, the microcode that is loaded depends on which option you selected. If you reached this point by typing in an "M" at the Initial Menu, the processor will load 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.

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

1. Normally you will want to boot up LISP, so you enter "L" or simply press RETURN, since "L" is the default. You will then be presented with a menu of available load bands on the device you have selected (previously). The default load band will have an asterisk to the left of the letter that selects that band. If there is no load band with an asterisk, there is no default load band on that boot unit. If the unit is a disk you can select a default load band and microcode by using the form (si:edit-disk-label) once the LISP system is running.

Once you have selected a load band, you will be presented with a menu of available microcode bands. The one with an asterisk by it is the default microcode

as specified in the partition table on that device. 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. You can tell when the system microcode has started execution because the display will turn black for a few seconds.

2. 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.
3. 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.
4. 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, similar 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 NuBus physical addresses FSFFFF00 - FSFFFFF0, where S is the slot number of the board. Configuration ROMs are addressed as though the data is stored one byte per word. (Some boards have extended configuration ROMs that may start as low as FSFFFE6C, but this is of no concern here.)

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 contains 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 >C3, 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 16 bit field which contains (among other information) the following:

Bit 2 Test LED - A one turns on the red LED on the board.
 A zero turns it off.

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 dnterface diagnostic in their ROM which is written in Diagnostic Engine code.

There are three type of DDRs: boot source, monitor, and keyboard. They provide the processor with a generic interface to the three peripheral resources it requires to perform a boot. The monitor and keyboard allow the processor to communicate with the operator, and the boot source provides a source from which to load code. A boot source is currently defined to be a disk, tape, or LAN controller, but 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

2.7 ERROR CODES

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 NUPI or SIB ROM when an unexpected NUBUS error occurred.

DEVICE ERROR: 00000006 - 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: 00000010 - 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 - Error Codes.

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.

2.8 LIGHT CODE TABLE

In some cases the processor can not proceed and can not display a message. In these cases the processor will crash and display a code in the eight amber colored lights 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 with the topmost amber light as the most significant bit and the lowest amber light as the least significant bit. The codes shown below are hexadecimal.

2.8.1 STBM Light Codes.

- 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 can not 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 can not find a monitor on which to display the message "BAD MICROLOAD FORMAT"
- 8D = DEI PROBLEMS. This will only occur if the processor can not 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 MENUBOOT Light Codes.

- 8F = Unable to initialize monitor. The monitor device driver returned a non-zero completion code on the Initialize-monitor call.
- 90 = Unable to initialize keyboard. The keyboard device driver returned a non-zero completion code on the Initialize-keyboard call.

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. Its format provides for the loading of I-mem, A/M memories, D-mem, and main memory.

2.10 INTERFACE BETWEEN BOOT PROM AND MICROLOADS

When a microload is loaded by the boot PROM, 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 PROM 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 PROM sets up the following locations as parameters passed to a microload that has just been loaded:

a-boot-memory is set to FS000000 where S is the slot number of the first memory board found.

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 that was selected if the hidden "N" option was used. The format is ASCII little endian. If a load band was not selected, this word will contain a value of binary zero.

The boot PROM will only set up a-boot-command-block if the Explorer processor is being booted as a secondary processor.

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 any future WCS microcode) to load another microload into WCS, there is a special entry point in the PROM that WCS microcode can jump to. The following A memory locations must be set up: a-boot-memory, a-boot-monitor, a-boot-keyboard, a-boot-device, a-boot-mcr-name. Then a jump is performed to PROM location #x1E. In addition, you must turn off the bus-error-trap-enable bit in the machine control register (MCR) before jumping into the PROM.

SECTION 3

Interrupts

3.1 INTRODUCTION

Hardware event signals (interrupts) are the lowest level of mechanisms in the Lisp Machine 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 implementation (microcode) without reference to virtual memory, garbage collected memory, or macro instructions. (Virtual memory service for map handling is considered very low level and is permitted in the interrupt context.)

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 (that is, not allowed to be relocated) so that there is no chance that the garbage collector or page fault handler can be invoked by the interrupt handler. Memory references made by interrupt handlers must be to wired pages or to the physical address space.

A check for pending interrupts is made at most memory operations, especially instruction fetch. Interrupts may also be checked at other times during internal processing, but usually are not. 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 system is NuBus based, most interrupts are events signaled over the system bus by writing a word 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 Figure 3-1

NU BUS 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

Figure 3-1 Explorer Control Space

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 when it is convenient 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 Figure 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. 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 3-2.

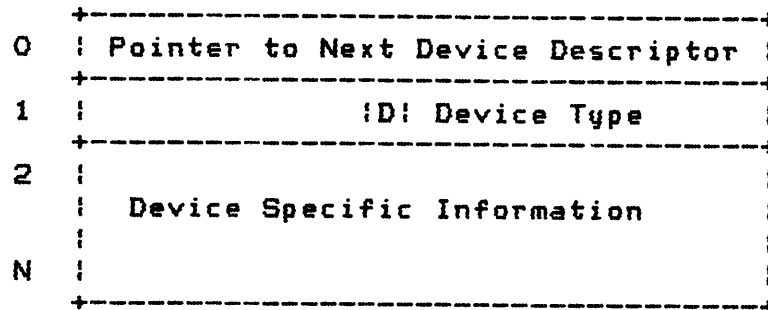


Figure 3-2 Device Descriptor Block

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 Figure 3-3.

Interrupt Priority Level	
0	Powerfail (empty vector)
1	Boot Request (empty vector)
2	Device Descriptor Block Address
.	.
15	Device Descriptor Block Address

Figure 3-3 Interrupt Vector Table

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 interrupt field of the MCR 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 chapter 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 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 (RGB) 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 follows. 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 as a fixnum. In this case the single word specifies the operation being requested. The second form is as 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.

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

4.3 DEVICE DETAILS

4.3.1 The NuBus Peripheral Interface Board.

The NuBus Peripheral Interface (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 may have a request in process. The device handler maintains a request queue for each device. The request at the front of the queue is being processed. When a request comes to the front of the queue the Busy Bit 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 map 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.

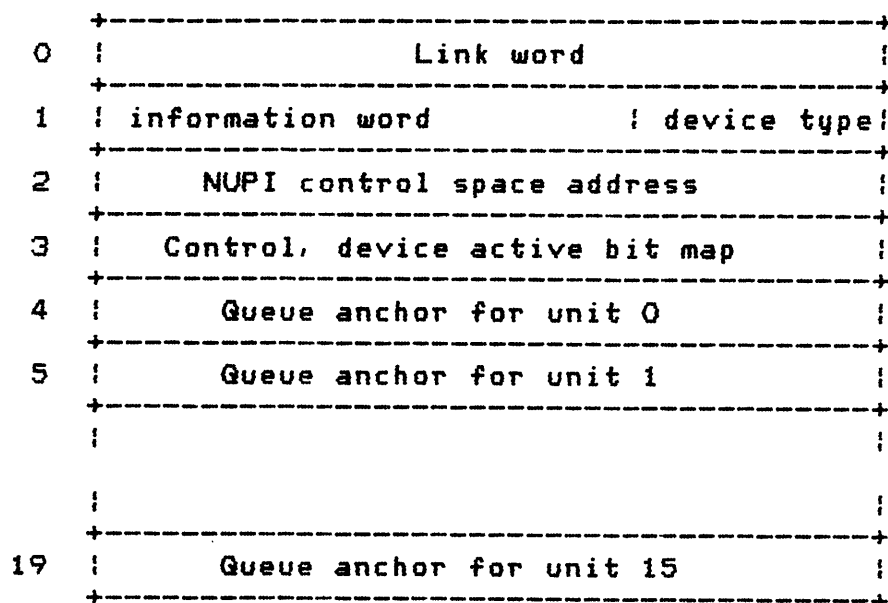


Figure 4-1 NUPI Device Descriptor Block

4.3.2 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.

4.3.3 TV Vertical Retrace (Interrupt).

An interrupt occurs when the TV begins the vertical retrace interval. At this time the mouse is checked to see if it has moved or 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 notices either mouse motion or mouse buttons. This process handles the timer, too.

4.4 SERIAL AND PARALLEL IO PORTS

The Explorer has one serial port for bidirectional asynchronous communication, and one Centronix standard unidirectional parallel port for high speed data transmission. Each has microcode support and can be used with a standard stream interface.

4.4.1 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 7 read registers. For details on the function of these registers see the Z8539 SCC Technical 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 `si:make-serial-stream`. Characters are transmitted with `:TYO` and received with `:TVI`. For more information on serial streams, see the Explorer Programming Concepts and Tools Manual.

4.4.2 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.

4.5 DISK AND TAPE STRUCTURES

There are several standard partitions (contiguous sections of the allocatable media) defined. These include the volume label, Partition Table, Save Partition, Test Zone, and Format

Information partition. The following sections briefly describe the function of each.

4.5.1 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 blocks 0 and 1 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.

4.5.2 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 table format information is the table of individual entries describing each partition. Each entry in the table has a set of partition "key" characteristics. These include:

1. Name of the partition
2. Partition Type
3. Partition User Type
4. 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 assume 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 infer 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.

4.5.3 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.

4.5.4 Format Information Partition.

The Format Information Partition (FMT) contains details about the way the volumemedia 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 specifier.

SECTION 5

Virtual Memory and Paging

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 heirarchy (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.

Memory and peripheral control registers reside within the same 32-bit address space. Not all bus addresses will be accessible directly from Lisp (in the virtual address space). This is true because the 25-bit virtual address is smaller than the 32-bit NuBus address space.

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 17 bits of the virtual address, and page offset is the low order 8 bits of the virtual address. Thus, each page contains 256 words (or 1024 bytes) of storage.

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.

The virtual page number is looked up in the page map which produces a 22-bit page frame number. The page frame number is concatenated with the page offset to make a 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, it contains a special bus to "local memory". This local memory also exists in the NuBus address space but the Explorer's own private 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 areas with respect to its use. A portion of memory is set aside for use by the microcode. Data in this space is said to reside in physical memory. These data items do not reside in the virtual memory address space. The rest of the local 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 MEMORY MAP HARDWARE

To avoid the need for a very large mapping memory, or an associative memory, a two-level map is used. The first level map is 4096 words long and is indexed by the high 12 bits of the virtual page number. The first level map produces a 7-bit index into the second level map along with several status bits, including a map-level-one entry valid bit. The second level map consists of 128 blocks of 32 registers each. After using the most significant 12 bits of the virtual page number to index into the level one map, the remaining 5 bits of the virtual page number select a register within one of the second level blocks. If both the level one and level two entries are valid, and if all other status indications are positive, the selected level two map register produces the 22-bit physical page frame address. Note that since the level two map blocks are indexed by the least significant 5 bits of the 17-bit virtual page address, each of the 128 blocks represents 32 contiguous virtual pages.

A page exception can be generated by the memory mapping hardware for a number of different reasons during the level one and level two map lookups. A level one miss can occur if the level one map entry does not have a level two block allocated to it (in fact only up to 128 of the level one entries will be valid at any time). If this occurs a second level block must be allocated and initialized to "map not valid". The referenced page may still be in physical memory, but not in the mapping hardware map. The microcode then consults the Page Hash Table (described below).

In general terms, the level one map can be thought of as a way of partitioning virtual memory into blocks of contiguous virtual

pages to aid memory management and storage allocation functions. The level two map can be thought of as a cache describing the most recently referenced 128 blocks of virtual pages.

Other page exceptions can occur if the page referenced does not have the appropriate read/write access privileges; if the page is actually in the processor's PDL buffer or in A-memory; if the page is marked as "trap on any access" (the "MAR" break feature); and for other reasons associated with garbage collection. 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.

5.6 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 memory resident. The physical page data table (PPD) contains an entry for every physical page of memory. And the reverse first level map contains an entry for every second level map block in the memory map. These tables are described in further detail below.

5.6.1 Page Hash Table.

The page hash table resides in physical memory but does not consume any of the virtual memory address space. It describes every virtual page that is resident in physical memory. When the microcode detects a map miss, the page hash table is consulted to see if the virtual page is still in physical memory. If so, the physical page frame is produced and the memory reference continues normally. If not, 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.

5.6.2 Physical Page Data Table.

The Physical Page Data Table is a physical memory resident table with one word for each page of primary memory. When the system is booted, the microcode determines the size of primary memory and allocates a suitable portion of physical memory for this table.

An entry for a page in Physical Page Data Table is shown in Table 5-1.

All the physical pages that are allocated to hold the microcode management tables are marked -1, 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 5-1 Physical Page Data Area Word Format

Value -----	Meaning -----
-1	Page is not available in virtual memory pool.
PHT Index	Normal page. Value contains the index of the page hash table entry for this page.

5.6.3 Reverse First Level Map Table.

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.

5.7 SECOND LEVEL MEMORY MAP BLOCK 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.

5.8 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 5-1.

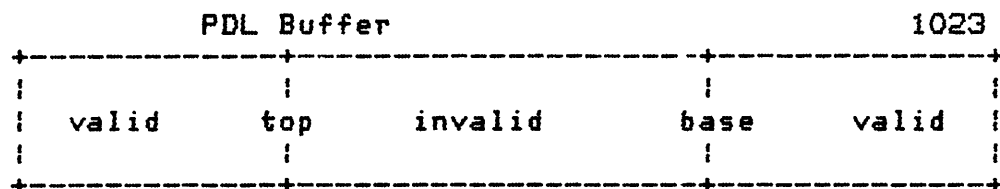


Figure 5-1 PDL Buffer Wrap Around

5.9 VIRTUAL PAGE MANAGEMENT

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

5.9.1 Page Aging Process.

The page aging process, which runs during the idle time while waiting for disk operations to complete, is used to determine which pages should be swapped out, so a needed virtual page can use this physical memory page frame.

The page age process, referred to as the ager, scans the Physical Page Data Table during disk operation idle time. For each physical page that is used by virtual memory management the Page Hash Table entry swap status for the virtual page is updated by the following algorithm:

1. If page status is normal then set status to age trap.
2. If page status is age trap then increment the page age by one.
3. If page age is above a (settable) threshold then mark the page as flushable.

Note that marking the status of a page as flushable is not equivalent to committing it to be flushed. The actual flushing operation does not occur until a physical page frame is required. This means that if a page is marked flushable but then referenced, no disk operations are required. The status is returned to normal to reflect the fact that the page has been recently referenced.

5.9.2 Page Hash Table.

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 will be calculated from a page address mapping scheme, described later.

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 entries are allocated for every physical page in the system. Each entry is two words long. See Figure 5-2 for sizes of the page hash table for different memory sizes. The page hash table requires 1.6% of the physical memory.

The format of an entry in the Page Hash Table is shown in Figure 5-3.

The Virtual Page Number field is the hash key indicating the virtual page that this entry describes.

The V bit field indicates that this entry is valid if it is set. If it is not set this entry is free for use.

Bits 0-2 of word 1 comprise the Swap Status code and indicate the current state of the virtual page. Refer to Table 5-2 for the values and interpretations of the swap status code field.

The AGE field is valid if the status field is Age Trap. Its value is an integer page age value.

The Map Level 2 Control field corresponds to the level 2 memory map data for the control field in the map hardware.

The Map Level 2 Address field indicates the physical page number of this virtual page.

Virtual Memory Size = 32M Words

Physical Memory Size	PHT Size
2MB (512K Words), 2048 pages	8192 words
4MB (1M Words), 4096 pages	16384 words
8MB (2M Words), 8192 pages	32786 words

Figure 5-2 Page Hash Table Sizes

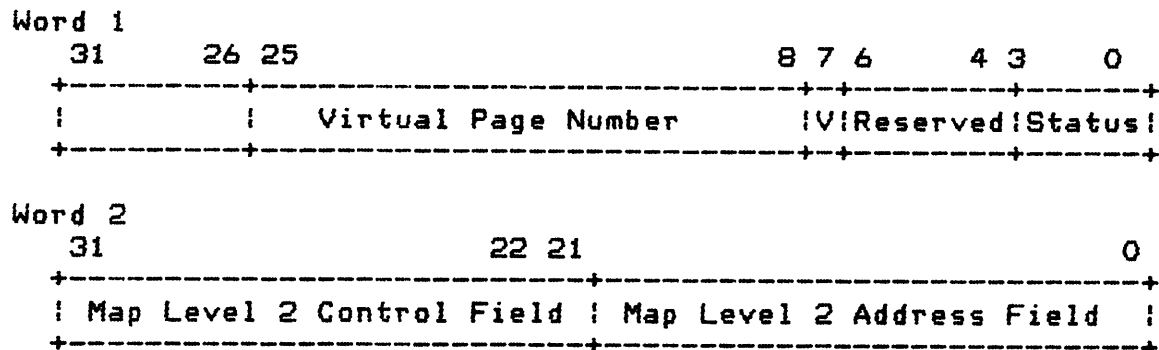


Figure 5-3 Page Hash Table Entry Format

Table 5-2 Page Hash Table: Swap Status Codes

Free:	0 - This virtual page is open for use. It has not yet been used.
Normal:	1 - An ordinary page is swapped in here.
Flushable:	2 - Means that there is a page here, but probably no one is using it, so the memory 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 - Treated the same as flushable, but means that the page came in via a pre-page (fetched with other pages in a block)), and has not yet been touched.
Age trap set:	4 - This page was in normal status, but is now being considered for swap-out. The second-level 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 permanently-wired pages in memory.
Not Used:	6
Not Used:	7

The Page Hash Table is searched using a hash technique. The virtual page number is the hash key. Hash collisions are resolved by a linear search from the hash position with wrap-around if the end of the table area is encountered. If during the hash search an entry is found with the V (valid) bit not set then the entry being searched for is not in the table.

Initially the Page Hash Table contains a dummy entry for every physical page of memory. The swap status code for this entry is set to 0 to indicate that this page is available for use. This works because the page replacement algorithms use the Physical Page Data Table, which points to the entries in the PHT.

5.9.3 Disk Page Mapping Scheme.

This section describes the mechanism by which a virtual page number is used to find the disk address assigned to it.

The scheme uses a Disk Page Map Table (DPMT) that is indexed by virtual page number and gives information about the disk address. Because of the large number of virtual pages it is not practical to have a one-to-one correspondence between virtual pages and DPMT entries. Therefore a cluster of sixteen pages share the mapping information. There is one entry in the DPMT for each group of sixteen contiguous virtual pages. The DPMT is indexed by the most significant 13 bits of the virtual page number. Disk space is allocated in blocks of sixteen pages. Each page is 1024 bytes (256 words).

Each entry of the DPMT specifies two paging bands. A bit map in the entry specifies which of the two bands a particular page in the cluster is mapped into. The corresponding page in the disk block, indexed by the low 4 bits of the virtual page number, is assigned to that virtual page. The disk page 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 disk page. The format of a DPMT entry is shown in Figure 5-4.

Device A and Device B are fields that indicate which "logical paging band" this cluster of virtual pages may be assigned to. A table is kept describing the logical paging devices known to the virtual memory system. This field is conceptually an index into a logical paging device table. 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 read/write accesses are allowed. The values for these fields are described in Figure 5-5. Note that the read-only page band status can be used to describe the Lisp-world load band.

A virtual page operation would proceed as follows:

1. Using the most significant 13 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).

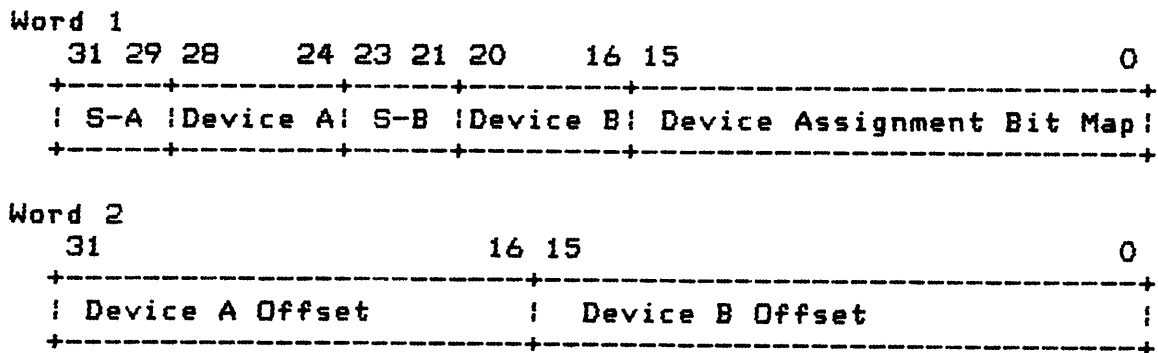


Figure 5-4 Disk Page Mapping Table

- *Status 0: No device assigned
- *Status 1: Read only band
- *Status 2: Read/Write band
- *Status 3: Read/Write band assigned, however, a disk block has not yet been allocated
- *Status 4: - 7: Unused

Figure 5-5 Device Status Codes

When the system is first booted, the DPMT is initialized to indicate that most virtual pages reside in the read-only logical device page band that is the Lisp-world load band. This avoids the necessity of copying the load band to the swap band before starting up Lisp.

5.9.4 Page Swapping.

This section describes the steps taken to resolve a page fault.

1. Determine how many pages should be swapped in on this operation.
2. Find memory page frames for the pages being read by scanning the PPD table. If the page chosen for eviction is dirty then it must be written to a swap partition. In either case the page hash table entry must be updated to indicate that the virtual page is no longer in physical memory (the PHT valid bit must be reset).
3. Issue the read operation.

When a virtual page has never been modified (dirty) there will not be a read/write page band location assigned. When a page becomes dirty a read/write page band location will be assigned to the cluster if it has not already been assigned.

5.9.5 Logical Paging Devices.

A logical paging device defines a set of contiguous pages on a secondary storage device, known as a paging band. Information is maintained to define the characteristics of the paging band associated with each logical paging device.

Referring to Figure 5-6, the information maintained is as follows:

	31	8 7	0
1	Reserved (Unused)		Unit Number
2	Band Starting Disk Block Number		
3	Band Ending Block Number + 1		
4	Next Free Block in Band		

Figure 5-6 Logical Page Device Information Block

- Word 1: Page device status information. The T bit field indicates the type of device. The values are 0 for a read-only band, 1 for a read/write band. The unit number field indicates which physical device this band is associated with. The remainder of this word is reserved for future expansion.
- Word 2: Starting block number of the page bank. Indicates the block number of the first block that may be used in this page band.
- Word 3: End of page band. This word indicates the block number of the first block that is outside of the band.
- Word 4: Current Allocation Pointer. This point indicates the block number of the next free disk block in this band.

A band is a contiguous set of disk blocks on an integral number of tracks. Each disk block is 1024 bytes long. A disk can be partitioned into as many bands as desired, as long as the above restrictions are met.

Disk blocks are allocated by finding a free page cluster in the page device information block bitmap. The bitmap has one bit for each page cluster (a cluster is 16 pages and is the smallest unit allocated in swap space). A 1 indicates the cluster is free and a 0 indicates it is in use. Word 3 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 a flag is set in the flag word and that swap band is no longer checked till a cluster is returned. Garbage collection returns clusters as the virtual memory they represent are no longer used.

SECTION 6

Internal Storage Formats

6.1 INTRODUCTION

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

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

Lisp can adhere to this convention because the basic data items manipulated by Lisp are not objects themselves but rather object references, which can be thought of as pointers to the referenced object. A CONS operation, for example, returns a pointer (object reference) to a two-word block of memory, the first word of which contains the CONSs CAR and the second word of which contains the CONSs CDR. The CAR and CDR are themselves object references; that is, if the CAR is a symbol, the CAR word contains an object reference (pointer) to the symbol, and if the CDR is an array, the CDR word contains an object reference (pointer) to the array.

Some Lisp objects can be represented completely in one storage cell. These are called INUM types (for "immediate number") since the pointer field of such a cell is an actual value rather than a pointer to the actual value. FIXNUMs (small integers) are an example of an INUM type. INUM types are typically implemented for efficiency reasons. The other object reference types are called pointer types.

This pointer based organization yields very flexible data structuring as will become clear as the structuring data types are described.

Rather than fully partitioning memory into different spaces where the different types of lisp objects can be stored, the Explorer system uses the tagged data method of object representation. Thus, a pointer to a CONS cell also includes a tag indicating that the type of the thing referred to is a CONS; likewise with all Lisp object representations supported at the architectural level. (Note that the Explorer system does use reserved spaces for lists for efficiency reasons. See the chapter on storage allocation for further details.)

NOTE

Declarations for Lisp named constants corresponding to the codes, byte descriptors, and offsets described in this chapter are in the file `SYS:COLD-BAND;QCOM.LISP`. Those symbols should be used in any Lisp code that deals with data formats instead of copying numbers from this document.

6.2 Q FORMAT

The 32-bit storage cell in the Lisp Machine is called a Q or quantum. The format of a Q is shown in Figure 6-1. The fields of a Q are the CDR CODE, DATA TYPE and POINTER fields. These are explained below.

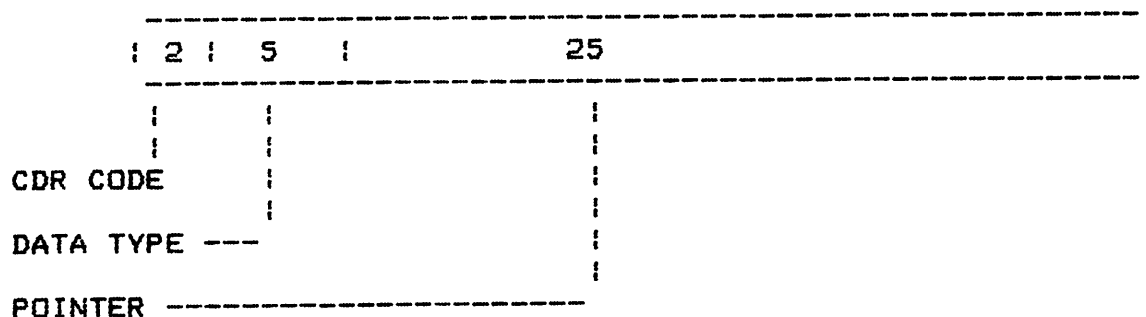


Figure 6-1 Q Format

6.2.1 CDR Code.

The CDR CODE field is a 2-bit field in the Q. If this Q is in a list structure region or in a structure that is treated as a list (a stack list or ART-Q-LIST array), this code indicates how the CDR of this CONS is stored. In some other contexts these 2 bits are used for other specialized purposes. The DATA TYPE and Pointer fields of this Q are for the CAR of the CONS. The encoding of the CDR CODE is shown in Table 6-1.

CDR NORMAL means that the Q following this one contains the CDR. This is the two-pointer form of CONS used in most Lisps. 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. CDR NIL means that the CDR of this node is the symbol NIL; this is the end of an ordinary list.

The CDR NEXT designation is set up to allow a special, high-density storage scheme for regular lists. In such a storage scheme (called cdr coding) a list of N elements can be stored in N consecutive Q's using CDR NEXT and CDR NIL. CDR NEXT indicates that the CDR is a (not physically present) pointer to the Q following. When the CDR of a normal list is requested, a copy of the contents of the CDR cell is returned; when some CDR of a cdr-coded list is requested, a pointer to the location one Q past the CAR is constructed and returned.

The functions APPEND and LIST always form these compact lists while CONS and related operations always create full nodes (CDR NORMAL, CDR ERROR). Note that to RPLACA an element of a CDR NEXT list, you simply clobber the contents of the location, but RPLACDing is more difficult. The LISP machine does this by using the CAR-CDR Invisible pointer which is implemented as a DTP-HEADER-FORWARD (see below).

Table 6-1 CDR Codes

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

6.2.2 Data Type.

The 5-bit DATA TYPE field determines the data type of the Q. The data types are shown below. Note that some of the data types are useful mostly for their meaning in the context of function calling.

6.2.2.1 Data Type 0 - DTP-TRAP.

Any attempt to reference this cell will cause a trap. This is mostly for error checking.

6.2.2.2 Data Type 1 - DTP-NULL.

This datatype is used for various things to mean "nothing." For example, an unbound atom has this type as its value. The pointer field points back at the atom, for ease in debugging.

6.2.2.3 Data Type 2 - DTP-FREE.

This cell is free unallocated storage. The user should not see this too often.

6.2.2.4 Data Type 3 - DTP-SYMBOL.

This is a non-numeric atom. The pointer points to a five Q symbol header.

6.2.2.5 Data Type 4 - DTP-SYMBOL-HEADER.

This is always the first word of a five word block, which is pointed to by a word of DTP-SYMBOL. The header itself acts just like an array pointer (see below).

6.2.2.6 Data Type 5 - DTP-FIX.

A FIXNUM (fixed point number). This is an INUM type. The pointer field is not really a pointer; it is the actual value of the number, so FIX numbers with the same value will always be EQ.

6.2.2.7 Data Type 6 DTP-EXTENDED-NUMBER.

FLONUM, BIGNUM, COMPLEX, RATIONAL; any type of number other than FIXNUMs. It points to a DTP-HEADER word (see below).

6.2.2.8 Data Type 7 DTP-HEADER.

This word is the beginning of a block of storage of some kind. The pointer field does not contain an address: instead it has a HEADER-TYPE field which explains what purpose the header is serving.

6.2.2.9 Data Type 8 DTP-GC-FORWARD.

The forwarding address left behind by the garbage collector. If this is the first word of an object in old space, forwards the entire object to its new location in new space. Any address that got to this object is altered to point to the object's new location (the GC-FORWARD is "snapped out") on the next reference.

6.2.2.10 Data Type 9 DTP-EXTERNAL-VALUE-CELL-POINTER.

This is a kind of invisible pointer. It is used by the closure feature to point to external value cells; it is also used for the exit areas of FEFs to point to the value and function cells of symbols.

6.2.2.11 Data Type 10 DTP-ONE-Q-FORWARD.

This is a simple kind of invisible pointer used to hide a single cell of memory. Forwards only the Q that it is in, not the whole structure. Can be used to alias a symbol's value to that of another symbol.

6.2.2.12 Data Type 11 DTP-HEADER-FORWARD.

This word is the beginning of a block of storage which has been forwarded. The pointer field points to the new location of the header.

6.2.2.13 Data Type 12 DTP-BODY-FORWARD.

This word is a word in a block of storage which has been forwarded. The pointer field points where the header used to be (which should now be a DTP-HEADER-FORWARD). To follow a BODY-FORWARD, follow the header forwarding to reach the new location of the structure and access the word at the same offset into that structure. Needed to forward an array that might have pointers into its body (eg. ART-Q-LIST).

6.2.2.14 Data Type 13 DTP-LOCATIVE.

A locative is a general-purpose pointer to a single cell of memory. It is not an invisible pointer type. It is used for many things; for example, pointers to bound cells on the binding PDL. Both CAR and CDR return the same thing, namely the contents of the cell pointed at.

6.2.2.15 Data Type 14 DTP-LIST.

The pointer field points to a cons cell. The storage format for LISTS is explained above, under CDR CODE.

6.2.2.16 Data Type 15 DTP-U-ENTRY.

This is an INUM type representing a microcoded function. 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-SYMBOL-AREA. The number found there is the control store address of the microcode to run for this function. If the entry in the MICRO-CODE-ENTRY-AREA is not a FIXNUM, then the current definition of this function is not microcoded and this entry is the function to run instead.

6.2.2.17 Data Type 16 DTP-FEF-POINTER.

Points to a macro-compiled function. It points to a word of DTP-FEF-HEADER (DTP-HEADER on a Cadr or Lambda), which is the first of a block of words at least 8 long.

6.2.2.18 Data Type 17 DTP-ARRAY-POINTER.

This is an array object. The pointer points to a word of DTP-ARRAY-HEADER which is followed by the array storage.

6.2.2.19 Data Type 18 DTP-ARRAY-HEADER.

This word is the header word of an array. It may be followed by some extra formatting information, (if it is a long or multidimensional array) and then by the array storage. Preceding it may optionally be an array leader. The pointer field does not actually contain an address, but rather several fields of data describing the array.

6.2.2.20 Data Type 19 DTP-STACK-GROUP.

This is a stack group. This works just like an ARRAY-POINTER; it points to an array header.

6.2.2.21 Data Type 20 DTP-CLOSURE.

This is a closure. It points to a block of storage $2*N+1$ long, where N is the number of cells closed over.

6.2.2.22 Data Type 21 DTP-SMALL-FLONUM.

A small floating point number. The pointer is not really a pointer; instead it is a 24 bit floating point number. There is a 7-bit excess-100 exponent ($10^{**} -19$ TO $10^{**} +19$, approximately) and a 17-bit 2's complement normalized mantissa (5 digits, approximately).

6.2.2.23 Data Type 22 DTP-SELECT-METHOD.

Method table for a flavor.

6.2.2.24 Data Type 23 DTP-INSTANCE.

A flavor instance. Points to a word of DTP-INSTANCE-HEADER.

6.2.2.25 Data Type 24 DTP-INSTANCE-HEADER.

The header word of an INSTANCE. Pointed to by a DTP-INSTANCE.

6.2.2.26 Data Type 25 DTP-ENTITY.

A closure which also binds SELF if it is called (similar to DTP-INSTANCE). This type is to be considered obsolete. This data type should not be used, as it is slated for possible elimination in future releases.

6.2.2.27 Data Type 26 DTP-STACK-CLOSURE.

A funarg. This is a lexical closure which has temporarily been created on the stack. Can only be stored shallower on the stack than where it is. If stored anywhere else, must be copied out of the stack.

6.2.2.28 Data Type 27 DTP-SELF-REF-POINTER.

This is a special data type used for referencing instance variables, accessing self-mapping tables, and monitoring variables. For a further description, see paragraph on Self-Reference-Pointer Format below.

6.2.2.29 Data Type 28 DTP-CHARACTER.

A character. Primarily for Common Lisp compatability. Can be used in arithmetic like a FIXNUM.

6.2.2.30 Data Type 29 - DTP-FEF-HEADER.

This is the header for a FEF on the Explorer. Cadr and Lambda use DTP-HEADER.

6.2.2.31 Data Type 30 THROUGH 31.

Data type 30 through 31 are unused. They are treated the same as DTP-TRAP.

6.2.3 Pointer.

POINTER (25 bits) - The use of the pointer field is determined by the data type of the Q. Usually it points to some other object in memory. Sometimes it just contains miscellaneous data.

6.3 STRUCTURE HEADERS

A word of DTP-HEADER is the first word of a number of structure types. The type of the structure is indicated by the HEADER TYPE field. Header types are shown in Table 6-2. Each type of structure is discussed in detail in the following paragraphs.

Table 6-2 Header Types

Call	Symbol	Meaning
----	-----	-----
0	Q-HEADER-TYPE-ERROR	Bad header, not used
1	Q-HEADER-TYPE-FEF	Macro-compiled function (unused on Explorer)
2	Q-HEADER-TYPE-ARRAY-LEADER	The word before the leader of an array (which is before the DTP-ARRAY-HEADER word). Present to help the garbage collector find all the storage used by an array.
3	Q-HEADER-TYPE-UNUSED	
4	Q-HEADER-TYPE-FLONUM	Header word of floating point extended number.
5	Q-HEADER-TYPE-COMPLEX	Header word of complex extended number.
6	Q-HEADER-TYPE-BIGNUM	Header word of infinite precision integer.
7	Q-HEADER-TYPE-RATIONAL-BIGNUM	Header word of ratio of two BIGNUMs

6.4 INVISIBLE FORWARDING POINTERS

Invisible forwarding pointer types provide data indirection. That is, whenever an invisible pointer is read, the read is indirected along the invisible pointer. This 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. Thus if you take the CAR of a Q which contains an invisible pointer as its CAR Q, you will really be given the CAR of what the invisible pointer points to.

DTP-EVCP-FORWARD and DTP-HEADER-FORWARD are invisible pointer data types. Other forwarding types are similar; they are DTP-GC-FORWARD and DTP-BODY-FORWARD. These perform additional work beyond following the invisible pointer. Even DTP-SELF-REFERENCE-POINTER (called an SRP) exhibits behavior similar to an invisible pointer. Even though an SRP is an INUM type, when one is used for a mapped or unmapped instance variable reference, it indirections the reference to an instance variable, much as an invisible pointer.

6.5 IN THE MACHINE

In order to facilitate garbage collection and to make the microcode simpler, there is a group of storage conventions that define which data types can be "in the machine". The "machine" refers to a set of internal processor registers used by the microcode, and the PDL buffer. It is not necessary to understand these storage conventions fully unless the microcode is being modified. The normal Lisp user almost never needs to be aware of these conventions either. The exception is if Lisp code is being written that uses miscellaneous operations (generally, the %-functions, described in the chapter on subprimitives). These microcoded functions do not do type checking on their arguments hence their use can violate the machine's storage conventions. For more information about which data types are allowed "in the machine", see the chapter on garbage collection.

6.6 SYMBOLS

A symbol is stored as a Q of datatype DTP-SYMBOL whose pointer points to a five Q symbol block. The five words are listed in Table 6-3.

The PRINT-NAME-CELL holds a word of DTP-SYMBOL-HEADER pointing to a STRING array which is the PNAME for the symbol. (See ARRAY formats).

The VALUE-CELL holds the value of the symbol, and so can be of almost any data type. Instead of containing a value, a symbol's VALUE-CELL may be empty or unbound. If the symbol is unbound, this cell contains DTP-NULL. Symbols may be used as dynamic variables; this use is described in the paragraph Binding PDL below.

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.

Instead of containing a value, a symbol's FUNCTION-CELL may be empty, in which case it contains DTP-NULL.

Table 6-3 Qs of Symbol

Offset	Cell Name
-----	-----
0	PRINT-NAME-CELL
1	VALUE-CELL
2	FUNCTION-CELL
3	PROPERTY-CELL
4	PACKAGE-CELL

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 functions PRINT-NAME-CELL-LOCATION, VALUE-CELL-LOCATION, can be used to obtain DTP-LOCATIVE pointers to these locations and the contents can, of course, be gotten by taking the CAR of the pointers thus obtained.

6.7 ARRAYS

An array consists of a group of cells, each of which may contain an object. The individual cells are selected by numerical subscripts. The rank of an array is the number of subscripts used to refer to one of 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, 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 zero, one, two, three, and four.

There are many types of arrays. Some types of arrays can hold Lisp objects of any type; the other types of arrays can only hold fixnums or flonums. The array types are known by a set of symbols whose names begin with "ART-" (for ARray Type).

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

An array object is represented as a DTP-ARRAY-POINTER. The pointer field must point to an array header word. Every array has an array header word. An array header word is of DTP-ARRAY-HEADER. Its pointer field has the format shown in Figure 6-2.

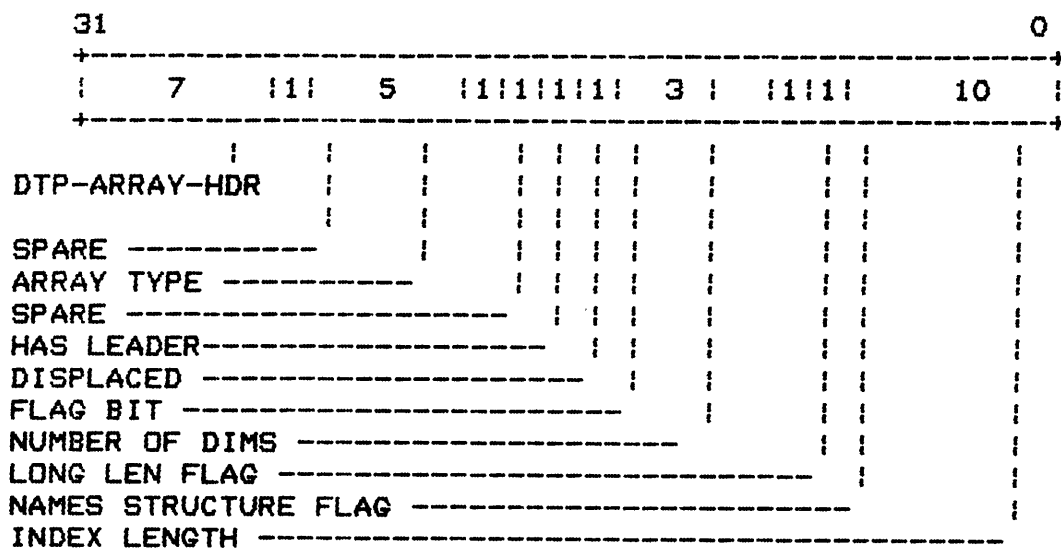


Figure 6-2 Array Header Word

6.7.1 Has Leader.

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 Q immediately before the header word is a FIXNUM Q holding the number of array leader words. Then before that are the array leader words, which may have any data type (since any object can be stored there), and before that is a word of data type DTP-HEADER and header type Q-HEADER-TYPE-ARRAY-LEADER. The presence of this header is necessary for such routines as the garbage collector which scans through memory in the usual direction (that is, from low to high addresses). The storage layout of an array with leader is shown in Figure 6-3.

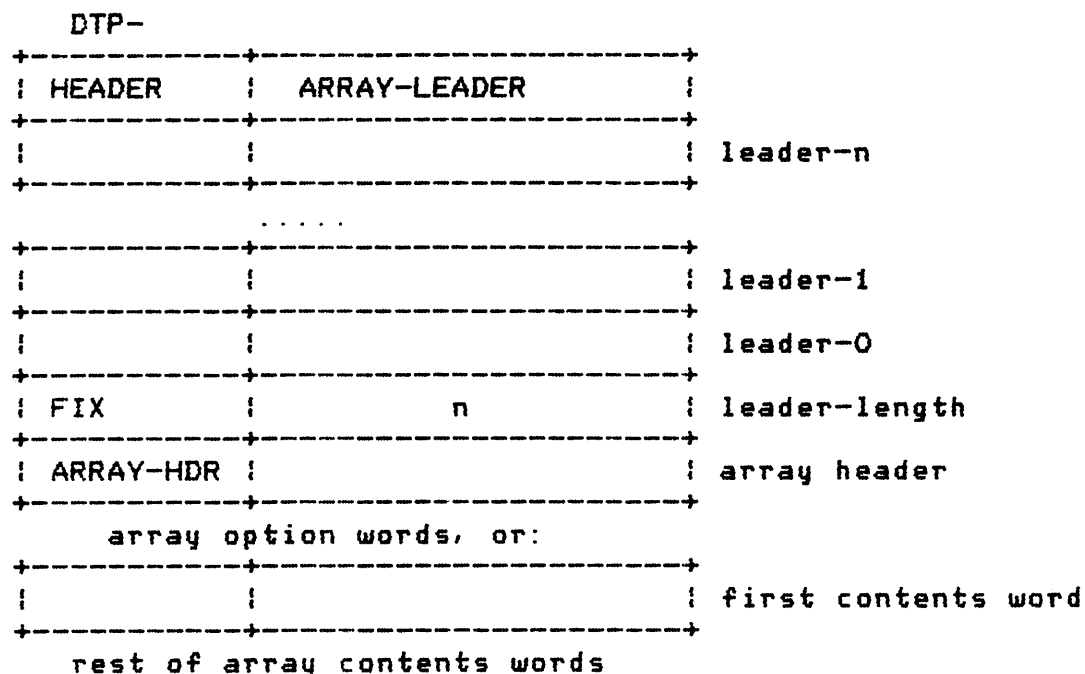


Figure 6-3 Array with Leader

6.7.2 Displaced.

An array may optionally be displaced, according to the **DISPLACED** bit in the header. If the array is not displaced, then the data words follow thereafter (in a 1-dimensional non-displaced array, the data follows immediately after the header). However, if the array is displaced, then the word which would be the first data word is actually a pointer to the data cells.

Thus, a displaced array can be used to point at the beginning of an area (this is done often, in fact). Following the displacement word, in what would have been the **SECOND** data cell, is the length of the data in Q's for the array. This is used instead of the normal index length, since that will be 2 to indicate the length of the displaced array.

If the array is displaced and the word which would be the pointer has data type DTP-ARRAY-POINTER, then it points to another array header. This is called an indirect array. Call the array pointed to the indirected array, and the displaced array the indirect array. Then the index length of the indirect array appears to be $\min(x, y)$ where x is the index length of the indirected array and y is the second data word of the indirect array. Computing \min prevents referencing beyond the actual end of the indirected array.

If the indirect array has a third element, then this array has an index-offset from the indirected array. This means that whenever the indirected array is referenced, it is as if that array were referenced, but with an index n higher. The offset, n , is stored as a FIXNUM in what would be the third data cell if the indirect array were not displaced. The offset is expressed in elements (not Q 's), and is always 1 dimensional (it is added after all the dimensions have been multiplied out). The resulting index is checked against the index length computed as above.

6.7.3 Number of Dims.

If the array has more than one dimension, then there is a block of $\langle \text{number of dims} \rangle - 1$ Q 's immediately after the array header holding the size of each dimension. Note that only $\langle \text{number of dims} \rangle - 1$ are needed because one can compute the total index length from the array header itself.

6.7.4 Long Length Flag.

If the index length of the array (number of data elements) is too big to fit in the field allocated for it in the array header Q , an extra Q is inserted between the header and the dimensions, which has data type FIXNUM and contains the index length. The LONG LENGTH FLAG bit in the header Q is on to indicate the presence of this extra Q . The long length Q is the word immediately after the header word if it is present.

6.7.5 Named 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, etc). The structure name is found in array leader element 1 if ARRAY LEADER is set, otherwise 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.

6.7.6 Index Length.

The INDEX LENGTH of a one dimensional array is the maximum value for the index. In a multidimensional array, it is the product of the sizes of each of the dimensions. Note: If the INDEX LENGTH of an array is larger than will fit in this field, it is stored in the next Q and the LONG LENGTH FLAG is set.

6.7.7 Array Type.

The array type indicates the type of the array. The array type indicates how the data would be accessed and the type of data that may be stored in the array. The array types are summarized in Table 6-4 and explained in the Explorer Lisp Reference Manual. Note: the elements of arrays (those which are smaller than 32 bits) are stored right-to-left within each word (i.e., the first element of a 4 BIT ARRAY would be stored right-justified, including the least significant bit).

Table 6-4 Array Types

Code ----	Type -----	Code ----	Type -----
0	ART-ERROR	10	ART-STACK-GROUP-HEAD
1	ART-1B	11	ART-SPECIAL-PDL
2	ART-2B	12	ART-HALF-FIX
3	ART-4B	13	ART-REG-PDL
4	ART-8B	14	ART-FLOAT
5	ART-16B	15	ART-FPS-FLOAT
6	ART-32B	16	ART-FAT-STRING
7	ART-Q	17	ART-COMPLEX-FLOAT
8	ART-Q-LIST	18	ART-COMPLEX
9	ART-STRING	19	ART-COMPLEX-FPS-FLOAT
		20	ART-FIX (Explorer only)

6.8 DTP-SELF-REF-POINTER FORMAT

The format of a DTP-SELF-REF-POINTER is shown in Table 6-5. A SELF-REF-POINTER is used for three different purposes depending on whether MAP-LEADER-FLAG or MONITOR-FLAG is set. If neither is set, an SRP is an invisible pointer to an instance variable in SELF. If MAP-LEADER-FLAG is set, the SRP is an invisible pointer to a slot of SELF-MAPPING-TABLE. If MONITOR-FLAG is set, the SRP

is an invisible pointer to the next location on read and causes a trap on write. This is used to monitor variables.

Table 6-5 SELF-REF-POINTER Format

Bit 19:	SELF-REF-RELOCATE-FLAG	(#o2301)
Bit 18:	SELF-REF-MAP-LEADER FLAG	(#o2201)
Bit 17:	SELF-REF-MONITOR-FLAG	(#o2101)
Bits 12-0:	SELF-REF-INDEX	(#o0014)
Bits 12-1:	SELF-REF-WORD-INDEX	(#o0113)

The RELOCATE-FLAG, when set, says to use SELF-MAPPING-TABLE. This is the standard case (about 75% of SRPs). The INDEX is the index into the mapping table, the contents of which are an offset into the instance. The WORD-INDEX is the index in words; mapping tables are ART-16B arrays. A SELF-REF-POINTER with this flag set is used to access most instance variables. Unmapped instance variables are created by the :ORDERED-INSTANCE-VARIABLES option to DEFFLAVOR, and the index in this case is the offset directly into the instance.

The MAP-LEADER-FLAG, when set, says 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 25% of SRPs). The INDEX is the index into the array leader.

The MONITOR-FLAG, when set, means that this SELF-REF-POINTER is in fact a monitor pointer. No monitor pointers seem to appear within methods. A monitor pointer indirects to the next location (SRP-location+1) on read and traps on write.

SELF-REF-POINTERS are created and manipulated within the code for flavors, mostly in the mapping-table sections.

6.9 PDL FORMAT

The stack in the LISP Machine is stored in main memory, with the top kept in the PDL BUFFER of the processor. The PDL Buffer acts as a 1K cache which greatly speeds up almost all references to the stack. The cache is maintained by microcode invisibly to the macro-code and all higher levels.

The PDL buffer is "inside the machine" and therefore is not allowed to contain illegal or forwarding data types. In addition, it always contains boxed (typed) data.

6.10 FEF FORMATS

When a function is macro-compiled, the macrocompiler produces a Function Entry Frame (FEF). The FEF contains various things including random information about the function, symbols and constants used in the function, and the macrocode itself. See the section on Function Calling for details.

6.11 FLOATING POINT FORMATS

There are several floating point formats supported in the Explorer. Small 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 its value. Normal floating point numbers are represented as a pointer type of DTP-EXTENDED-NUMBER pointing to a DTP-HEADER with header type HDR-TYPE-FLONUM. The exponent is stored in the header word and the mantissa is stored in the header word and the next word. A flonum has a storage length of 2.

The format of a small flonum is shown in Figure 6-4. This format has an 8-bit exponent and a 17-bit mantissa.

A normal floating point number is represented as an object of DTP-EXTENDED-NUMBER pointing to a two-word structure as shown in Figure 6-5. The Flonum Header has the format shown in Figure 6-6. The exponent is stored in excess-4000 (octal) form. The high order mantissa has the most significant eight bits of a two's complement mantissa (sign bit is the MSB). This is concatenated with the low order 24 bits of mantissa from the Flonum mantissa.

The Flonum Mantissa has the format shown in Figure 6-7.

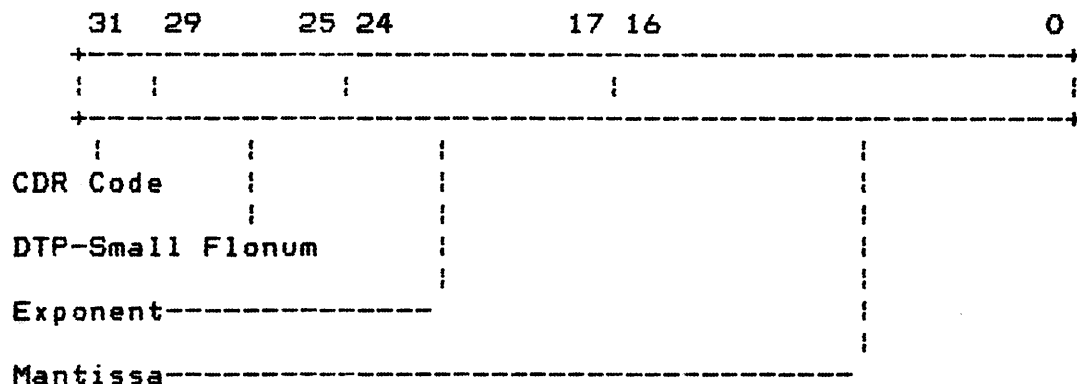


Figure 6-4 Small Flonum Format

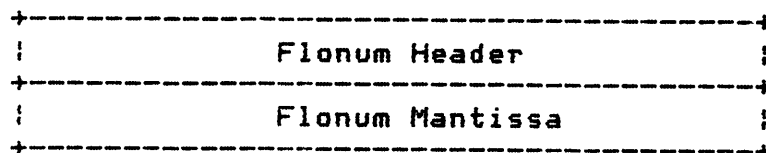


Figure 6-5 Flonum Structure Format

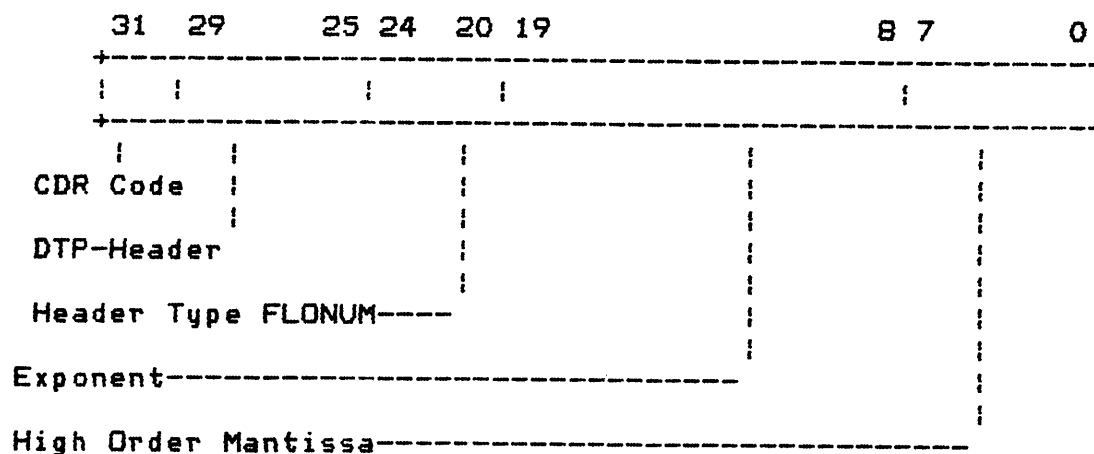


Figure 6-6 Flonum Header Format

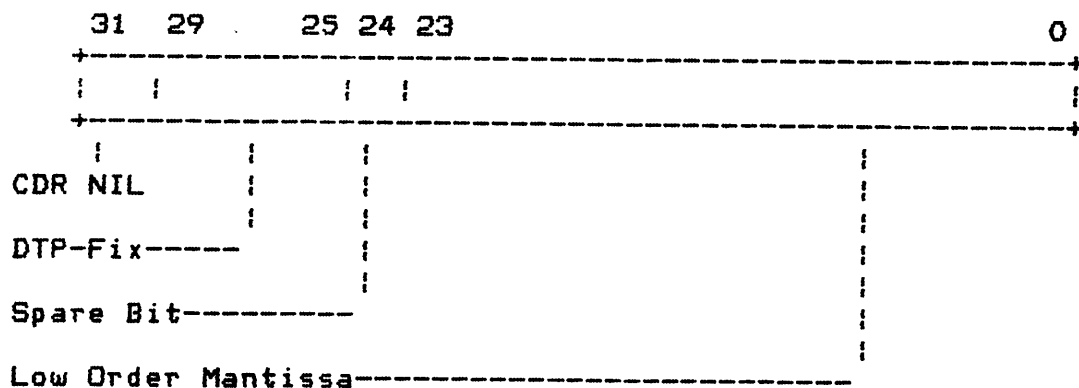


Figure 6-7 Flonum Mantissa Format

6.12 BIGNUM FORMAT

A bignum is an extended precision integer. The storage length of a bignum is determined by the size of the integer it represents. A bignum is represented as an object of DTP-EXTENDED-NUMBER pointing to a structure. The format of the bignum structure is shown in Figure 6-8. 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 6-9. The LENGTH field gives the length of the bignum in words; this is the length of the bignum structure minus one.

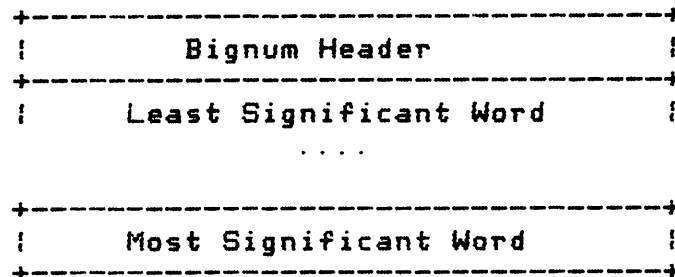


Figure 6-8 Bignum Structure

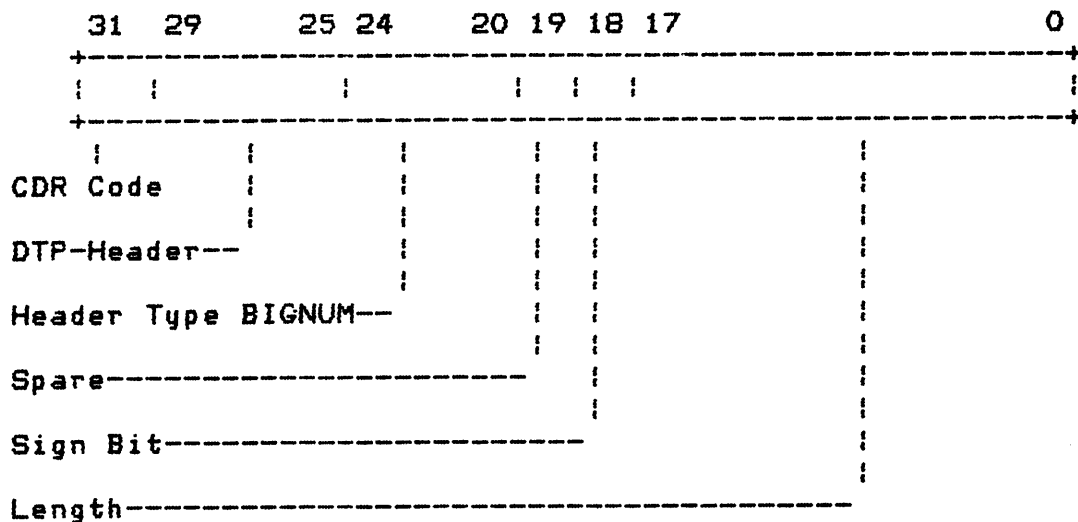


Figure 6-9 Bignum Header Format

Each word of the bignum has the format shown in Figure 6-10. The high order bit is always 0. The remaining bits are a section of the bits of the positive integer that is represented.

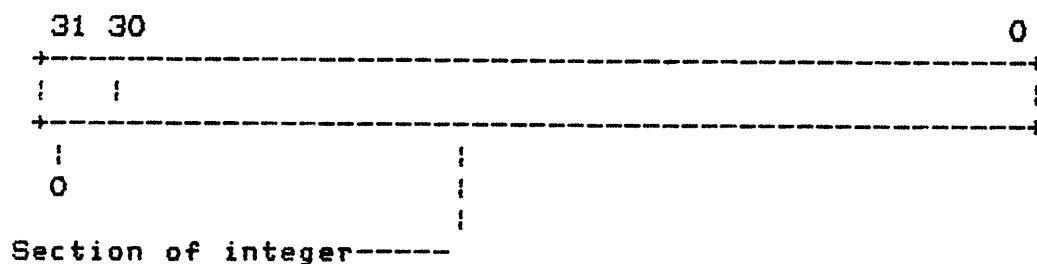


Figure 6-10 Bignum Data Format

6.13 CHARACTER FORMAT

A character object has the format shown in Figure 6-11.

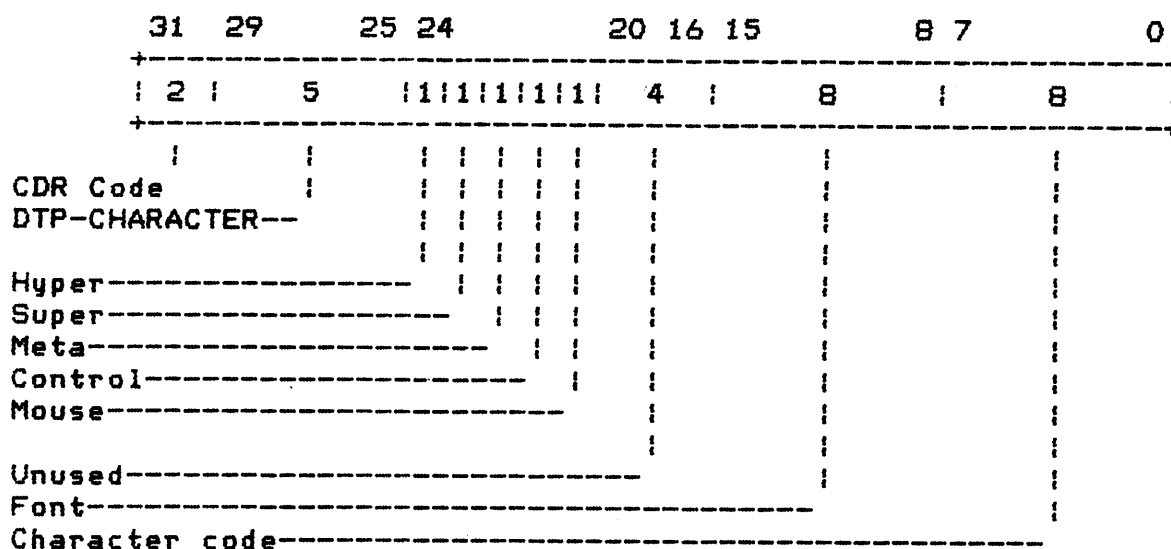


Figure 6-11 Character Format

6.14 CLOSURE FORMATS

Conceptually, a 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 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). 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 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 symbol's 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 SPEC PDL) 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.

Closure binding is a special kind of binding which may be shared; that is, other functions and closures in the same binding environment as the CLOSURE function can access and change the 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, 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 binding 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 symbol's EVCP. Finally, CLOSURE returns an object of datatype DTP-CLOSURE which points at the block. This is the closure itself.

A symbol's internal value cell contains an EVCP only when its binding is being remembered by some 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 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 closure are placed in the symbols' internal value cells, restoring the saved environment. Finally, the function is invoked with the arguments passed to the closure.

SECTION 7

Storage Management

7.1 INTRODUCTION

This chapter explains Explorer storage management. This includes areas and regions. Spaces and garbage collection will be covered in the next section.

Storage allocation for Lisp objects and structures 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 the allocation of address space to Lisp objects.

The storage allocation system manages the address space by breaking it down by two levels into smaller pieces. The first level breaks the address space into areas. An area is a collection of regions, which are the second level of storage management.

Areas are created by explicit commands. 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). Virtual address space is then assigned to a region when a region is created in an area. Actual allocation of virtual address space to objects takes place in regions as storage requests are made. We use the general term consing to refer to any allocation of storage, whether to actual cons cells or to other objects such as arrays or locatives.

7.2 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 of the 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 area descriptor table.

Each area is described by 5 words of data in the area descriptor table.

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 - The value for the Region Bits word for a region allocated in this area.

Area Region Size - The size of a region when a region is allocated in this area.

Area Maximum Size - The maximum size this area is allowed to occupy.

The set of attributes for an area is defined by the Area Region Bits word. When a new region is created it inherits the attributes of the area to which it belongs. These attributes will be described in detail in the section on regions.

Logically, the area descriptor table is a table of five-word entries as shown above. In reality, the table exists as five separate tables indexed 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. 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. In addition, paging can be controlled on an area basis.

7.3 REGIONS

A region is a block of contiguous virtual address space. Each region has a set of properties which hold for all objects in that region. The region's size is one of those properties. Normally, the amount of virtual address space assigned to a region must be a multiple of the region quantum size, which is currently 64 pages. When all the address space assigned to a region has been allocated, new regions may be created in the same area by the

system until the area's maximum size has been reached. There is a default region size associated with each area.

Each region is identified by a number from 0 to 2047. This number is used as an index into the region descriptor table. The Region Descriptor Table contains information about the properties of all regions. Each region has a six-word entry in the table. Like the Area Descriptor Table, the Region Descriptor Table is actually implemented as separate tables indexed by the region number; each table corresponds to one of the words of an entry. The words of an entry are:

Region Origin - Starting virtual address of the region.

Region Length - The total amount of virtual address space assigned to this region.

Region Bits - Specific attributes of this region. See Figure 7-1.

Region Free Pointer - Offset into this region of the next free word that can be allocated. The virtual address of the next free word in this region can thus be calculated by adding the region free pointer to the region origin address.

Region GC Pointer - Offset into this region of the next object which needs to be scavanged. See chapter on garbage collection.

Region List Thread - Region number of the next region in the list.

The regions in an area are linked in a list. 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.

Regions can store two types of data: list data and structure data (structure data is any object other than a list). A particular region can store only one type of data; thus each region has a representation type.

The space type attribute defines the storage allocation scheme that is used. The encoding of this field is shown in Table 7-1.

Table 7-1 Space Type Codes

Code	Region Type
0	Free
1	Oldspace region of dynamic area
2	Permanent newspace region of dynamic area
3	Temporary space level 1
4-8	Temporary space level n - 2
9	Static area
10	Fixed, static, not growable, no consing
11	Extra PDL for a stack group
12	
13	
14	Copy space

The Region Bits word defines several attributes of the region. The fields within the region bits word are shown in Figure 7-1.

31	26	25	20	19	18	17	16	14	13	12	9	8	7	5	4	0
+-----+-----+-----+-----+-----+-----+-----+-----+																
Res		Status			Rep		O		GCV		R		Type		S Res Swap Size	
+-----+-----+-----+-----+-----+-----+-----+-----+																

Res: Reserved, unused.

Status: Access and status bits to be used in the level 2 memory map.

Rep: Representation type. 0 = list, 1 = structure, 2 and 3 are unused.

O: Oldspace meta bit. 0 = old space or free, 1 = new space, static space or fixed.

GCV: GC Volatility for this region

R: Reserved, unused.

Type: Space type.

S: Scavenger Enable. Value: 1 = Scavenger can touch this area.

Res: Reserved, unused.

Swap Size: Number of pages the Virtual Memory System should try to swap at a time.

Figure 7-1 Region Bits Area Entry Description

7.4 STANDARD AREAS

When a machine comes up after a cold boot there are about 30 areas allocated for used by the system itself. The first dozen or so areas are 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 (GCOM) 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 GCOM 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 64 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 Physical Page Data table and the Page Hash Table are allocated enough virtual space to handle up to a 10 megaword physical memory. However, only enough physical memory is used as is needed for management of the actual physical memory size. The physical memory originally allocated to these tables is returned to the virtual memory page pool and the memory maps are rolled back to only refer to the actual memory wired down.

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 utilites. See discussion of SCA below.
- * Scratch Pad Init Area (wired, read only).
- * Micro Code Symbol 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.
- * Page Table Area (wired) - Contains the Page Hash Table for the Virtual Memory System.
- * Disk Page Map Area (wired) - Contains the Disk Page Map Table for the Virtual Memory System.
- * Physical Page Data Area (wired) - Contains the Physical Page Data 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.

- * Constants Area (fixed, read only) - Contains some constants, references to which are generated by the compiler.
- * 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 flonums 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 Symbol Area for each microcoded function (see discussion of DTP-U-ENTRY in chapter on data types).
- * Microcode Entry Name Area (fixed) - Contains names of all microcoded functions.
- * Microcode Entry Args Info Area (fixed) - Contains information about arguments to microcoded functions.
- * Microcode Entry Max PDL Usage (fixed)
- * Microcode Entry Arglist Area (fixed)
- * Microcode Symbol Name Area (fixed, read only)

- * **Linear PDL Area (fixed)** - The Linear PDL Area contains the Linear PDL (Push Down List, or stack) for each 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 processors 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)**
- * **Init List Area (fixed, read only)** - Contains list structures for the system's initialization lists (warm, cold, system, etc.).
- * **Working Storage Area** - The default cons area; most objects created by users are created in this area.
- * **Permanent Storage Area**
- * **Property List Area**
- * **Print Name String Area**
- * **Control Tables Area**
- * **OBT Tails Area**
- * **Non-Resident Symbol Area**
- * **Macro Compiled Program Area** - The Macro Compiled Program Area is where all compiled functions are loaded. (This includes methods which are a special kind of function.)

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.

- * **Special PDL Area** - The Special PDL Area contains the Special or Binding PDL for each process. The Binding PDL contains the variable binding information for a process.
- * **FASL Table Area**
- * **FASL Temp Area**

7.5 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:COLD-BAND;GCOM.LISP` contains the definitions of items in this area. The Systems Communication Area is wired and at the fixed address of 400 (octal).

A map of systems communications areas is shown in Figure 7-2.

(octal)	
Addresses 400-437:	Miscellaneous words
Addresses 440-477:	Not currently assigned in Explorer
Addresses 500-511:	Keyboard buffer header
Addresses 600-637:	Disk Error Log
Addresses 700-777:	Not currently assigned in Explorer

Figure 7-2 Map of Systems Communications Area

The miscellaneous words (400 - 437) 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
3. Page Table Pointer - virtual address of the Page Hash Table
4. Page Table Size
5. Object Array Pointer
6. Ether Free List
7. Ether Transmit List
8. Ether Receive List
9. Band Format
10. GC Generation Number

11. Unibus Interrupt List - list of interrupt descriptors for simple buffered devices. This list is searched when an interrupt occurs and there is no special handler register to handle it.
12. Temporary
13. Free Area Number List
14. Free Region Number List
15. Memory Size
16. Wired Size
17. Chaos Free List
18. Chaos Transmit List
19. Chaos Receive List
20. Debugger Requests
21. Debugger Keep Alive
22. Debugger Data 1
23. Debugger Data 2
24. Major Version
25. Desired Microcode Version
26. Highest Virtual Address

SECTION 8

Garbage Collection

8.1 INTRODUCTION

This chapter explains how the Lisp system recovers storage that is no longer in use. The collection (GC) algorithm used is based on the Baker algorithm. (See H.G. Baker, "List Processing in Real Time on a Serial Computer," Communications of the ACM 21,4 (April 1978) p. 280-294.)

8.2 IN THE MACHINE

An important concept that is needed to explain how the garbage collector works is the concept of in the machine. The universe of places to store Lisp objects includes virtual memory, the processors PDL Buffer, and processor registers. It is very helpful for efficiency reasons to divide these places into those that are "inside the machine" and those that are not. While the storage conventions regarding the locations inside the machine do not solely exist for the benefit of the garbage collector, garbage collection does make heavy use of the convention enforcing facilities (such as the transporter) in order to enforce its own rules about old space and new space, and to ensure proper treatment of its structure forwarding mechanism (the DPT-GC-FORWARD pointer).

The locations declared to be inside the machine are:

- * The lettered processor registers M-ZR, M-A, M-B, M-C, M-D, M-E, M-I, M-J, M-K, M-Q, M-R, M-S, M-T. These are the working "state" registers that are saved in stack groups.
- * The "Q storage" section of A-memory, which include the locations A-Version through A-End-Q-Pointers. Most of these locations contain values of variables accessible from Lisp. The global variable sys:a-memory-location-names contains a list of these variables.
- * Other A- and M-memory locations that are stored in stack groups: M-Flags, M-AP, A-QLBNBP, A-IPMARK, A-GDPDLO, A-Trap-AP-Level.

- * The processor's PDL Buffer, which is a cache of the top 1K words of the current process's runtime stack.

Many of the data types described in chapter 5 are not allowed inside the machine. In addition, pointer type objects which point to old space (space where garbage collection is occurring) are also not allowed. Together these form the set of active values that are not legal in the machine. The disallowed data types are:

- * The illegal data types: DTP-TRAP, DTP-NUL, and the unused data types (numbers 30 and 31).
- * The header types: DTP-HEADER, DTP-SYMBOL-HEADER, DTP-ARRAY-HEADER, DTP-INSTANCE-HEADER and DTP-FEF-HEADER.
- * The forwarding types: DTP-ONE-Q-FORWARD, DTP-QC-FORWARD, DTP-HEADER-FORWARD, DTP-BODY-FORWARD, DTP-EXTERNAL-VALUE-CELL-POINTER.
- * The special type DTP-SELF-REFERENCE-POINTER.

The set of data types can be divided into two groups: those that represent actual Lisp objects (such as DTP-LIST, DTP-SYMBOL, DTP-INSTANCE, and so forth), and those that are "internal" data types used for special purposes (such as structure definition and forwarding) by the microcode. Note that the disallowed data types are all the data types in the latter category. The processor locations inside the machine can be thought of as a kernel data area for the parts of the microcode that support Lisp directly, such as function calling and macrocode support. Since these routines logically only deal with actual Lisp objects, it makes the microcode much simpler to disallow other data types in their working area. Since no active values are allowed inside the machine, tests for active values are not then constantly needed when accessing data inside the machine.

8.3 THE READ BARRIER

In order to assure that active values are not stored inside the machine, two things are required. First, there must be a rule that no active value is ever generated and stored inside the machine. This rule is followed both by the microcode that manipulates locations inside the machine and by most Lisp operations that can directly manipulate machine locations. Note, however, that many of the miscellaneous operations implemented as %-functions do no type checking on their arguments, which are placed on the PDL. These functions must be used carefully since they can potentially allow illegal values inside the machine. If they do, the machine may later crash when some microcoded function finds a bad data type, or when the garbage collector

encounters the bad type in a place it did not expect.

The second requirement is that there must be a "barrier" to protect against reading an active value into the machine from memory. This read barrier is implemented as the transporter.

The transporter consists of a test for active values and routines to take the appropriate action when an active value is encountered. The decision on whether the value is an active value is based on the data type of the object read from memory and the OLDSPACE property of the region that the object points to (if it is a pointer type object).

Attempts to read active values into the machine cause some action to take place. The specific action depends on the type of the object. Possible actions include calling the trap routine, calling illop (to crash), following a forwarding pointer in order to "snap it out", or copying an old space object to new space. These last two are the transporter features relied on by the garbage collector to ensure its proper operation. In any case, the prohibited object is not allowed to pass into the machine.

When the transporter traps due to reading a reference to old space, the object is immediately copied to copy space (unless it has already been copied). Hence objects that are inside the machine cannot point to old space. It is not possible therefore to store a pointer to old space back into memory since it must first be read into the machine and transported.

8.4 INCREMENTAL GC

Incremental garbage collection allows the collection process to occur at the same time that new storage is being allocated by running programs. In addition, incremental GC performs the collection gradually, without long delays between operations noticed by running programs. It does this by doing a relatively constant portion of the garbage collection work every time new storage is consed. Thus the rate of garbage collection is related to the rate of consing in the system.

When the system is booted, all space is initially designated as new space. Storage is gradually allocated in these new space regions, and garbage accumulates in them when programs relinquish the pointers to storage they previously requested. At some point, when garbage collection begins, the dynamic new space regions are re-classified as old space. This process is known as flipping. Fresh new space regions are allocated during the flip since new consing after the flip must only take place in new space. Old space is then the domain the garbage collector works upon to purge of garbage. The goal of the garbage collection cycle is to copy everything useful out of old space (into copy

space regions) and then reclaim the storage used by old space. The net savings gained by garbage collection is then the difference between the sizes of the original old space regions and the copy space regions where only the objects in use have been copied.

While useful objects are being copied from old space to new space (through the process called scavenging, described below), any requests for new storage are allocated in new space regions. Furthermore, so that new space does not itself need to be scavenged at a later time, nothing in new space is allowed to refer to old space. The transporter's read barrier is used to guarantee that any pointers to old space will be correctly altered to refer to the object's new representation in copy space before being written to new space. If the object has not yet been copied to copy space, the appropriate copying routines must be called.

The remaining type of space is static space. Static space contains objects that are intended to remain forever. Garbage collections does not reclaim space in static space, although static spaces must be scavenged. Any references to objects in old space that occur in static space must be replaced by references to the object's copy in copy space.

8.4.1 Scavenging.

Scavenging is the operation of cleaning up the references to old space that occur in copy space and static space (recall that since new space contains no pointers to old space, there is no need to scavenge it). Typed words are examined in the spaces being scavenged, and if a pointer to old space is found, the object is copied out of old space into copy space. When none of the spaces contain pointers into old space, there are no more useful objects remaining in old space and it may be reclaimed.

It is important to guarantee that there is no useful object remaining in old space when it is reclaimed. The scavenging algorithm ensures this. 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 copy space. As it progresses, it updates a scavenge pointer which delimits the portion of the space that has been scavenged. The storage before the scavenge pointer cannot refer to old space since it has already been scavenged, and no pointer to old space is allowed to be stored to "dirty" any already scavenged structure since the transporter does not allow this. When the scavenge pointer reaches the end of copy space, copy space does not contain any pointers to old space.

Static space is likewise scavenged, with the copied objects moved to copy space. When all of both static and copy spaces have been

completely scavenged, no pointers to old space exist and it may be reclaimed.

8.4.2 Shared Objects.

There is a problem with this scheme when an object is referred to by several pointers. After garbage collection, the copied object must still be shared in the same way the original in old space was. The scheme outlined above would make several copies of the object, defeating the sharing.

In order to preserve sharing, when an object is copied out of old space, it is replaced with DTP-GC-FORWARD which refers to its new location in copy space. Before copying an object, a check is first made for a GC forwarding pointer. If the object is forwarded, it is not recopied; instead a pointer to its new location is returned.

DTP-GC-FORWARD is not valid except in old space. In other spaces it is an ILLOP to read a DTP-GC-FORWARD.

8.4.3 Areas.

The description above is incomplete insofar as it does not take into account division of the address space into areas. Every object is in some region that is part of an area. Each region has either the old, new, copy or static space property. Every area that has one or more old space regions has one or more copy space regions once garbage collection has begun.

When an object is copied from old space to copy space, the copy occurs between two regions in the same area. Garbage collection does not change the area in which an object is stored.

8.4.4 Flipping.

The remaining phase of incremental GC that has not been fully discussed is flipping. At the latest moment when GC can begin, a flip occurs and all new space and copy space regions are redesignated old space, and the copy process is started again. Just before the flip occurs all pointers in the machine are written to memory. After the flip occurs the machine is reloaded from memory. Of course, all pointers into old space are transported, so that the machine never contains pointers to old space.

8.5 THE WRITE BARRIER

The write barrier is implemented by GC Write Test. It is used to detect the writing of pointers to the Extra PDL Area. Every Lisp object written is tested with the GC write test.

The Extra PDL Area, or number consing area, is used to reduce the garbage generated when evaluating arithmetic expressions. All bignums and flonums are first consed in the Extra PDL Area. Pointers into the Extra PDL Area are only allowed inside 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 inside 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.

SECTION 9

Function Calling

9.1 FUNCTIONAL OBJECTS

There are many kinds of functions in the Explorer system software. Each of them is described here. Note that these functional objects are also Lisp data objects that can be passed as an argument, returned, or stored in a variable or a data structure (eg. a list or an array). These Lisp data objects are special because they can be meaningfully applied to arguments, that is, used as functions.

When a list of the form (`<symbol> <args...>`) is evaluated, EVAL looks at the contents of `<symbol>`'s FUNCTION-CELL to decide how to evaluate the function. The way EVAL uses the contents of the FUNCTION CELL is called the interpretation of the datum in functional context. When a symbol is used as the destination of a CALL instruction, or the first argument to APPLY, its FUNCTION CELL is likewise examined and the contents considered in function context.

Function objects can be grouped into four categories by how they work. The four categories are summarized below.

First are interpreted functions: you define them with DEFUN or LAMBDA; they are represented as list structure and interpreted by the Lisp evaluator, EVAL.

Secondly, there are compiled functions: they are defined by compiling a function or by loading a XFAST file, represented by a special Lisp data type, and executed directly by the microcode. Similar to compiled functions are microcode functions, which are written in microcode (either by hand or by the micro-compiler) and executed directly by the hardware.

Thirdly, there are various types of Lisp objects that can be applied to arguments, but which when applied find some other function and apply it instead. These include select-methods, closures, instances, and entities.

Finally, there are various types of Lisp objects that, when used as functions, do something special related to the specific data type. These include arrays and stack-groups.

Here is what some of the data types mean in function context.

9.1.1 DTP-LIST Functions.

The evaluation of lists as functions should be handled by the interpreter. Usually the list is a LAMBDA expression. It can also be a MACRO expression. If a list is encountered in a call from a compiled function (eg. EVAL), the microcode calls out to the macrocoded interpreter which it finds via the support vector.

9.1.2 DTP-SYMBOL Functions.

When a symbol is encountered in a function context, the contents of the FUNCTION-CELL of the specified symbol is used as the function. The contents of the symbol's FUNCTION-CELL is gotten and the function interpretation mechanism is reinvoked (tail recursively) to interpret this object in the function context.

9.1.3 DTP-FEF-POINTER Functions.

This function is macro-compiled, so, the FEF (Function Entry Frame) it points to is used by the microcode. This is the kind of function most often encountered. Most of this chapter addresses the interpreting of FEF's.

9.1.4 DTP-U-ENTRY Functions.

This function is in microcode. A routine in microcode is called for this function; this pointer field is an index into the microcode entry area. If the microcode entry area contains a fixnum at that entry, it is the index into the microcode symbol area for this micro-compiled function. That entry in the microcode symbol area contains the address of the first microinstruction of the micro-compiled function. If the selected entry in the microcode entry area is not a fixnum, the function is not a currently installed micro-compiled function, and the entry is the function object to use instead.

9.1.5 DTP-ARRAY-POINTER Functions.

This function is an array. This is not really a function call, but is an array reference. The arguments to the array are the indices and the value is the contents of the element of the array. Array referencing is handled by the microcode, so there is no code associated with an array. The feature is for Maclisp compatibility and is not recommended usage. Use of AREF is recommended instead.

9.1.6 DTP-STACK-GROUP Functions.

Stack groups can be used as functions. The action taken when a stack group is called depends on the state of the called state group. Stack groups accept one argument. If the stack group called is resumed, the argument is transmitted. The calling stack group is placed in a resumable state. When it is resumed (not necessarily by the stack group it called) the object transmitted by that resumption will be returned as the value of the function call. Calling is one of the simple ways to resume a stack group.

9.1.7 DTP-INSTANCE Functions.

An instance is a message receiving object that has both state and a table of message-handling function, called methods. When an instance is called, the method for the message is located in the method table by examining the first argument. The instance variables are found and the selected method is applied to the arguments.

9.1.8 DTP-CLOSURE Functions.

A closure is a kind of function that contains another function and a set of special variable bindings. When the closure is applied, it puts the bindings into effect and then applies the other function. When that returns, the closure bindings are removed.

9.1.9 DTP-ENTITY Functions.

This obsolete data type acts like a closure, but also binds SELF like an instance.

9.1.10 DTP-STACK-CLOSURE Functions.

These are similar to DTP-CLOSURE.

9.1.11 DTP-LOCATIVE Functions.

These are treated like a list, i.e., invoke the interpreter to deal with it.

9.2 PDL LAYOUT

For each function call, a CALL BLOCK is stored on the PDL. The format of a call block is shown in Figure 9-1. The possible additional information is used by certain complex types of calls such as multiple-value calls which need to convey more information.

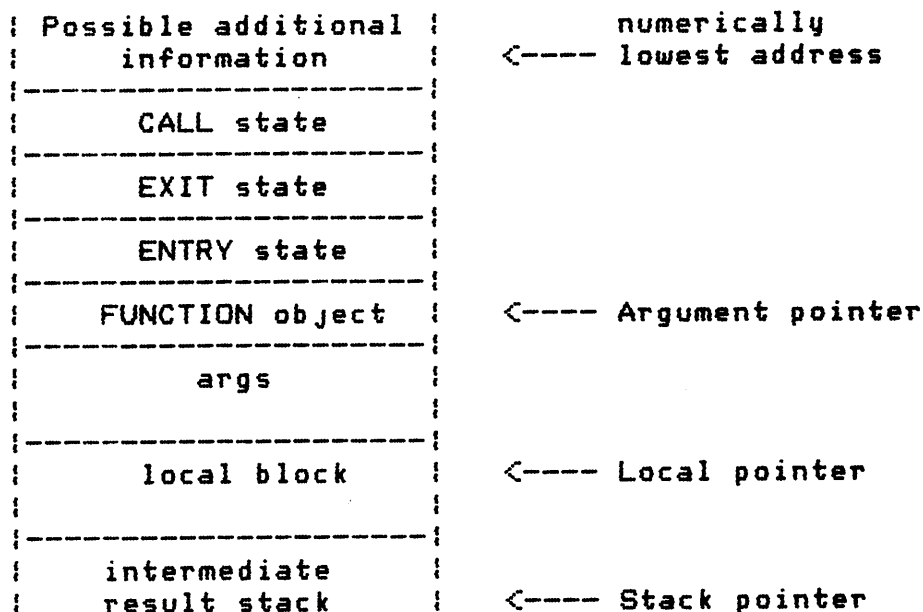


Figure 9-1 Call Block

The first four words contain various information used by the microcode which performs calls to and returns from functions. The arguments appear when instructions with destinations D-PDL and D-LAST are executed. When the block is activated, space is reserved for that block's local variables (i.e., LET and DO variables).

9.3 FUNCTION CALLING

Each CALL instruction creates a new open block. An open block consists of 3 state words, the function object, and an incomplete set of arguments (initially none). Call builds the state words and function object on the stack as described below.

The CALL instruction computes the CALL state word by computing the delta to the active block, the delta to the open block, and the destination code and packing these fields into the CALL state

word. The delta (offset) to the ACTIVE block at the time of the CALL (i.e., the function which called it) is in the low 8 bits. This is used to restore the argument pointer when leaving the function. The delta to the previous OPEN block (just the previous block on the stack) is in the next 8 bits. Its DESTINATION field is in another 4-bit field, so that when the called function returns, its result can be stored in the correct place.

The CALL instruction also reserves two words for the EXIT and ENTRY state words, and then pushes the FUNCTION object, which is typically a FEF pointer (DTP-FEF-POINTER, that is) when a macro-compiled function is being called.

When something is stored in destination D-LAST, the currently active block is exited (processing leaves it) and the current open call block (the last block pushed) is activated. The currently active block's PC is stored in the current block's EXIT state word as the return address, and the PC is set to the starting address of the new function. Also stored in the EXIT state word is the BINDS-PUT-BINDING-PDL bit.

Then the new block is entered, and in the low 8 bits of the new call block's ENTRY word, the relative location of the LOCAL BLOCK is stored. Also, in the next 6 bits of the ENTRY word is stored the number of args supplied to the new function.

When something is stored in destination D-RETURN, execution is finished in the current block and the value is to be returned as the value of this block. The microcode follows the pointer stored in the dying block's CALL state word to find its way back to the previous active call block, and then restores the PC from that block's EXIT state word where it was saved at exit time. The dying block is popped off the PDL and the value is sent to the destination saved in the CALL state word.

9.4 FEF LAYOUT

When a function is macro-compiled, the macrocompiler produces a Function Entry Frame (FEF). The FEF contains various things including random information about the function, symbols and constants used in the function, and the macrocode itself.

9.4.1 FEF Header.

The FEF begins with a word of DTP-FEF-HEADER, the format of which is shown in Figure 9-2.

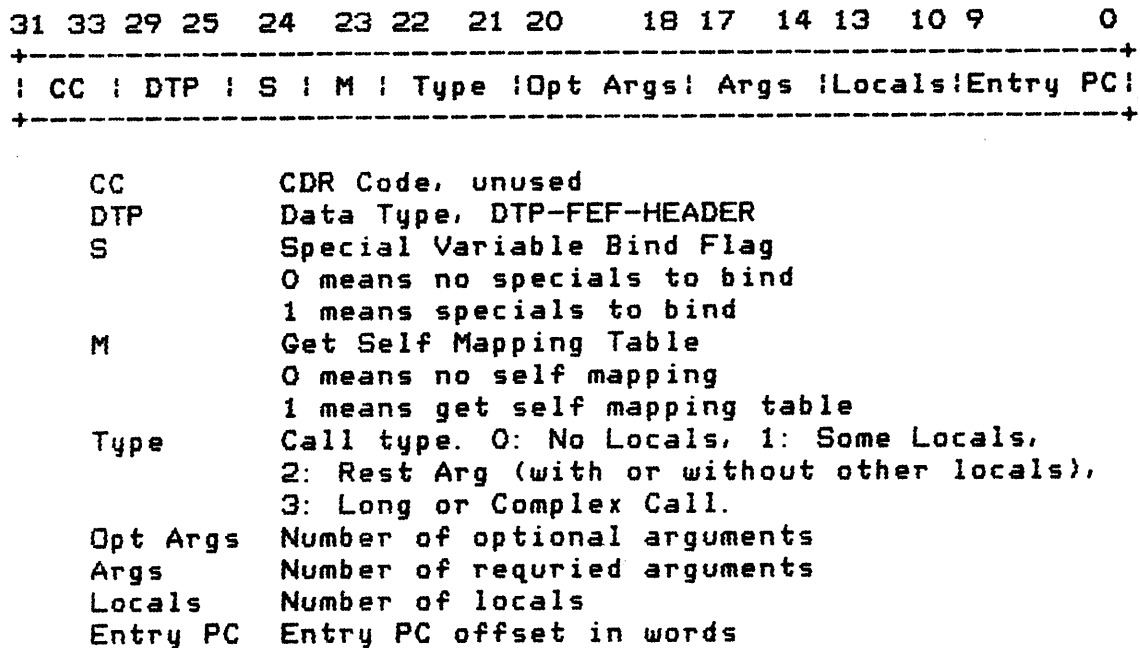


Figure 9-2 FEF Header Fields

9.4.2 FEF Storage Length.

The second word of the FEF is the storage length of the FEF. It is a FIXNUM which indicates the number of Gs occupied by the FEF. This is used for garbage collection and other system primitives that require the length of a structure.

9.4.3 FEF Name.

The third word of the FEF is the function name. This is used for debugging. It is usually a symbol, but may be a list such as (:METHOD FLAVOR :OPERATION).

9.4.4 Numeric Argument Descriptor.

The fourth word of the FEF is the FAST-ARG-OPT word, the format of which is shown in Figure 9-3. This word contains the numeric argument descriptor.

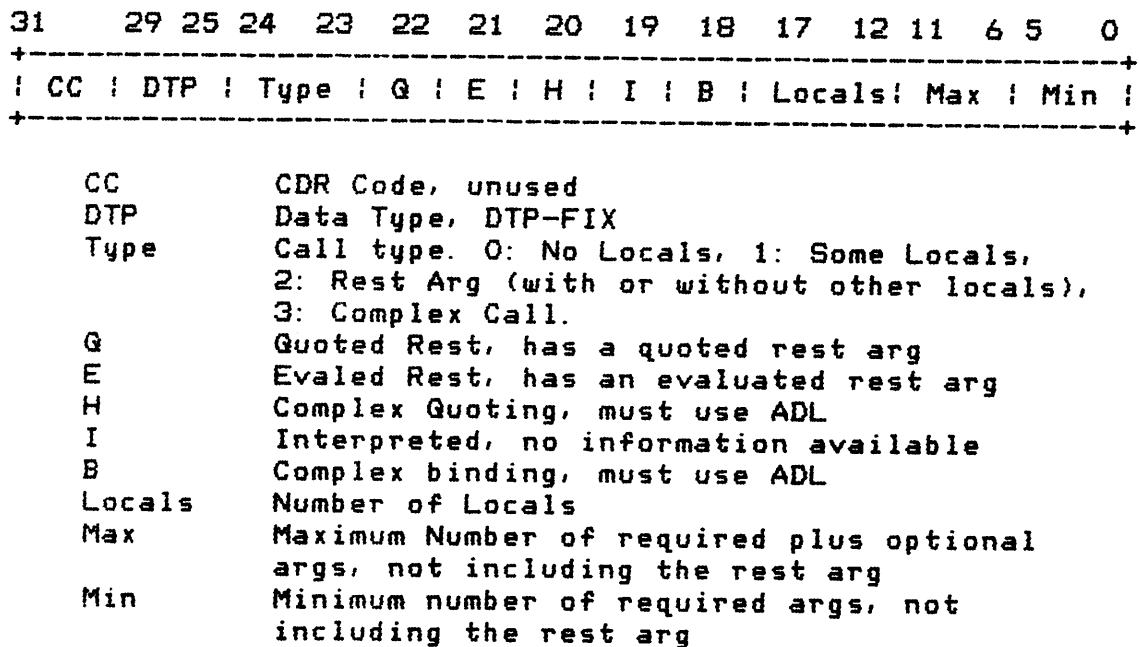


Figure 9-3 FEF Fast Argument Option Fields

9.4.5 Special Variable Bit Map.

The fifth word of the FEF is the SV-BITMAP word. The Special Variable Bit Map word contains one bit telling whether it is active, and also (if it is indeed active) 22 bits of bit map. If there are special variables bound by this function, but the word is not active, it is either because (1) there are more than 22 arguments+local vars, or (2) there is a &REST arg and so it is not clear how much room will be allocated on the stack for args, and therefore not clear where the local variables will end up. Therefore in this case, the information on whether various args and locals are special must be obtained from the Argument Descriptor List (ADL).

If the Special Variable Bit Map word is active, it is interpreted by considering it as a bit map, in which the most significant bit corresponds to the first variable.

9.4.6 FEF Miscellaneous.

The sixth word of the FEF is the MISC word. It contains miscellaneous flags and values related to the function. The sixth word has three fields as shown in Figure 9-4. Locals is the size of the local block (number of locals). When the

function is activated, this many words will be reserved on the PDL for local variables. ADL pos is the location of the ADL relative to the start of the FEF. ADL len is the length of the ADL in entries, i.e., the number of variables described. (Each entry may have one, two or three words.) See the paragraph on Argument Descriptor List.

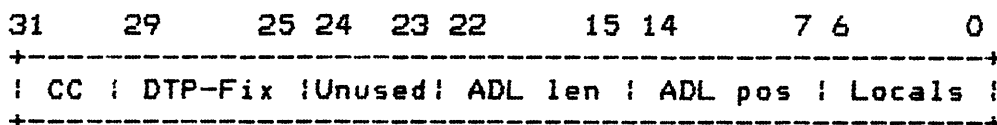


Figure 9-4 FEF Misc Word

9.4.7 Special Value Cell Pointers.

The seventh word of the FEF is the SPECIAL-VALUE-CELL-PNTRS word. This contains a list of pointers to value cells of the special variables indicated in the SV-BITMAP. If the SV-BITMAP is not active, this Q is irrelevant and contains NIL.

9.4.8 Optional FEF Header Words.

After the first seven words comes the variable FEF header. Each word in the variable FEF header is optional depending on some bit in the fixed header area above.

The MAPPING-TABLE-FLAVOR word is present if FEFH-GET-SELF-MAPPING-TABLE is set. The mapping table flavor is stored in the Q just before the ADL. This Q should contain a symbol.

The ADL is present unless FEFH-NO-ADL is set. The ADL is at the offset stored in the Misc word.

After this comes the symbols and constants used by the code. The most common entry in this section is an invisible pointer to the value or function cell of a symbol. DTP-External-Value-Cell-pointer is used for this type of invisible pointer. This scheme gives access to the current value or function definition of the symbol. Constants are also stored in this part of the FEF. Quoted symbols, numbers and constant valued lists are likely to be found here.

The word before the first word containing instructions holds the DEBUGGING-INFO list. This is an ALIST containing information such as the names of local variables that is useful for debugging.

Last, the macroinstructions for the function are stored starting in the word at offset Entry PC in words. The remainder of the FEF, starting with this word, is not typed, but contains two 16-bit macroinstructions per word. If the last word contains only one valid instruction, the odd halfword contains 0.

9.4.9 Function Entry.

There are three types of function entries: simple, long, and complex.

If the call type in the header word is not long or complex, then the call type is simple. If the SV-BITMAP bit is set then the simple call requires the Special Variable Bit Map word and the SPECIAL-VALUE-CELL-PNTRS word. If the FEFH-GET-SELF-MAPPING-TABLE bit is set then the simple call requires the MAPPING-TABLE-FLAVOR word. If it is the simplest case of the simple call, only the header word is consulted.

If the call type in the header word is long or complex, then the FAST-ARG-OPT word is consulted. If the call type in the FAST-ARG-OPT word is not complex, then the call type is long and the values in the FAST-ARG-OPT word override the values in the header word. As in the simple entry, the long entry may require the Special Variable Bit Map word and the SPECIAL-VALUE-CELL-PNTRS word and/or the MAPPING-TABLE-FLAVOR word.

If the call type in the FAST-ARG-OPT word is complex, then the call type is complex. Complex entries use the ADL for argument processing, which includes special bindings. Thus, a complex call never uses the Special Variable Bit Map word or the SPECIAL-VALUE-CELL-PNTRS word. However, a complex call may require the MAPPING-TABLE-FLAVOR word.

9.4.10 ARG Descriptor List.

When the ADL is used: If the FEF-QUOTE-HAIR bit is set, or the FEF-BIND-HAIR bit is set, or if the "S.V. bit map active" bit is clear and the Special Variables Bind bit is set, then the ADL must be present. (It may be present anyway for debugging purposes.) The absence of an ADL is indicated by an ADL length of zero.

Also, note that pass 2 of the macro-compiler always generates an ADL, and never the Numeric Arg Description word or the S.V. bit map word; the LAP program looks at the ADL, and determines what the Numeric Arg Description Word should be, and possibly creates an S.V. Bit map and possibly doesn't actually generate the ADL.

The format of the ADL is as follows: For each argument and each local variable there are either one, two, or three Q's in the

ADL. The first Q is numeric, and specifies just about everything about the variable in an encoded format. The second word is optional (presence indicated by a bit in the first Q), and stores the name of the variable (usually a pointer to a LISP atom). None of the code uses this; it is for debugging purposes only. The third Q, if present, is used to initialize the variable, under the control of various options specified by the first Q.

The fields of the first Q are shown in Figure 9-5.

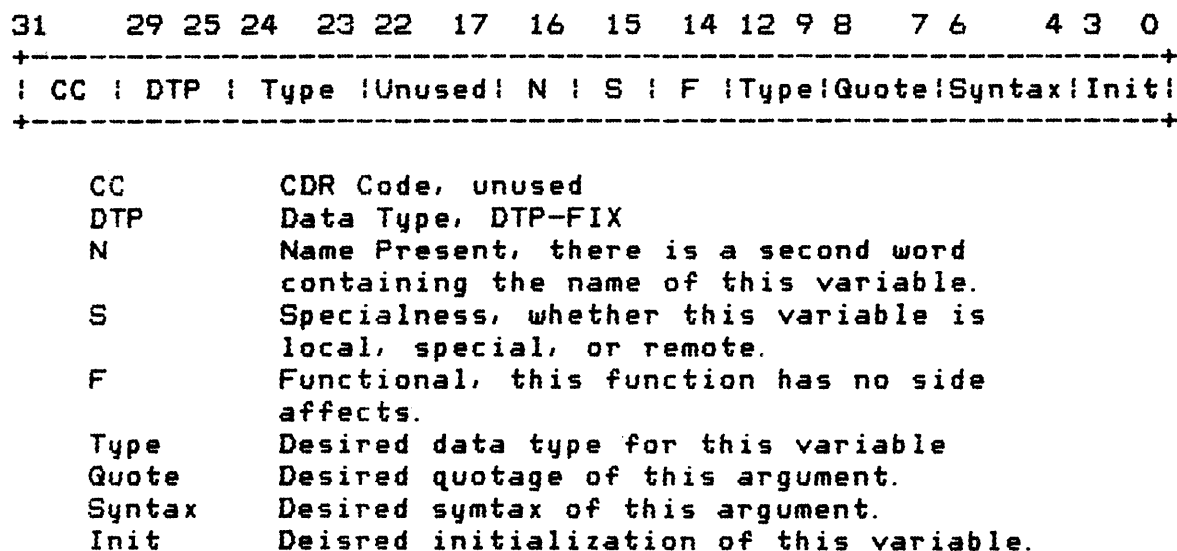


Figure 9-5 ADL First Q

When the macrocode refers to special variables, the actual code compiled will refer to an area in the FEF called the Special Variable Value Cell Pointer List (the effective addresses of the functions use the FEF "register" (or FEF+100 or FEF+200 etc.)). The pointer list contains invisible pointers to the value cells of the special variable themselves.

When a special variable is given as a local variable (a PROG or DO or &AUX variable) it must be bound. Instead of binding it by saving it on the Linear Binding PDL, the old values are saved in the slots in the Local Block on the main PDL, which would otherwise be unused. This is done for greater efficiency (sort of. Additional flavor would perhaps be a better description).

9.4.11 FEF Specialness.

If FEF-SPECIALNESS is odd, get a pointer to the variable's value cell from the next entry in the S.V. Value Cell Pointer List, and save the value in the Local Block of the PDL. The codes for FF Specialness are shown in Table 9-1.

Table 9-1 FEF Specialness

Value	Name	Meaning
0	FEF-LOCAL	Not special
1	FEF-SPECIAL	This variable is special
2	FEF-SPECIALNESS-UNUSED	
3	FEF-REMOTE	

9.4.12 Desired Data Type.

The desired datatype is specified in a 4-bit field, encoded as shown in Table 9-2.

Table 9-2 Desired Data Type

Value	Name	Meaning
0	DT-DONTCARE	We don't care what we get.
1	DT-NUMBER	Any number.
2	DT-FIXNUM	Only FIXNUM.
3	DT-SYM	Only SYMBOL.
4	DT-ATOM	Any number or symbol.
5	DT-LIST	Only LIST.
6	DT-FRAME	Only FRAME (i.e. FEF)

9.4.13 Quote Status.

The encoding of quote status is shown in Table 9-3.

Table 9-3 Quote Status

Value	Name	Meaning
0	FEF-QT-DONTCARE	We don't care what we get.
1	FEF-QT-EVAL	Should be EVALed.
2	FEF-QT-QT	Should be QUOTEd (not EVALed).
3	FEF-QT-UNUSED	

9.4.14 Argument Syntax.

FEF-ARG-SYNTAX is a 3-bit field that may take on the values shown in Table 9-4.

Table 9-4 Argument Syntax

Value	Name	Meaning
0	FEF-ARG-REQ	Required.
1	FEF-ARG-OPT	Optional. May be initialized. if arg not present.
2	FEF-ARG-RESET	Rest arg. (may only be one).
--- below here, not really arguments		
3	FEF-ARG-AUX	Prog-variable. May be initialized.
--- below here, ignored by function entry operation		
4	FEF-ARG-FREE	Variable is referenced free. Might be nice to know but not used.
5	FEF-ARG-INTERNAL	cell used to pass an argument to an internal LAMBDA.
6	FEF-ARG-INTERNAL-AUX	cell used by an internal PROG.

9.4.15 Init Option.

FEF-INIT-OPTION is a 4-bit field which may take the values shown in Table 9-5.

Table 9-5 Quote Status

Value	Name	Meaning
0	FEF-INIT-NONE	Do not initialize (All required args have this)
1	FEF-INIT-NIL	Initialize NILZ(Default for locals)
2	FEF-INIT-PNTR	Initialize variable to 3rd Q
3	FEF-INIT-C-PNTR	Initialize variable to the object pointed to by the 3rd Q
4	FEF-INIT-OPT-SA	Optional starting address. Start function here if this optional arg is supplied. Code between normal starting address and here initializes variable if it is not supplied.
5	FEF-INIT-COMP-C	Variable initialized by compiled code. Initialization too hairy to be done by above mechanisms.
6	FEF-INIT-EFF-ADR	Interpret 3rd Q as macro-code effective address (i.e. 3-bit register, 6-bit delta). Reference that address and initialize variable to what you get. [This is used to compile (LAMBDA (A &OPTIONAL (B A)) ...) with A and B local, for example.]
7	FEF-INIT-SELF	Initialize to self. Used for (LAMBDA (&OPTIONAL (FOO FOO)) ...) which is not reasonable unless FOO is special.

9.4.16 FEF Constants.

If the macro-compiled program uses constants, the code generated will be either of two things; if the constant is one of a few which many programs use, such as NIL, T, and some small numbers, it may be on the Constants page, and the code addresses it with the Constants page "register". But if it is a constant most likely only used by this function, the constant will be placed in the FEF in an area following the ADL. The macro-compiler will, in both cases, generate a reference called QUOTE-VECTOR; it is the LAP program which actually decides whether to reference the Constants page, or to create a new constant in the FEF and reference it instead.

SECTION 10

Flavors

10.1 INTRODUCTION

There is a great deal of support in Lisp Machine Lisp for the flavors object-oriented programming system. That support will be detailed here. You will notice that this support has very little that makes it flavor-specific: instead the support for instances of classes is more flexible to allow for other object-oriented programming systems. This flexibility is not currently used.

Special support is provided to keep the special variables SELF and SELF-MAPPING-TABLE bound to the currently executing instance and the mapping table to map the actual positions of the instance variables into slot numbers used by this method.

10.2 INSTANCE DATA STRUCTURE

A DTP-INSTANCE Lisp object points to a structure whose header is of type DTP-INSTANCE-HEADER. The pointer field of that header points to a structure (generally an array) which contains the fields described below. This structure is called an instance-descriptor and contains the constant or shared part of the instance. The instance structure, after its DTP-INSTANCE-HEADER, contains several words used as value cells of instance variables, which are the variable, state, or unshared part of the instance.

10.3 INSTANCE DESCRIPTOR DATA STRUCTURE

All instances of a flavor share an instance descriptor which is an ART-Q array or other structure. The instance descriptor contains information that is the same for all instances of a flavor and is not really dynamic (although some dynamic changes can be accommodated). The elements of the instance descriptor are as indicated in Table 10-1. Note that these are offsets, not indices into the array. They are defined here this way because microcode uses them. This could be a CDR-coded list or an instance rather than an array.

The word pointed to by the instance header is %INSTANCE-DESCRIPTOR-HEADER. It is usually an array header. The next

word, %INSTANCE-DESCRIPTOR-RESERVED, is not used by instance support and may be used by the containing structure (e.g. for named-structure symbol).

Table 10-1 Instance Descriptor Offsets

0	%INSTANCE-DESCRIPTOR-HEADER
1	%INSTANCE-DESCRIPTOR-RESERVED
2	%INSTANCE-DESCRIPTOR-SIZE
3	%INSTANCE-DESCRIPTOR-BINDINGS
4	%INSTANCE-DESCRIPTOR-FUNCTION
5	%INSTANCE-DESCRIPTOR-TYPENAME
6	%INSTANCE-DESCRIPTOR-MAPPING-TABLE-ALIST
7	%INSTANCE-DESCRIPTOR-IGNORE
8	%INSTANCE-DESCRIPTOR-ALL-INSTANCE-VARIABLES
9	%INSTANCE-DESCRIPTOR-IGNORE
10	%INSTANCE-DESCRIPTOR-IGNORE
11	%INSTANCE-DESCRIPTOR-IGNORE
12	%INSTANCE-DESCRIPTOR-IGNORE
13	%INSTANCE-DESCRIPTOR-IGNORE
14	%INSTANCE-DESCRIPTOR-DEPENDS-ON-ALL

The third word is %INSTANCE-DESCRIPTOR-SIZE. This is the size of each instance; this is one more than the number of instance-variable slots. When the garbage collector needs to copy or scavenge an instance, it refers to this slot of the instance descriptor.

%INSTANCE-DESCRIPTOR-BINDINGS describes bindings to perform when the instance is called. If this is a list, then SELF is bound to the instance and the elements of the list are locatives to cells which are bound to External Value Call Pointers (EVCPs) to successive instance-variable slots of the instance. If this is not a list, it is something reserved for future facilities based on the same primitives. NIL is a list and so binds SELF and nothing else. If the a list element is a FIXNUM rather than a locative, it is the number of slots to skip over and not bind; a FIXNUM is not allowed to be the last element of the list. Note that if this is a list, it must be CDR-CODED. The microcode depends on this for a little extra speed.

Next is %INSTANCE-DESCRIPTOR-FUNCTION which is the function to be called when the instance is called. Typically a hash table.

Next is the %INSTANCE-DESCRIPTOR-TYPENAME which is a symbol. This is what is returned by TYPEP.

%INSTANCE-DESCRIPTOR-MAPPING-TABLE-ALIST is an alist of mapping tables to instances of this descriptor for various method-flavors.

All slots mared `%INSTANCE-DESCRIPTOR-IGNORE` are used only at higher levels.

`%INSTANCE-DESCRIPTOR-ALL-INSTANCE-VARIABLES` is a list of all instance variables in the same order as they appear in the instance.

Last is `%INSTANCE-DESCRIPTOR-DEPENDS-ON-ALL` which is a list of all component flavors names. This is used by `TYPEP-STRUCTURE-OR-FLAVOR`.

10.4 SELF MAPPING TABLE

A self mapping table is used to map slot numbers used by a method into actual positions within the instance.

It is highly desirable that when a method is inherited by more than one flavor, that the FEF for that method be shared. This can dramatically reduce the space used to represent a large network of related flavors. However, it is not possible to produce a single ordering of instance variables that all combined flavors can see a consistent ordering of the instance variables they share.

Instead, for each flavor there is a table that maps instance variables to their actual positions within the instance. This self mapping table is an ART-16B array. To access an instance variable in slot *n*, element *n* of the array is read and is used as the index into the instance to reference the variable.

When flavors are composed, a mapping table is created for each flavor from which this flavor inherits methods. Along with the method to call, the mapping table for the flavor from which this method is inherited is stored in the hash table which serves as the method decode table.

Instance variables may be accessed directly if no self-mapping table is needed. If the flavor has `:ORDERED-INSTANCE-VARIABLES`, the methods know which position will contain each instance variable. They can then access the variables without using the self-mapping table.

If the flavor is a base flavor that inherits no methods and the instance is of that flavor, the instance variables are in the same order as the slots. In this case, there is a special form of mapping-table that provides the identity map. It is represented as `NIL`. If `SELF-MAPPING-TABLE` is `NIL` mapped access to an instance variable acts like an unmapped access.

Each mapping table contains in its leader slot 1, a pointer to the instance descriptor for the method flavor it maps.

10.5 METHOD DECODE TABLE

The method decode table is stored in the %INSTANCE-DESCRIPTOR-FUNCTION slot of the instance-descriptor. It is a callable hash array, which means that it is a named structure with a leader and the FUNCALL-AS-HASH-TABLE bit set. One leader slot contains the size of the hash table in entries. This size is assumed to be a power of two. These entries are 3 words each.

Entries in the hash array are 3 words long. The first word is the key which matches this entry. The second word is a locative to a value (or function) cell containing the function to call. The third entry if non-NIL is the self mapping table to install before calling the function in the second word.

More on the hash array is explained in the next section. The use of the hash array for decoding and calling a method is explained there.

10.6 CALLING AN INSTANCE

When an instance is called, SELF is bound to the instance. Then the pointer to the instance-descriptor is gotten from the instance header. The bindings list is read from the instance descriptor. If it is a list, the bindings on the list are done as described above. If it is NIL, no additional bindings are done. If it is not a list or NIL, then it is for some unsupported object protocol and an error is signalled.

Next the function is read from the instance-descriptor. If it is an array, it is a callable hash array. If it is not, perform a normal function entry on it. To call a hash array, the number of entries in the hash table, n , which is assumed to be a power of two. The key is the first argument to the instance which is a symbol. $(n - 1)$ is used as a bit mask to get the low bits of the key. This is used as the index of the first candidate entry in the hash array.

If the first word of a candidate entry does not match (is not EQ to) the key, it is either some other key or DTP-NULL. If it is DTP-NULL, the key could not be found and there has been a hash failure. A special function, SI:INSTANCE-HASH-FAILURE is called via the support vector. In some cases such as when the hash array has been forwarded or references a key in old space, the method failure function is called when the method is actually in the table but the table needs moving or rehashing.

If the first word of a candidate entry does not match the key and is not DTP-NULL, a rehash is required. A hash array rehashes by advancing to the next entry with wrap around. Every hash table must have at least one entry of DTP-NULL; hash tables are usually

maintained with many more empty entries, since their number of entries must be a power of two.

When the key is found, the method will be called. The second word is read and the locative followed to read the function. If the third word is non-NIL SELF-MAPPING-TABLE is bound to what it contains. If the function is not a symbol, the flag is set to indicate that the self-mapping table is supplied. This flag indicates to function entry that there is no need to search for and set SELF-MAPPING-TABLE. Finally, the function is called as normal.

10.7 INSTANCE VARIABLE ACCESSING

Several ways are provided of accessing instance variables. Self-reference pointers can be used to access instance variables either mapped or unmapped. See paragraph on self-reference pointers in the Internal Storage Format section for more details.

There is an addressing mode provided in macroinstructions that allows access to instance variables.

There are several miscellaneous instructions that provide access to instance variables. They are %INSTANCE-REF, SET-%INSTANCE-REF, %INSTANCE-SET, and %INSTANCE-LOC.

SECTION 11

Stack Groups

11.1 INTRODUCTION

This chapter explains stack groups, the building blocks of multiprocessing.

A stack group is the 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 could be activated to handle it.

11.2 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 an ARRAY-POINTER 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 data section of the array holds the main PDL for the stack group. The array leader holds many other relevant data including 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 SPEC PDL; 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 occurred 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.

11.3 SPECIAL PDL

The Special PDL is used to hold saved bindings of special variables. The LISP machine 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 (SPC) of the machine is stored there (this is needed because the hardware SPC is of a small fixed size).

The Special PDL is block oriented. The blocks are delimited by setting the SPEC PDL-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 11-1.

SPEC PDL-BLOCK-START-FLAG and SPEC PDL-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. SPEC PDL-CLOSURE-BINDING indicates that this binding was made "before" entering the function (ie. by closure binding, or by the binding of SELF for a method). SPEC PDL-BLOCK-START-FLAG is bit 31.

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 Q or LIST).

The processor micro-stack (SPC) 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 QBBFL bit in the PC status flags is turned on if a binding block has been opened on the Special PDL. This bit is saved during macro-to-macro calls on the regular PDL in the exit state word. It is restored when returning to a call block. This assures that when a compiled function is done, QBBFL 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 binding of the top-most block of the Special PDL must be undone. If not on, it means that not one pair has yet been pushed.

Micro-to-micro calls can also cause bindings, and in order to keep that straight, a bit on the SPC 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-GLBNP when the stack group is executing on processor. There is an area devoted to storing the Special PDL's called LINEAR-BIND-PDL-AREA.

Table 11-1 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 (SPC). Each word in the block should be a fixnum containing the old contents of the SPC. Only the active part of the stack is transferred.

SECTION 12

Subprimitives

12.1 SUBPRIMITIVES

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 on other systems. In fact, most subprimitive functions are simple lisp-callable interfaces to miscellaneous operation macro instructions (miscops). These miscops are generally written in Explorer microcode.

Subprimitives by their very nature cannot do full checking. Improper use of subprimitives can destroy the environment. Subprimitives 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.

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. For more information about these storage conventions, see the section on garbage collection. If you create such an illegal pointer, it probably will not be detected immediately. Later on, however, parts of the system (in particular the garbage collector) may see it, notice that it is illegal, and halt the machine. Of course you can also use subprimitives 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, car, cdr, rplaca, and rplacd can also 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. Extreme caution should be exercised when using any of these functions.

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.

12.2 DATA TYPES

`data-type (arg)`

`data-type` returns a symbol that is the name for the internal data-type of `arg`. The section on internal storage format contains a list of all internal data types along with a description of each. 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, `ntp-extended-number` is the symbol that `data-type` would return for either a single-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 List object. These include header types such as `ntp-symbol-header`, which identify the first word of a structure, and forwarding or "invisible" pointer types such as `ntp-one-q-forward`.

`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 "ntp-". The values of these symbols are the internal numeric data-type codes for the various types.

`q-data-types(type-code)`

Given the internal numeric data-type code, returns the corresponding symbolic name. This "function" is actually an array.

12.3 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. 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 `ntp-one-q-forward`. It is used to forward a single word of memory to someplace else. The invisible pointers with data types `ntp-header-forward` and `ntp-body-forward` are used for moving whole Lisp objects (such as cons cells or arrays) somewhere else. The `ntp-external-value-cell-pointer` is similar to `ntp-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 `ntp-external-value-cell-pointer` that points to some other word (the external value cell), then `symsval` 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 `ntp-external-value-cell-pointer` itself, and stores the new value into the internal value cell of the symbol. This is how closures are implemented.

`ntp-gc-forward` is not an invisible pointer at all; it only appears in "old space" and can never be seen by any program other than the garbage collector. When an object is found not to be garbage, and the garbage collector moves it from "old space" to "new space", a `ntp-gc-forward` is left behind to point to the new copy of the object. This ensures that other references to the same object get the same new copy.

`structure-forward (old-object) (new-object)`

This causes references to `old-object` actually to reference `new-object`, by storing invisible pointers in `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 returns `object`, but if `object` has been structure-forward'ed, returns the object at the end of the chain of forwardings. 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 `ntp-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-tag-and-pointer
  locative1 ntp-one-q-forward locative2)
```

`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 `t`, external value cell pointers are followed; if it is `nil` they are not.

12.4 POINTER MANIPULATION

It should again be emphasized that improper use of these functions can damage or destroy the Lisp environment. It is possible to create pointers with illegal data types, pointers to non-existent objects, and pointers to untyped storage. Any of these can cause problems for systems programs such as the garbage collector.

%data-type (x)

Returns the data-type field of x, as a fixnum.

%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)

Makes up a pointer, with data-type in the data-type field and pointer in the pointer field, and returns it. data-type should be an internal numeric data-type code; these are the values of the symbols that start with dtp-. pointer may be any object; its pointer field is used. This is most commonly used for changing the type of a pointer. Do not use this to make pointers which are not allowed to be in the machine, such as dtp-null, invisible pointers, etc.

%make-pointer-offset (data-type pointer offset)

Returns a pointer with data-type in the data-type field, and pointer plus offset in the pointer field. The data-type and pointer arguments are like those of %make-pointer; offset may be any object 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, note that it is illegal to have a pointer to untyped data, such as the inside of a FEF or a numeric array.

%pointer-difference (pointer-1 pointer-2)

Returns a fixnum which is pointer-1 minus pointer-2. 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.

%pointerp (object)

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

`%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.

12.5 SPECIAL 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.

`%p-pointerp-offset (location offset)`

Similar to `%p-pointerp` but operates on the word offset words beyond location.

`%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

There is no `%p-contents`, since `car` performs 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`).

`%p-contents-as-locative (pointer)`

Given a pointer to a memory location containing a pointer that isn't allowed to be "in the machine" (typically an invisible pointer) this function 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.

%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.

%p-safe-contents-offset (location offset)

Returns the contents of the word offset words beyond location as accurately as possible without getting an error.

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.

Forwarding pointers are checked as in %p-contents-offset.

%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-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. This is a good way to store a forwarding pointer from one structure to another (for example).

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

%p-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. For example, (%p-ldb %p-cdr-code loc) returns the cdr-code of the location addressed by loc.

%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 like %p-contents-offset.

This is the way to reference byte fields within a structure without violating system storage conventions.

%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.

%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.

NOTE

%p-ldb-offset, %p-dpb-offset, %p-deposit-field and %p-deposit-field-offset should never be used to modify the pointer field of a boxed word if the data type is one which actually points to storage, unless you are sure that the new pointer is such as to cause no trouble (which would be the case, for example, if the new pointer will point to a static area). Likewise, it should never be used to change a data type which does not point to storage into one which does. Either action could confuse the garbage collector.

%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. You can use this to alter data types, cdr-codes, etc., but see the note above for restrictions.

%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 in the location addressed by base-pointer as in %p-contents-offset. nil is returned.

This is the way to alter unboxed data within a structure without violating system storage conventions. You can use this to alter boxed words too, but see the note above for restrictions.

%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.

%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.

%p-pointer (pointer)

Extracts the pointer field of the contents of the location addressed by pointer and returns it as a fixnum.

%p-data-type (pointer)

Extracts the data-type field of the contents of the location addressed by pointer and returns it as a fixnum.

%p-cdr-code (pointer)

Extracts the cdr-code field of the contents of the location addressed by pointer and returns it as a fixnum.

%p-store-pointer (pointer value)

Stores value in the pointer field of the location addressed by pointer, and returns value.

%p-store-data-type (pointer value)

Stores value in the data-type field of the location addressed by pointer, and returns value.

%p-cdr-code (pointer value)

Stores value in the cdr-code field of the location addressed by pointer, and returns value.

%stack-frame-pointer

Returns a locative pointer to its callers 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 "callers stack frame" really means "the frame for the FEF that executed the %stack-frame-pointer instruction".

12.6 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 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 alloc 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 alloc 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 alloc.

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.

12.7 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. They are not documented in this manual since they are in the system package rather than the global package. The variables whose names start with %% are byte specifiers, intended to be used with subprimitives such as %p-ldb.

NOTE

Changing the values of these special variables will almost certainly cause the machine to crash.

%%q-cdr-code	Constant
The field of a memory word that contains the cdr-code.	
%%q-data-type	Constant
The field of a memory word that contains the data-type code.	
%%q-pointer	Constant
The field of a memory word that contains the pointer address or immediate data.	
%%q-pointer-within-page	Constant
The field of a 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.	

<code>%%q-high-half</code>	Constant
<code>%%q-low-half</code>	Constant
The halves of a memory word. These fields are only used in storing compiled code.	
<code>cdr-normal</code>	Constant
<code>cdr-next</code>	Constant
<code>cdr-nil</code>	Constant
<code>cdr-error</code>	Constant
The values of these four variables are the numeric values that go in the <code>cdr-code</code> field of a memory word.	

12.8 ANALYZING STRUCTURES

`%find-structure-header (pointer)`

This subprimitive finds the structure into which `pointer` points, by searching backward for a header. It is a basic low-level function used by such things as the garbage collector. `pointer` is normally a locative, but its data-type is ignored. Note that it is illegal to point into an "unboxed" portion of a structure, for instance the middle of a numeric array.

In structure space, the "containing structure" of a pointer is well-defined by system storage conventions. In list space, it is considered to be the contiguous, `cdr`-coded segment of list surrounding the location pointed to. If a cons of the list has been copied out by `rplacd`, the contiguous list includes that pair and ends at that point.

`find-structure-header (pointer)`

This primitive finds the structure into which `pointer` points, by searching backward for a header. `pointer` is normally a locative, but its data-type is ignored. Note that it is illegal to point into an "unboxed" portion of a structure, for instance the middle of a numeric array. Structure forwarding is followed.

In structure space, the "containing structure" of a pointer is well-defined by system storage conventions. In list space, it is considered to be the contiguous, `cdr`-coded segment of list surrounding the location pointed to. If a cons of the list has been copied out by `rplacd`, the contiguous list includes that pair and ends at that point.

%find-structure-leader (pointer)

This is 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-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 normal 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 boxed, but their contents are considered garbage and are not protected by the garbage collector.

%structure-total-size (object)

Returns the total number of words occupied by the representation of object, including boxed Q's, unboxed Q's, and garbage Q's off the ends of PDLs.

find-structure-leader (pointer)

This is 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. If pointer is an address in a forwarded structure, all structure forwarding is followed first.

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 normal 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 boxed, but their contents are considered garbage and are not protected by the garbage collector. Structure forwarding is followed.

structure-total-size (object)

Returns the total number of words occupied by the representation of object, including boxed Q's, unboxed Q's, and garbage Q's off the ends of PDLs. Structure forwarding is followed.

12.9 CREATING OBJECTS

`%allocate-and-initialize` (data-type header-type header second-word area size)

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 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 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 is different from `%allocate-and-initialize` because arrays have a more complicated header structure.

The basic functions for creating list-type objects are `cons` and `make-list`; no special subprimitive is needed. Closures, entities, and `select-methods` are based on lists, but there is no primitive for creating them. To create one, create a list and then use `%make-pointer` to change the data type from `dtp-list` to the desired type.

12.10 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. By the same token, it is a potentially dangerous operation; if any pointers to the list remain after the function exits, they will become meaningless. 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.

`with-stack-list` (variable element...) body... Special Form

`with-stack-list* (variable element...) body...` Special Form
 These special forms create stack lists that live inside the stack frame of the function that they are used in. You should assume that the stack lists are only valid until the special form is exited.

```
(with-stack-list (foo x y)
  (mumblify foo))
```

is equivalent to

```
(let ((foo (list x y)))
  (mumblify foo))
```

except for the fact that `foo`'s value in the first example is a stack list.

The list created by `with-stack-list*` looks like the one created by `list*`. `tail`'s value becomes the ultimate `cdr` rather than an element of the list.

Here is a practical example. `condition-resume` might have been defined as follows:

```
(defmacro condition-resume (handler &body body)
  '(with-stack-list* (eh:condition-resume-handlers
                    ,handler eh:condition-resume-handlers)
    ,body))
```

It is an error to do `rplacd` on a stack list (except for the tail of one made using `with-stack-list*`). `rplaca` works normally.

`sys:rplacd-wrong-representation-type (error)` Condition
 This is signaled if you `rplacd` a stack list (or a list overlaid with an array or other structure).

12.11 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 checks that field, interfacing suitably with the garbage collector if necessary. %blt does not check the data type fields of the copied words.

%blt may be used on any data except boxed data containing pointers to storage, while %blt-typed may be used on any boxed data. Both %blt and %blt-typed can be used validly on data which is formatted with data types (boxed) but whose contents never point to storage. 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 the machine is told to examine the data types of such data makes no difference since, on examining them, it would decide that nothing needed to be done.

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

For boxed data which may contain pointers, only %blt-typed may be used. If %blt were used, it would appear to work, but problems could appear mysteriously later because %blt would fail to notice the presence of the pointer there. For example, the pointer might point to a bignum in the number consing area, and moving it in this way would fail to copy the bignum out into a nontemporary area (as required by the number consing area garbage collection rules). Then the pointer would become invalidated the next time the number consing area was emptied out. There could also be problems with lexical closures and garbage collection.

12.12 RETURNING STORAGE

`return-storage (object)`

This function attempts to return object to free storage. If it is a displaced array, this returns the displaced array itself, not the data that the array points to. Currently `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 area, nor the most recently allocated list in its area.

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 (progn 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`.

12.13 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.

12.14 EXPLORER I/O DEVICE SUBPRIMITIVES

`sys:%nubus-read (slot byte-address)`

Returns the contents of a word read from the Nu bus. Addresses on the Nu bus 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 512k main memory board, then the valid address range from 0 to $4 * (512k - 1)$. (Of course, main memory boards are normally accessed through the virtual memory mechanism.)

`sys:%nubus-write (slot byte-address word)`

Writes word into a word of the Nu bus, whose address is specified by slot and byte-address as described above.

`sys:%nubus-physical-address (apparent-physical-page)`

The valid portions of the Nubus address space are not contiguous. Each board is allocated 16m bytes of address space, but no memory board actually provides 16m bytes of memory.

The Lisp Machine virtual memory system maps virtual addresses into a contiguous physical address space. On the Explorer, this contiguous address space is mapped a second time into the discontinuous Nu bus address space. Unlike the mapping of virtual addresses to physical ones, the second mapping is determined from the hardware configuration when the machine is booted and does not change during operation.

This function performs exactly that mapping. The argument is a physical page number (a physical address divided by `sys:page-size`). The argument is a "Nu bus page number"; multiplied by `sys:page-size` and then by four, it yields the Nu bus byte address of the beginning of that physical page.

12.15 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.

NOTE

These functions may not be supported in future releases due to improvements planned in the way function calling is implemented.

%open-call-block (function n-adi-pairs destination)

Starts a call to function. n-adi-pairs is the number of pairs of additional information words already %push'ed; normally this should be 0. destination is where to put the result; the useful values are 0 for the value to be ignored, 1 for the value to go onto the stack, 3 for the value to be the last argument to the previous open call block, and 2 for the value to be returned from this frame.

%push (value)

Pushes value onto the stack. Use this to push each argument.

%activate-open-call-block

Causes the call to happen.

%pop

Pops the top value off of the stack and returns it as its value. Use this to recover the result from a call made by %open-call-block with a destination of 1.

%assure-pdl-room (n-words)

Call this before doing a sequence of %push's or %open-call-block'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.

12.16 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.

12.17 CLOSURE SUBPRIMITIVES

These functions are used to implement 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.

`sys:%binding-instances (list-of-symbols)`

This is the primitive that could be used by the closure function. First, if any of the symbols in `list-of-symbols` has no external value cell, a new external value cell is created for it, with the contents of the internal value cell. Then a list of locatives, twice as long as `list-of-symbols`, is created and returned. The elements are grouped in pairs: pointers to the internal and external value cells, respectively, of each of the symbols. closure could have been defined by:

```
(defun closure (variables function)
  (%make-pointer dtp-closure
    (cons function (sys:%binding-instances variables))))
```

`sys:%using-binding-instances (instance-list)`

This function is the primitive operation that invocation of closures could use. It takes a list such as `sys:%binding-`

instances returns, 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.

sys:%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 sys:%using-binding-instances.

sys:%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.

12.18 PAGING SYSTEM SUBPRIMITIVES

The subprimitives and special variables described below affect the paging algorithms of the virtual memory system.

sys:%disk-switches

variable

This variable contains bits that control various disk usage features.

Bit 0 (the least significant bit) enables read-compares after disk read operations. This causes a considerable slowdown, so it is rarely used.

Bit 1 enables read-compares after disk write operations.

Bit 2 enables the multiple page swap-out feature. When this is enabled, as it is by default, each time a page is swapped out up to 16, contiguous pages are also written out to the disk if they have been modified. This greatly improves swapping performance.

Bit 3 controls the multiple page swap-in feature, which is also on by default. This feature causes pages to be swapped in in groups: each time a page is needed, several contiguous pages are swapped in in the same disk operation. The number of pages swapped in can be specified for each area using si:set-swap-recommendations-of-area.

si:set-swap-recommendations-of-area (area-number recommendation)
Specifies that pages of area area-number should be swapped in in groups of recommendation at a time. This recommendation is used only if the multiple page swap-in feature is enabled.

Generally, the more memory a machine has, the higher the swap recommendations should be to get optimum performance. The recommendations are set automatically according to the memory size when the machine is booted.

si:set-all-swap-recommendations (recommendation)
Specifies the swap-in recommendation of all areas at once.

si: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.

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

sys: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, using as few disk operations as possible. Consecutive disk pages are transferred together, taking advantage of the full speed of the disk. If object is large, this is much faster than bringing the pages in one at a time on demand. The storage occupied by object is defined by the %find-structure-leader and %structure-total-size subprimitives.

sys:page-in-array (array &optional from to)
This is a version of sys: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.

sys:page-in-pixel-array (array &optional from to)
Like sys: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.

sys: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 in with as few disk operations as possible.

sys:page-in-area (area-number)

sys:page-in-region (region-number)

All swapped-out pages of the specified region or area are brought into main memory.

sys:page-out-structure (object)

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

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

sys:page-out-words (address n-words)

sys:page-out-area (area-number)

sys:page-out-region (region-number)

These are similar to the above, except that they take pages out of main memory rather than bringing them in. Actually, they only mark the pages as having priority for replacement by others. Use these operations when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

sys:%page-status (virtual-address)

If the page containing virtual-address is swapped out, or if it is part of one of the low-numbered fixed areas, this returns nil. Otherwise it returns the entire first word of the page hash table entry for the page.

The %%pht1- symbols in SYS:COLD-BAND;QCOM.LISP are byte specifiers you can use with %logldb for decoding the value.

sys:%change-page-status (virtual-address swap-status)

access-status-and-meta-bits

The page hash table 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.

sys:%compute-page-hash (virtual-address)
Makes the hashing function for the page hash table available to the user.

sys:%physical-address virtual-address)
Returns the physical address which virtual-address currently maps into. The value is unpredictable if the virtual page is not swapped in; therefore, this function should be used on wired pages, or you should do

(without-interrupts
(%p-pointer virtual-address) ;swap it in
(sys:%physical-address virtual-address))

sys:%create-physical-page (physical-address
This is used when adjusting the size of real memory available to the machine. It adds an entry for the page frame at physical-address to the page hash table, with virtual address -1, swap status flushable, and map status 120 (read only). This doesn't make error checks; you can really screw things up if you call it with the wrong arguments.

sys:%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.

sys:%disk-save (physical-mem-size high-16-bits low-16-bits)
Copies virtual memory into the partition named by the concatenation of the two 16-bit arguments (0 means the default), then restarts the world, as if it had just been restored. The physical-mem-size argument should come from %sys-com-memory-size in system-communication-area. If physical-mem-size is negative, it is minus the memory size, and an incremental save is done. This is the primitive used by the disk-save function.

si:set-memory-size (nwords)
Specifies the size of physical memory in words. The Explorer system determines the actual amount of physical memory when it is booted, but with this function you can tell it to use less memory than is actually present. This may be useful for comparing performance based on the amount of memory.

12.19 SUBPRIMITIVES TO SHUT DOWN THE LISP ENVIRONMENT

sys:%halt
Stops the machine.

sys:%crash (code object paws-up-p)
Halts the machine after writing a crash record that includes code and object (each stored as one word). If paws-up-p is non-nil, the monitor screen will invert its video characteristic.

12.20 DISTIGUISHING PROCESSOR TYPES

The Explorer system software may (but is not guaranteed to) run on CADR and Lambda processors if it is converted to the correct FEF format for those processors. However, obscure or internal I/O code sometimes needs to behave differently at run-time depending on the type of processor. This is possible through the use of these macros.

sys:processor-type-code Variable
This variable is 1 on a CADR processor or equivalent, 2 on a Lambda, and 3 for the Explorer.

if-in-cadr Body... Macro
Executes body only when executing on a CADR.

if-in-lambda body... Macro
Executes body only when executing on a Lambda.

if-in-explorer body... Macro
Executes body only when executing on an Explorer.

if-in-cadr-else-lambda if-cadr-form else-body Macro
Executes if-cadr-form when executing on a CADR, executes else-body when executing on a Lambda or Explorer.

if-in-lambda-else-cadr if-lambda-form else-body... Macro
Executes if-cadr-form when executing on a Lambda; executes else-body when executing on a CADR or Explorer.

select-processor Clauses

Macro

Each clause consists of :cadr, :lambda, or :explorer keywords followed by forms to execute when running on that kind of processor. Example:

```
(format t "~&Processor is a ~A. ~%"
  (select-processor
    (:cadr "CADR")
    (:lambda "Lambda")
    (:explorer "Explorer")))
```

12.21 MICROCODE VARIABLES

The following variables' values actually reside in the scratchpad memory of the processor. They are put there by dtp-one-q-forward invisible pointers. The values of these variables are used by the microcode. Many of these variables are highly internal, and their full meanings are not necessarily documented here.

%microcode-version-number

Variable

This is the version number of the currently-loaded microcode, obtained from the version number of the microcode source file.

sys:%number-of-micro-entries

Variable

Size of micro-code-entry-area and related areas. g(default-cons-area is documented on page 297).

sys: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 value of sys:extra-pdl-area, which enables the "temporary storage" feature for numbers, saving garbage collection overhead.

sys:%current-stack-group-state

Variable

The sg-state of the currently-running stack group.

sys:%current-stack-group-calling-args-pointer

Variable

The argument list of the currently-running stack group.

sys:%trap-micro-pc

Variable

The microcode address of the most recent error trap.

sys:%initial-fef

Variable

The function that is called when the machine starts up. Normally this is the definition of si:lisp-top-level.

sys:%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.

sys:%error-handler-stack-group Constant
The stack group that receives control when a microcode-detected error occurs. This stack group cleans up, signals the appropriate condition, or assigns a stack group to run the debugger on the erring stack group.

sys:%scheduler-stack-group Constant
The stack group that receives control when a sequence break occurs.

sys:%inhibit-read-only Variable
If non-nil, you can write into read-only areas. This is used by fasload.

inhibit-scavenging-flag Variable
If non-nil, the scavenger is turned off. The scavenger is the the quasi-asynchronous portion of the garbage collector, which normally runs during consing operations.

inhibit-scavenging-ws-enable Variable
If this is nil, scavenging can compete for all of the physical memory of the machine. Otherwise, it should be a fixnum, which specifies how much physical memory the scavenger can use: page numbers as high as this number or higher are not available to it.

sys:%region-cons-alarm Variable
Increments whenever a new region is allocated.

sys:%page-cons-alarm Variable
Increments whenever a new page is allocated.

sys:%gc-flip-ready Variable
t while the scavenger is running. nil when there are no pointers to oldspace.

sys:%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 %pointer of an object may have changed.

sys:%disk-run-light Constant
A fixnum, 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 garbage collector.

sys:%loaded-band Variable
A fixnum, the high 24 bits of the name of the disk partition from which virtual memory was booted. Used to create the greeting message.

sys:%disk-blocks-per-track Variable

sys:%disk-blocks-per-cylinder Variable
Configuration of the disk being used for paging.

sys:%qlaryh Variable
This is the last array to be called as a function, remembered for the sake of the function store.

sys:%qlaryl Variable
This is the index used the last time an array was called as a function, remembered for the sake of the function store.

sys:%currently-prepared-sheet Variable
Used for communication between the window system and the microcoded graphics primitives.

sys:%meter-global-enable Variable
t if the metering system is turned on for all stack-groups.

sys:%meter-buffer-pointer Variable
A temporary buffer used by the metering system

sys:%meter-disk-address Variable
Where the metering system writes its next block of results on the disk.

sys:%meter-disk-count Variable
The number of disk blocks remaining for recording of metering information.

sys:lexical-environment Variable
This is the static chain used in the implementation of lexical scoping of variable bindings in compiled code.

sys:amem-evcp-vector Variable
No longer used.

sys:a-memory-location-names Constant
A list of all of the above symbols (and any others added after this documentation was written).

12.22 MICROCODE METERS

Microcode meters are locations in the scratchpad memory which contain numbers. Most of them are used to count events of various sorts. They are accessible only through the functions read-meter and write-meter. They have nothing to do with the Lisp metering tools.

read-meter (name)
Returns the contents of the microcode meter named name, which can be a fixnum or a bignum. name must be one of the symbols listed below.

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.

The microcode meters are as follows:

sys:%count-chaos-transmit-aborts Meter
The number of times transmission on the Chaosnet was aborted, either by a collision or because the receiver was busy.

sys:%count-cons-work Meter

sys:%count-scavenger-work Meter
Internal state of the garbage collection algorithm.

sys:%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.

sys:%count-first-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.

sys:%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.

sys:%count-meta-bits-map-reloads Meter
The number of times the virtual address map was reloaded to contain only "meta bits", not an actual physical address.

sys:%count-pdl-buffer-read-faults Meter
The number of read references to the pdl buffer that were virtual memory references that trapped.

sys:%count-pdl-buffer-write-faults Meter
The number of write references to the pdl buffer that were virtual memory references that trapped.

sys:%count-pdl-buffer-memory-faults Meter
The number of virtual memory references that trapped in case they should have gone to the pdl buffer, but turned out to be real memory references after all (and therefore were needlessly slowed down).

sys:%count-disk-page-reads Meter
The number of pages read from the disk.

sys:%count-disk-page-writes Meter
The number of pages written to the disk.

sys:%count-fresh-pages Meter
The number of fresh (newly-consed) pages created in core, which would have otherwise been read from the disk.

sys:%count-disk-page-read-operations Meter
The number of paging read operations: this can be smaller than the number of disk pages read when more than one page at a time is read.

sys:%count-disk-page-write-operations Meter
The number of paging write operations: this can be smaller than the number of disk pages written when more than one page at a time is written.

sys:%count-disk-prepages-used Meter
The number of times a page was used after being read in before it was needed.

sys:%count-disk-prepages-not-used Meter
The number of times a page was read in before it was needed, but got evicted before it was ever used.

sys:%count-disk-page-write-waits Meter
The number of times the machine waited for a page to finish being written out in order to evict the page.

sys:%count-disk-write-busys Meter
The number of times the machine waited for a page to finish being written out in order to do something else with the disk.

sys:%disk-wait-time Meter
The time spent waiting for the disk, in microseconds. This can be used to distinguish paging time from running time when measuring and optimizing the performance of programs.

sys:%count-disk-errors Meter
The number of recoverable disk errors.

sys:%count-disk-recalibrates Meter
The number of disk seek mechanism was recalibrated, usually as part of error recovery.

sys:%count-disk-ecc-corrected-errors Meter
The number of disk errors that were corrected through the error correcting code.

sys:%count-disk-read-compare-differences Meter
The number of times a read compare was done, no disk error occurred, but the data on disk did not match the data in memory.

sys:%count-disk-read-compare-rereads Meter
The number of times a disk read was done over because after the read a read compare was done and did not succeed (either it got an error or the data on disk did not match the data in memory).

sys:%count-disk-read-compare-rewrites Meter
The number of times a disk write was done over because after the write a read compare was done and did not succeed (either it got an error or the data on disk did not match the data in memory).

sys:%disk-error-log-pointer Meter
Address of the next entry to be written in the disk error log. The function si:print-disk-error-log prints this log.

sys:%count-aged-pages Meter
This number of times the page ager set an age trap on a page, to determine whether it was being referenced.

sys:%count-age-flushed-pages Meter
The number of times the page ager saw that a page still had an age trap and hence made it "flushable", a candidate for eviction from main memory.

sys:%aging-depth Meter
A number from 0 to 3 that controls how long a page must remain unreferenced before it becomes a candidate for eviction from main memory.

sys:%count-findcore-steps Meter
The number of pages inspected by the page replacement algorithm.

sys:%count-findcore-emergencies Meter
The number of times no evictable page was found and extra aging had to be done.

sys:%a-memory-counter-block-names Meter
A list of all the above symbols (and any others added after this documentation was written).

SECTION 13

Low-Level Lisp Data Structures

13.1 INTRODUCTION

This section describes manipulation of some internal data structures that most Lisp programmers don't need to know about, but which may be of interest to systems programmers.

The definition of a function spec actually has two parts: the basic definition, and encapsulations. The basic definition is what is created by functions like `defun`, and encapsulations are additions made by `trace` or `advise` to the basic definitions. The purpose of making the encapsulation a separate object is to keep track of what was made by `defun` and what was made by `trace`. If `defun` is done a second time, it replaces the old basic definition with a new one while leaving the encapsulations alone.

Only advanced users should ever need to use encapsulations directly via the primitives explained in this section. The most common things to do with encapsulations are provided as higher-level, easier-to-use features: `trace`, `breakon` and `advise`.

The actual definition of the function spec is the outermost encapsulation; this contains the next encapsulation, and so on. The innermost encapsulation contains the basic definition. The way this containing is done is as follows: An encapsulation is actually a function whose debugging info alist contains an element of the form

```
(si:encapsulated-definition uninterned-symbol encapsulation-type)
```

The presence of such an element in the debugging info alist is how you recognize a function to be an encapsulation. An encapsulation is usually an interpreted function (a list starting with `named-lambda`) but it can be a compiled function also, if the application which created it wants to compile it.

uninterned-symbol's function definition is the thing that the encapsulation contains, usually the basic definition of the function spec. Or it can be another encapsulation, which has in it another debugging info item containing another uninterned symbol. Eventually you get to a function which is not an encapsulation; it does not have the sort of debugging info item which encapsulations all have. That function is the basic definition of the function spec.

Literally speaking, the definition of the function `spec` is the outermost encapsulation, period. The basic definition is not the definition. If you are asking for the definition of the function `spec` because you want to apply it, the outermost encapsulation is exactly what you want. But the basic definition can be found mechanically from the definition, by following the debugging info lists. So it makes sense to think of it as a part of the definition. In regard to the function-defining special forms such as `defun`, it is convenient to think of the encapsulations as connecting between the function `spec` and its basic definition.

An encapsulation is created with the macro `si:encapsulate`.

13.2 ENCAPSULATION FUNCTIONS

Macro

`si:encapsulate function-spec outer-function`
`type body-form extra-debugging-info`)

All the subforms of this macro are evaluated. In fact, the macro could Almost be replaced with an ordinary function, except for the way `body-form` is handled.

`function-spec` evaluates to the function `spec` whose definition the new encapsulation should become. `outer-function` is another function `spec`, which should often be the same one. Its only purpose is to be used in any error messages from `si:encapsulate`.

`type` evaluates to a symbol which identifies the purpose of the encapsulation and says what the application is. For example, that could be `advise` or `trace`. The list of possible types is defined by the system because encapsulations are supposed to be kept in an order according to their type. The `type` argument should have an `si:encapsulation-grind-function` property which tells `grindef` what to do with an encapsulation of this type.

`body-form` evaluates to the body of the encapsulation- definition, the code to be executed when it is called. Backquote is typically used for this expression (see Section 16.3.1.2); `si:encapsulate` is a macro because, while `body` is being evaluated, the variable `si:encapsulated-function` is bound to a list of the form `(function uninterned-symbol)`, referring to the uninterned symbol used to hold the prior definition of `function-spec`. If `si:encapsulate` were a function, `body-form` would just get evaluated normally by the evaluator before `si:encapsulate` ever got invoked, and so there would be no opportunity to bind `si:encapsulated-function`. The form `body-form` should contain `(apply ,si:encapsulated-function arglist)` somewhere if the encapsulation is to live up to its name and truly serve to encapsulate the original definition. (The variable `arglist` is

bound by some of the code which the si:encapsulate macro produces automatically. When the body of the encapsulation is run the value of arglist will be the list of the arguments which the encapsulation received.)

extra-debugging-info evaluates to a list of extra items to put into the debugging information alist of the encapsulation function (besides the one starting with si:encapsulated-definition, which every encapsulation must have). Some applications find this useful for recording information about the encapsulation for their own later use.

If compile-encapsulations-flag is non-nil, the encapsulation is compiled before it is installed. The encapsulations on a particular function spec can be compiled by calling compile-encapsulations. See compile-encapsulations. Compiled encapsulations can still be unencapsulated since the information needed to do so is stored in the debugging information alist, which is preserved by compilation. However, applications which wish to modify the code of the encapsulations they previously created must check for encapsulations that have been compiled and uncompile them. This can be done by finding the sys:interpreted-definition entry in the debugging information alist, which is present in all compiled functions except those made by file-to-file compilation.

When a special function is encapsulated, the encapsulation is itself a special function with the same argument quoting pattern. Therefore, when the outermost encapsulation is started, each argument has been evaluated or not as appropriate. Because each encapsulation calls the prior definition with apply, no further evaluation takes place, and the basic definition of the special form also finds the arguments evaluated or not as appropriate. The basic definition may call eval on some of these arguments or parts of them; the encapsulations should not.

Macros cannot be encapsulated, but their expander functions can be; if the definition of function-spec is a macro, then si:encapsulate automatically encapsulates the expander function instead. In this case, the definition of the uninterned symbol is the original macro definition, not just the original expander function. It would not work for the encapsulation to apply the macro definition. So during the evaluation of body-form, si:encapsulated-function is bound to the form (cdr (function uninterned-symbol)), which extracts the expander function from the prior definition of the macro.

Because only the expander function is actually encapsulated, the encapsulation does not see the evaluation or execution of the expansion itself. The value returned by the encapsulation is the expansion of the macro call, not the value computed by the expansion.

It is possible for one function to have multiple encapsulations, created by different subsystems. In this case, the order of encapsulations is independent of the order in which they were made. It depends instead on their types. All possible encapsulation types have a total order and a new encapsulation is put in the right place among the existing encapsulations according to its type and their types.

Variable

si:encapsulation-standard-order

The value of this variable is a list of the allowed encapsulation types, in the order in which the encapsulations are supposed to be kept (innermost encapsulations first). If you want to add new kinds of encapsulations, you should add another symbol to this list. Initially its value is

```
(advise breakon trace si:rename-within)
```

advise encapsulations are used to hold advice. breakon encapsulations are used for implementing breakon. trace encapsulations are used for implementing tracing. si:rename-within encapsulations are used to record the fact that function specs of the form (:within within-function altered-function) have been defined. The encapsulation goes on within-function.

Every symbol used as an encapsulation type must be on the list si:encapsulation-standard-order. In addition, it should have an si:encapsulation-grind-function property whose value is a function that grindef will call to process encapsulations of that type. This function need not take care of printing the encapsulated function because grindef will do that itself. But it should print any information about the encapsulation itself which the user ought to see. Refer to the code for the grind function for advise to see how to write one.

To find the right place in the ordering to insert a new encapsulation, it is necessary to parse existing ones. This is done with the function si:unencapsulate-function-spec.

Function

si:unencapsulate-function-spec function-spec
&optional encapsulation-types

This takes one function spec and returns another. If the original function spec is undefined, or has only a basic definition (that is, its definition is not an encapsulation), then the original function spec is returned unchanged.

If the definition of function-spec is an encapsulation, then its debugging information is examined to find the uninterned symbol

that holds the encapsulated definition and the encapsulation type. If the encapsulation is of a type that is to be skipped over, the uninterned symbol replaces the original function spec and the process repeats.

The value returned is the uninterned symbol from inside the last encapsulation skipped. This uninterned symbol is the first one that does not have a definition that is an encapsulation that should be skipped. Or the value can be function-spec if function-specs definition is not an encapsulation that should be skipped.

The types of encapsulations to be skipped over are specified by encapsulation-types. This can be a list of the types to be skipped, or nil meaning skip all encapsulations (this is the default). Skipping all encapsulations means returning the uninterned symbol that holds the basic definition of function-spec. That is, the definition of the function spec returned is the basic definition of the function spec supplied. Thus,

```
(fdefinition (si:unencapsulate-function-spec foo))
```

returns the basic definition of foo, and

```
(fdefine (si:unencapsulate-function-spec foo) bar)
```

sets the basic definition (just like using fdefine with carefully supplied as t).

encapsulation-types can also be a symbol, which should be an encapsulation type; then we skip all types that are supposed to come outside of the specified type. For example, if encapsulation-types is trace, then we skip all types of encapsulations that come outside of trace encapsulations, but we do not skip trace encapsulations themselves. The result is a function spec that is where the trace encapsulation ought to be, if there is one. Either the definition of this function spec is a trace encapsulation, or there is no trace encapsulation anywhere in the definition of function-spec, and this function spec is where it would belong if there were one. For example,

```
(let ((tem (si:unencapsulate-function-spec spec trace))) (and (eq
tem (si:unencapsulate-function-spec tem (trace))) (si:encapsulate
tem spec trace `(... body...))))
```

finds the place where a trace encapsulation ought to go and makes one unless there is already one there.

```
(let ((tem (si:unencapsulate-function-spec spec trace))) (fdefine
tem (fdefinition (si:unencapsulate-function-spec tem (trace)))))
```

eliminates any trace encapsulation by replacing it by whatever it encapsulates. (If there is no trace encapsulation, this code changes nothing.)

These examples show how a subsystem can insert its own type of encapsulation in the proper sequence without knowing the names of any other types of encapsulations. Only the variable si:encapsulation-standard-order, which is used by si:unencapsulate-function-spec, knows the order.

One special kind of encapsulation is the type si:rename-within. This encapsulation goes around a definition in which renamings of functions have been done.

How is this used?

If you define, advise, or trace (:within foo bar), then bar gets renamed to altered-bar-within-foo wherever it is called from foo, and foo gets a si:rename-within encapsulation to record the fact. The purpose of the encapsulation is to enable various parts of the system to do what seems natural to the user. For example, grindef notices the encapsulation, and so knows to print bar instead of altered-bar-within-foo when grinding the definition of foo.

Also, if you redefine foo, or trace or advise it, the new definition gets the same renaming done (bar replaced by altered-bar-within-foo). To make this work, everyone who alters part of a function definition should pass the new part of the definition through the function si:rename-within-new-definition-maybe.

Function

si:rename-within-new-definition-maybe function-spec
new-structure

Given new-structure, which is going to become a part of the definition of function-spec, perform on it the replacements described by the si:rename-within encapsulation in the definition of function-spec, if there is one. The altered (copied) list structure is returned.

It is not necessary to call this function yourself when you replace the basic definition because fdefine with carefully supplied as it does it for you. si:encapsulate does this to the body of the new encapsulation. So you only need to call si:rename-within-new-definition-maybe yourself if you are replacing part of the definition.

For proper results, function-spec must be the outer-level function spec. That is, the value returned by si:unencapsulate-function-spec is not the right thing to use. It will have had one or more encapsulations stripped off, including the si:rename-within encapsulation if any, and so no renamings will be done.

13.3 DEFSTRUCT DESCRIPTION

This subsection discusses the internal structures used by defstruct that might be useful to programs that want to interface to defstruct nicely. For example, if you want to write a program that examines structures and displays them the way describe and the Inspector do, your program should work by examining these structures. The information in this section is also necessary for anyone who is thinking of defining his own structure types.

Whenever the user defines a new structure using defstruct, defstruct creates an instance of the si:defstruct-description structure. This can be found as the si:defstruct-description property of the name of the structure; it contains such useful information as the number of slots in the structure, the defstruct options specified, and so on.

This is a simplified version of the way the si:defstruct-description structure is defined. (The actual definition is in the system-internals package.)

```
(defstruct (defstruct-description
            (:default-pointer description)
            (:conc-name defstruct-description-))
  name
  size
  property-alist
  slot-alist
  documentation)
```

The name slot contains the symbol supplied by the user to be the name of his structure, such as spaceship or phone-book-entry. The size slot contains the total number of slots in an instance of this kind of structure. This is not the same number as that obtained from the :size-symbol or :size-macro options to defstruct. A named structure, for example, usually uses up an extra location to store the name of the structure, so the :size-macro option will get a number one larger than that stored in the defstruct description. The property-alist slot contains an alist with pairs of the form (property-name . property) containing properties placed there by the :property option to defstruct or by property names used as options to defstruct (see the :property option). The slot-alist slot contains an alist of pairs of the form (slot-name . slot-description). A slot-description is an instance of the defstruct-slot-description structure. The defstruct-slot-description structure is defined something like this, also in the si package:

```
(defstruct (defstruct-slot-description
  (:default-pointer slot-description)
  (:conc-name defstruct-slot-description-))
  number byte
  init-code type
  property-alist
  ref-macro-name
  documentation)
```

(This is also a simplified version of the real definition.) The number slot contains the number of the location of this slot in an instance of the structure. Locations are numbered, starting with 0, and continuing up to a number one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of the structure. The byte slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, then the byte slot contains nil. The init-code slot contains the initialization code supplied for this slot by the user in his defstruct form. If there is no initialization code for this slot then the init-code slot contains the symbol si:%%defstruct-empty%. The ref-macro-name slot contains the symbol that is defined as a macro or a subst that expands into a reference to this slot (that is, the name of the accessor function).

13.4 MORE EXTENSIONS TO DEFSTRUCT

The macro defstruct-define-type can be used to teach defstruct about new types that it can use to implement structures.

Macro

defstruct-define-type

This macro is used for teaching defstruct about new types; it is described in the rest of this chapter.

Example

Let us start by examining a sample call to defstruct-define-type. This is how the :list type of structure might have been defined:

```
(defstruct-define-type :list
  (:cons (initialization-list description
    keyword-options)
    :list '(list . , initialization-list))
  :ref (slot-number description argument)
    '(nth , slot-number , argument)))
```

This is the simplest possible form of defstruct-define-type. It provides defstruct with two Lisp forms: one for creating forms to construct instances of the structure, and one for creating forms to become the bodies of accessors for slots of the structure.

The keyword :cons is followed by a list of three variables that will be bound while the constructor-creating form is evaluated. The first, initialization-list, will be bound to a list of the initialization forms for the slots of the structure. The second, description, will be bound to the defstruct-description structure for the structure. The third variable and the :list keyword will be explained later.

The keyword :ref is followed by a list of three variables that will be bound while the accessor-creating form is evaluated. The first, slot-number, will be bound to the number of the slot that the new accessor should reference. The second, description, will be bound to the defstruct-description structure for the structure. The third, argument, will be bound to the form that was provided as the argument to the accessor.

The syntax of defstruct-define-type is:

(defstruct-define-type type option-1 option-2 ...)

where each option is either the symbolic name of an option or a list of the form (option-name . rest). Different options interpret rest in different ways. The symbol type is given an si:defstruct-type-description property of a structure that describes the type completely.

Options

This section is a catalog of all the options currently known about by defstruct-define-type.

:cons

With the :cons option to defstruct-define-type you supply defstruct with the code to cons up a form that will construct an instance of a structure of this type.

The :cons option has the syntax:

(:cons (inits description keywords) kind body)

body is some code that should construct and return a piece of code that will construct, initialize, and return an instance of a structure of this type.

The symbol inits will be bound to the information that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol kind. There are currently two kinds of initialization. There is the :list kind, where inits is bound to a list of initializations, in the correct order, with nils in uninitialized slots. And there is the :alist kind, where inits is bound to an alist with pairs of the form (slot-number . init-code).

The symbol description will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure. This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol keywords will be bound to an alist with pairs of the form (keyword . value), where each keyword was a keyword supplied to the constructor that wasn't the name of a slot, and value was the Lisp object that followed the keyword. This is how you can make your own special keywords, like the existing :make-array and :times keywords. See the section on using the constructor. You specify the list of acceptable keywords with the :cons-keywords option. It is an error not to supply the :cons option to defstruct-define-type.

:ref

With the :ref option to defstruct-define-type you supply defstruct with the code to cons up a form that will reference an instance of a structure of this type.

The :ref option has the syntax:

(:ref (number description arg-1 ... arg-n) body)

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol number will be bound to the location of the slot that is to be referenced. This is the same number that is found in the number slot of the defstruct-slot- description structure.

The symbol description will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure.

The symbols arg-i are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The last argument is the one that will be defaulted

by the :default-pointer option. defstruct will check that the user has supplied exactly n arguments to the accessor function before calling the reference consing code.

It is an error not to supply the :ref option to defstruct-define-type.

:overhead

The :overhead option to defstruct-define-type is how the user declares to defstruct that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of :overhead is (overhead n), where n is a fixnum that says how many locations of overhead this type needs.

This number is used only by the :size-macro and :size-symbol options to defstruct.

:named

The :named option to defstruct-define-type controls the use of the :named option to defstruct. With no argument, the :named option means that this type is an acceptable "named structure". With an argument, as in (:named type-name), the symbol type-name should be the name of some other structure type that defstruct should use if someone asks for the named version of this type. (For example, in the definition of the :list type the :named option is used like this: (:named :named- list).)

:cons-keywords

The :cons-keywords option to defstruct-define-type allows the user to define additional constructor keywords for this type of structure. (The :make-array constructor keyword for structures of type :array is an example.) The syntax is: (:cons-keywords keyword-1 ... keyword-n) where each keyword is a symbol that the constructor conser expects to find in the keywords alist (explained above).

:keywords

:keywords is an old name for the :cons-keywords option.
endkey

:defstruct-keywords

The :defstruct-keywords option to defstruct-define-type allows the user to define additional defstruct options allowed for this type of structure. (The :times option for structures of type :grouped-array is an example.) The syntax is: (:defstruct-keywords keyword-1 ... keyword-n) where each keyword is a keyword that defstruct will recognize as an option. defstruct puts such options, with

their values, in the property-alist slot of the defstruct-description structure (defined above)

:predicate

With the :predicate option to defstruct-define-type, defstruct is told how to produce predicates for a particular type (for the :predicate option to defstruct). Its syntax is:

```
(predicate (description name) body...)
```

The variable description will be bound to the defstruct-description structure maintained for the structure we are to generate a predicate for. The variable name is bound to the symbol that is to be defined as a predicate. body is a piece of code to evaluate to return the defining form for the predicate. A typical use of this option might look like:

```
(predicate (description name)
  ' (defun name (x)
      (and (frobbozp x)
           (eq (frobbozref x) 0)
           ', (defstruct-description-name))))
```

:copier

defstruct-define-type defstruct knows how to generate a copier function using the constructor and reference code that must be provided with any new defstruct type. Nevertheless it is sometimes desirable to specify a specific method of copying a particular defstruct type. The :copier option to defstruct-define-type allow this to be done:

```
(copier (description name) body)
```

As with the :predicate option, description is bound to an instance of the defstruct-description structure, name is bound to the symbol to be defined, and body is some code to evaluate to get the defining form. For example:

```
(copier (description name)
```

:defstruct

The :defstruct option to defstruct-define-type allows the user to run some code and return some forms as part of the expansion of the defstruct macro.

The :defstruct option has the syntax:

```
(:defstruct (description) body)
```

body is a piece of code that will be run whenever defstruct is expanding a defstruct form that defines a structure of this type. The symbol description will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure.

The value returned by the body should be a list of forms to be included with those that the {b defstruct} expands into. Thus, if you only want to run some code at defstruct expand time, and you don't want to actually output any additional code, then you should be careful to return nil from the code in this option.

defstruct will cause the body forms to be evaluated as early as possible in the parsing of a structure definition, and for the returned forms to be evaluated as late as possible in the macro-expansion of the defstruct forms. This is so that body can rehack arguments, signal errors, and the like before many of defstructs internal forms are executed, while enabling it to return code which will modify or extend the default forms produced by a vanilla defstruct.

SECTION 14

Error Handling

14.1 INTRODUCTION

This chapter explains the signalling of errors. It explains how the microcode signals an error condition to a program or to the user.

This chapter is organized from the abstract to the concrete. First, the error conditions are described. Then progressively lower levels of the implementation are discussed.

14.2 MICROCODE ERROR CONDITIONS

There are a number of condition 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-ERROR: 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 FLONUM.

FLOATING-EXPONENT-UNDERFLOW-ERROR: result is too small in magnitude to be represented as a FLONUM.

FUNCTION-ENTRY-ERROR: a problem was encountered in entering a function, like too many or too few arguments, or an argument of a bad data type.

INTERNAL-MEMORY-LOCATION-OOB-ERROR: 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-SEEN-ERROR: a THROW was done to a tag for which there is no pending *CATCH.

UNBOUND-VARIABLE: an access to a symbols value cell or to a closure variable found it unbound (DTP-NULL). Built on CELL-CONTENTS-ERROR.

UNDEFINED-FUNCTION: an access to a symbols function cell or to a method of a select-method found it unbound (DTP-NULL). Built on CELL-CONTENTS-ERROR.

UNIMPLEMENTED-HARDWARE-ERROR: 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.

14.2.1 Microcode Error Table.

This paragraph is a description of microcode error-table entries. The microcode error table is read from SYS:UBIN;CONTROL.TBL#nnn (where nnn is 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;EHF 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 ETE's are lists whose CARs are symbols. The symbol is defined by a DEF-UCODE-ERROR form in SYS:EH;EHF. 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. Types not used in the microcode are marked **. 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-T, M-R, M-Q, M-I, M-J, M-S, M-K, M-1, M-2, A-QCSTKG, 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 and M-2 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.

14.2.1.1 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-label 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.

** STACK-WORDS-PUSHED arg1. Saves arg1 as info about the number of words pushed, presumably how many. These entries are collected during eh:initialize, but nothing is done with them.

14.2.1.2 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 and M-2 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 and M-2 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 QARYR if this is an array called as a function.

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 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 wrong data type. arg is name. As it happens,

all the names either start with a vowel or do if pronounced as letters. Not continuable." The name pronunciation comment is no longer true, so the trap message might look funny. Of course, this sort of thing isn't supposed to happen.

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."

*FIXNUM-OVERFLOW number-location push-new-value-flag.

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.

FUNCTION-ENTRY.

"Function ~S called with only ~D argument ~10*~P."

"Function ~S called with too many arguments (~D)."

"Function ~S called with an argument of bad data type."

*IALLB-TOO-SMALL number-location.

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: ~D."

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."

*MVR-BAD-NUMBER bad-number-location.

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.

SELECT-METHOD-BAD-SUBROUTINE-CALL select-method-location. "Bad subroutine call found inside select-method."

SELECT-METHOD-GARBAGE-IN-SELECT-METHOD-LIST garbage location.
"The weird object ~S was found in a select-method alist."

SELECTED-METHOD-NOT-FOUND select-method-location message-
location. An unclaimed-message error for a select-method.

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-EXIT-TRAP.

*THROW-TAG-NOT-SEEN.

THROW-TRAP "THROW-TRAP is used for both exit trap and tag not
seen, starting in UCADR 260. If M-E contains NIL, the tag was
not seen." Trap here means tag not seen if M-E is NIL, means
throwing thru trap-on-exit frame otherwise. The error handler
knows which M-locations contain the information here.

TOO-MANY-PAGE-DEVICES. "There is no room for another logical
page device."

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~]."

TV-ERASE-OFF-SCREEN. "An attempt was made to do graphics past the end of the screen."

UNIMPLEMENTED-HARDWARE hardware-type. "Unimplemented hardware type ~S." Hardware-type is UNIBUS or XBUS.

USER-NUBUS-ERROR highy-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."

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-AEGS-TO-SELECT-METHOD select-method-location. "~S was applied to no arguments."

14.3 ILLEGAL OPERATION

When the microcode detects an irrecoverable or "can't happen" error, it will crash the system. This is known as ILLOP after the microcode routine that performs this function.

ILLOP causes the machine to drop dead. 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 in the NVRAM 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 kernel is well and that A-Zero, M-Zero, A-Ones, and M-Ones are set up, 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.

SECTION 15

Crash Handling

15.1 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 si:setup-nvram function. Only then can valid crash records be recorded. Since NVRAM is non-volatile memory, the si: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.

15.1.1 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 15-1. All of the allocation registers are 16-bit byte offsets into the NVRAM. Offsets are expressed in hexadecimal. Also since NVRAM is 8-bit memory stored one-byte-per-NuBus-word, only multiple-of-four addresses are used (eg, 0, 4, 8, C, 10, ...). 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, F4B2 (hex) would be stored in NVRAM-CRASH-BUFF-POINTER with B2 in NVRAM-BASE+90(16) and F4 in NVRAM-BASE+94(16).

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 15-1 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

15.1.2 Crash Record Contents.

The crash record format is shown in Table 15-2. 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:UBIN;QDEV.

Table 15-2 Crash Record Format

NVRAM-CRASH-BUFF-POINTER plus (hex)	Contents
<hr/>	
	General information
<hr/>	
0	Progress field. Indicates how far into the boot process the system progressed before halting.
<hr/>	
	Load information
<hr/>	
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)
<hr/>	
	Date and time information
<hr/>	
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
<hr/>	
	Flags field
<hr/>	
70	Report Flags
<hr/>	
	Shutdown information
<hr/>	
74	Halt Location (2)
7C	Halt Kind
<hr/>	
	Saved Registers
<hr/>	
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	currently unused (8)

The progress field indicates how far into the boot process the system progressed before crashing. Currently supported values for this field are: Initial-Value (crash record not yet allocated), Allocated (crash record allocation by the microcode was successful), Starting-Lisp (the microcode transferred control to Lisp), Warm-Initializations (the warm-initializations-list was being run), and Time-Initialized (the time base had been initialized and current time written to the crash record). If the reported value is Time-Initialized, the initial Lisp environment was probably reached before the halt occurred.

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 periodically from Lisp 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).

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. reserve block 9
- * 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 `si:%crash`) is stored in the halt address field. The second argument to `si:%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 `si:shutdown` or `si: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.

15.1.3 Crash Analyzer.

The crash analyzer consists of a number of Lisp routines that report the contents of crash records to the user in a readable format. The two user-callable functions to invoke the crash analyzer are `si:report-last-shutdown` and `si:report-all-shutdowns`.

`si:report-last-shutdown (&key (stream terminal-io) (pathname nil) (abnormal-only nil))`

Reports the results of analyzing the crash record from the previous boot. If abnormal-only is t, the crash record is only reported if it represents a crash (versus a normal Lisp shutdown or boot). abnormal-only defaults to nil. If pathname is non-nil, the crash record is written to the indicated file instead. pathname must be parsable into a pathname, and is opened in the append mode. If pathname is nil, the crash record analysis is written to the stream indicated by the stream keyword. stream defaults to `terminal-io`.

`si:report-all-shutdowns (&key (stream terminal-io) (pathname nil) (abnormal-only nil) (unlogged-only nil))`

Reports the results of analyzing all currently recorded crash records. If abnormal-only is t, the crash record is only reported if it represents a crash (versus a normal Lisp shutdown or boot). abnormal-only defaults to nil. Usually the analysis is written to the stream indicated by the stream keyword. If

pathname is non-nil, however, the crash record is written to the indicated file instead. pathname must be parsable into a pathname, and is opened in the append mode. If pathname is non-nil and unlogged-only is t, only records that have not previously been logged will be written to the log file. Crash records are marked internally as logged by being written to a log file either by this function or by si:report-last-shutdown.

The crash analysis format differs for each halt kind. For system boots only header information, consisting of the progress fields load information, time information, and halt kind, are displayed. For Lisp halts, the header information is displayed followed by the Lisp crash code, a text description of the Lisp crash code, (only "Normal shutdown by SHUTDOWN" is currently supported), and the saved register values. Microcode halts are reported in much the same way, except that the halt address is given along with a text string describing the microcode halt. This text string is looked up from the crash analysis database on the basis of the microcode halt address.

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;CONTROL.CRASH#nnn where nnn is the microcode revision number. The crash analyzer loads the appropriate version of this file into memory when a microcode crash is being reported. See SYS:UBIN;CONTROL.CRASH#nnn for a complete list of all currently implemented crash descriptor texts. Note that there are 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.

SECTION 16

Compiler Notes

16.1 CROSS-COMPILATIONS

The Explorer Lisp compiler also supports generation of object code for Cadr and Lambda Lisp Machines. Normally the compiler generates code for the machine it is running on, but when the compiled code is being written to a file, it can be told that the object file is to be loaded on a different processor. This is called "cross-compilation". Object files with type XFASL are used for object code to be loaded on an Explorer and type QFASL for a Cadr or Lambda.

Cross-compilation can be done in any of three ways:

1. Using the :TARGET keyword option of the COMPILE-FILE function. For example,

```
(COMPILE-FILE "do;dah" :TARGET "LAMBDA")
```

will create a QFASL file that can be loaded on either a Lambda or Cadr. The value for the :TARGET argument should be a symbol or string which is STRING-EQUAL to one of the names CADR, LAMBDA, or EXPLORER.

2. Explicit specification of object file type. If the functions COMPILE-FILE or QC-FILE are called with a full pathname specified for the output file, then the compilation will be done for the corresponding machine. For example,

```
(COMPILE-FILE "do;dah" :OUTPUT-FILE "do;dah.QFASL")
```

will have the same effect as the previous example.

3. Using MAKE-SYSTEM. The MAKE-SYSTEM function provides for passing optional arguments to the COMPILE-FILE function. For example,

```
(MAKE-SYSTEM "name" (:COMPILE :TARGET "LAMBDA"))
```

will update the appropriate QFASL files.

During a cross-compilation, the `*FEATURES*` list will be bound around the reader so that the reader macros `#+` and `#-` can be used with names `CADR`, `LAMBDA`, and `EXPLORER` to make code conditional on the machine type.

In order to support cross-compilation, there is also a cross-loader facility which reads a LISP file to define the values and properties of symbols as they will be on another machine. For example,

```
(COMPILER:LOAD-FOR-TARGET "SYS:COLD-BAND;GCOM.LISP" "CADR")
```

will load the virtual machine declarations for `Cadrs` and `Lambdas`, while

```
(COMPILER:LOAD-FOR-TARGET "SYS:COLD-BAND;GCOM.LISP" "EXPLORER")
```

will load the Explorer version. Within the file, the reader macros `#+EXPLORER` and `#-EXPLORER` are used to mark the differences. This is not a completely general facility; it does not accept compiled object files, it ignores any function or macro definitions, and it only intercepts certain forms. Since it is known to be needed for the files `COLD-BAND;GCOM.LISP` and `COLD-BAND;DEFMIC.LISP`, it will handle all of the forms currently used in those files. The forms which are recognized for evaluation in the target environment are: `DEFCONST`, `DEFCONSTANT`, `COMPILER:DEFMIC`, `DEFPROP`, `SI:DEFALTERNATE`, `SI:DEFENUM`, `SI:DEFSYSCONST`, `SI:DEFSYSVAR`, `DEFVAR`, `GET`, `PUTPROP`, `SET`, `SETQ`, `SYMBOL-VALUE`, `SYMEVAL`, and any other macros that expand into these.

Cross-loading can also be done as part of a `MAKE-SYSTEM` by using the transformation `:CROSS-LOAD` in the `DEFSYSTEM`, similarly to `:READFILE`. This will call `COMPILER:LOAD-FOR-TARGET` to load the file(s) for the non-host environments only, since it is expected that the compiled object file will have already been loaded for the host environment.

Note well that if you intend to use cross-compilation to support software for two processors, then the use of the reader macros `#+` and `#-` to select processor type have some limitations:

- * They can not be used in macros, substs, or inline functions that are defined in one file and used in another because only the host version will be loaded.
- * Any uses that affect definitions of variables, constants, and symbol properties should be in a file which is cross-loaded before compiling any files that depend on them. These restrictions do not apply when each processor is used to compile its own object files.

Within a compilation (for example, in a macro expander function), the function `COMPILER:EVAL-FOR-TARGET` may be used the same as `SI:EVAL1` except that the evaluation will be done using whatever is known of the target environment wherever that differs from the current host environment. `COMPILER:EVAL-FOR-TARGET` is used by `COMPILER:LOAD-FOR-TARGET` where `READFILE` uses `SI:EVAL1`. This does not currently handle correctly the binding of local variables, so should not be used to execute interpreted function definitions.

16.2 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 `b 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 `defvars` 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 contains 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 `xfasl` 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)
```

16.3 XFASL FILES

All xfasl 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 xfasl 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 xfasl 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 xfasl file can be divided into whacks. The fasl-table is cleared out at the beginning of each whack.

The other groups in the xfasl file perform operations such as evaluating a list previously constructed or storing an object into a symbol's function cell or value cell.

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 occurred, 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 is a small section which provides for dynamic allocation management of the rest of the NVRAM, including the "typed block" buffer area. The last standard (i.e. fixed addressed) section is the buffer for the "typed blocks", where a variable number of 32 byte, uniquely "typed" blocks are located such that either direct information or additional buffer control structures can be added.

A.1.1.1 Test and Boot Resources.

The NVRAM contains values which the System boot procedure references in order to find preselected resources. The NuBus slot and a 24 bit logical unit number are provided for a monitor, a keyboard, 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 compatible. 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 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 updates them to reflect the current boot date and time. Then the time since boot is 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 is logged in the NVRAM and the shutdown information valid flag is set to allow the next boot operation to determine that the shutdown cause was correctly recorded.

Note: * = Not used by Explorer currently.

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 the 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 first and last records), and a pointer to the currently active crash record.

A.1.1.4 NVRAM Allocation Management.

It is desirable to be able to reserve and then dynamically allocate areas of the NVRAM for future storage of information without destroying previously entered information. The NVRAM allocation management information allows future additions to be made in a predefined, controlled manner. This is accomplished using a pointer to the beginning of the as yet unallocated area of NVRAM. Allocation of new areas of the NVRAM requires reading the pointer to determine where the new allocation shall begin, and then updating the pointer to just beyond the newly created buffer area. Since the location of any new area in NVRAM is

dependent on all previous allocations in the NVRAM, a consistent method must be provided for establishing pointers to the new area. This can be done using the Typed Blocks as described in the next section.

A.1.1.5 Typed Blocks.

The Typed Block mechanism is provided in the NVRAM to allow upward compatible additions of data and control structures to the NVRAM. Typed Blocks are 32 byte data structures with the first two bytes providing a unique Type number that identifies either that the block is unused (>FFFF), or that it is in use. The rest of the block (30 bytes) may contain whatever data or data structures the user program desires. The Typed Blocks reside in a fixed location, variable size buffer in the NVRAM. The base of the buffer is fixed at NVRAM_address >0100. The size of the buffer is set at NVRAM setup to include some integral number of 32 byte Typed Blocks, with the total Number of Blocks stored in the two bytes preceeding the buffer. Access to Typed Blocks is done by searching through the blocks until the Type of block desired is found.

For example, a program wishing to add a new buffer area in the NVRAM can do the following:

1. Use the NVRAM allocation management to locate available area
2. Allocate the area by adjusting the Unallocated pointer
3. Search through the Typed Blocks until an unused one is found
4. Mark the unused block with a preassigned Type number
5. Store a buffer access control data structure in the data area of the block

When the same or another program requires access to the new buffer it can do so by:

1. Searching through the Typed Blocks until it finds the one uniquely Typed as containing the control data structure for the buffer.
2. Using the control structure to access the buffer in NVRAM.

A.1.1.6 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	Binary
	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.	

Configuration Checksum

base +>40=	Config. Checksum LSB byte	binary
base +>44=	Config. Checksum MSB byte	binary
	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 system configuration has not changed. (note that moving a board to another slot does not change the checksum).	
base +>48-4C	reserved (>4C used 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:                               binary
              >00= overvoltage shutdown
              >01= undervoltage shutdown
              >02= overvoltage after high temperature
              >03= high temperature shutdown
              >04-FF= reserved

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=    Crash Rec. Allocation Pointer LSB           Binary
base +>94=    Crash Rec. Allocation Pointer MSB           Binary
base +>98=    Crash Rec. Allocation Size LSB byte         Binary
base +>9C=    Crash Rec. Allocation Size MSB byte         Binary
base +>A0=    Crash Record Allocation Last LSB byte       Binary
base +>A4=    Crash Record Allocation Last MSB byte       Binary
base +>A8=    Crash Record Allocation Base LSB byte       Binary
base +>AC=    Crash Record Allocation Base MSB byte       Binary

base +>B0-EC= reserved

```

NVRAM allocation management:

base +>F0=	Start of unallocated NVRAM LSB byte.	Binary
base +>F4=	Start of unallocated NVRAM MSB byte.	Binary
base +>F8=	Number of 32 byte typed blocks LSB byte.	Binary
base +>FC=	Number of 32 byte typed blocks MSB byte.	Binary

typed block list (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 identifying block type:
 0000-0007= reserved for T.I. diagnostics
 0008-000F= reserved for T.I. Lisp
 0010-0017= reserved for T.I. Unix
 0018-00FF= reserved for T.I.
 0100-FFFE= available for assignment

block +>08-7C=	Defined by specific block type	TBD
----------------	--------------------------------	-----

A.2 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 systems.
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	+010 xxxx xNNN
	001=tape	7654 3210
	010=Wrt-Once-Rd-Many (WORM)	
	011-111=reserved	
4b. Bit 3 fixed:	0=removable; 1=fixed	+010 xxxx Nxxx
		7654 3210
4c. Bit 4 logical:	0=phys addr;	+010 xxxN xxxx
	1=logical addr	7654 3210
-----tape-----		
	bit: 765	
4d. Bit 5-7 tape opt:	000=stream only	+010 NNNx xxxx
	001=start/stop	7654 3210
	010=stream w/track select	
	011=start/stop w/track select	
	100-111=reserved	
-----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;	+014	"M"
CDC-0030) OR reserved (if reserved	+015	"A"
= >0)	:	:
	+01F	" "

Addressing:

6. # of bytes/block (required for partition table)	+020	LSB byte
	+021	MSB byte

-----disk-----

7. # of bytes/sector	+022	LSB byte
	+023	MSB byte

8. Reserved	+024-025	>0
-------------	----------	----

9. # of sectors/track	+026	byte
-----------------------	------	------

10. # of heads (tracks/cyl)	+027	byte
-----------------------------	------	------

11. # of cylinders	+028	LSB byte
	+029	MSB byte

12. # of reserved sectors for defects	+02A	LSB byte
	+02B	MSB byte

13. Reserved	+02C-02F	>0
--------------	----------	----

-----tape-----

14. # of bytes/physical block	+022	LSB byte
	+023	MSB byte

15. Reserved	+024-025	>0
--------------	----------	----

16. # of tracks (if supported)	+026	LSB byte
	+027	MSB byte

17. # of blocks (nominal)	+028	LSB byte
	+029	MDL byte
	+02A	MDH byte
	+02B	MSB byte

18. Reserved	+02C-02F	>0
--------------	----------	----

19. Volume name (16 characters of text)	+030	"N"
	+031	"A"
	+032	"M"
	+033	"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, it's use must be very carefully
controlled in multiprocessor system to
prevent problems.

31. Internal SAVE contents are processor specific. +000-3FFF

=====			
PARTITION TABLE (PTBL)		offset	value
=====			
1.	"PRTN" (identifier reserved for partition table)	+000	"P"
	Note: This is not the partition table name,	+001	"R"
	but rather a value used to identify a	+002	"T"
	valid partition table.	+003	"N"
2.	Partition table revision number	+004	LSB byte
		+005	MDL byte
		+006	MDH byte
		+007	MSB byte
3.	Number of partitions in partition table.	+008	LSB byte
		+009	MDL byte
		+00A	MDH byte
		+00B	MSB byte
4.	Size of each partition table entry in long words (32 bit words)	+00C	LSB byte
	(for current revision = >00000010)	+00D	MDL byte
		+00E	MDH byte
		+00F	MSB byte
5.	Offset in long words (32 bit) to comment	+010	LSB byte
	(for current revision = >00000004)	+011	MDL byte
		+012	MDH byte
		+013	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-FF=	reserved	

* = 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

>0000-01FF: original TI CPU type block of numbers
 >0000 = Explorer processor
 >0001 = NuMachine 68010 CPU

>0200-EFFF = blocks of CPU numbers available to
 be assigned (assigned with NuBus
 licenses)

>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)
 >FC00 = TI Lisp
 >FC01 = TI Unix System V

>FF00-FFFF: reserved for use as magic values
 (unique numbers to be controlled;
 single numbers to be assigned upon
 request as needed)

>FFFF = processor and O.S. independent
 (i.e. all "generic" bands)

12. Bytes 3 property bits (position encoded): entry +00F bit:31 byte 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

---- 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:	offset	value
-----	-----	-----
Read only test pattern of fixed values:	+000	>00
(Each byte contains the modulo 256 of	+001	>01
its own offset from the start of the	+002	>02
block.)	:	:
	+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	+014	LSB byte
(64 bit, LSB justified):	:	
	+01B	MSB byte
12. Data ECC or CRC polynomial	+01C	LSB byte
(64 bit, LSB justified):	:	
	+023	MSB byte
13. Encoding method, TEXT string	+024	"M"
(8 characters):	+025	"F"
(examples: "MFM ", "RL27")	+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:	7	6	5	4	3	2	1	0
19a. Bit 0: Nominal read		x	x	x	x	x	x	x	N
19b. Bit 1: Offset forward 1		x	x	x	x	x	x	N	x
19c. Bit 2: Offset forward 2		x	x	x	x	x	N	x	x
19d. Bit 3: Offset reverse 1		x	x	x	x	N	x	x	x
19e. Bit 4: Offset reverse 2		x	x	N	x	x	x	x	x
19f. Bit 5: Strobe early 1		x	N	x	x	x	x	x	x
19g. Bit 6: Strobe early 2		x	N	x	x	x	x	x	x
19h. Bit 7: Strobe late 1		N	x	x	x	x	x	x	x

MSB byte:

	bit:	7	6	5	4	3	2	1	0
19i. Bit 0: Strobe late 2		x	x	x	x	x	x	N	x
19j. Bit 1-7: Reserved		N	N	N	N	N	N	N	x
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:	+40E	"M"
(8 char. text string; format= "MmDdYyyy")	+40F	"m"
	+410	"D"
	+411	"d"
	+412	"Y"
	+413	"y"
	+414	"y"
	+415	"y"
25. Latest date of formatting:	+416	"M"
(8 char. text string; format= "MmDdYyyy")	+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

APPENDIX B

Acronyms

B.1 ACRONYMS

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
SPC	Processor Micro-Stack
STBM	System Test and Boot Master
TZON	Test Zone Partition
VMA	Virtual Memory Address
WCS	Writable Control Store