TI Lisp Machine
Virtual Architecture

Texas Instruments, Inc.

Version 1
March 6, 1985

This document is not Explorer user documentation. Information contained herein will be included in user documentation as appropriate. The correctness and usefulness of this information is not warranted in any way, expressed or implied.

Table of Contents

1. Introduction 5
1.1 Required Reading 5
1.2 Organization 5
1.3 Stability 5
2. Bootstrap Loading 7
2.1 Disk Format 7
2.2 Kinds of Loads 8
2.3 Microloads 9
2.4 Bootstrap PROM 10
2.5 Lisp Start 12
3. Interrupts 13
3.1 Interrupts on the LCL Processor 13
3.2 Special Programming Conventions 16
4. Device Handling 19
4.1 Device Decoding 19
4.2 The NuBus Peripheral Interface Board 20
4.3 The Keyboard 20
4.4 TV Vertical Retrace (interrupt) 21
4.5 The RS232 Serial Port 21
4.6 The Parallel Port 21
5. Virtual Memory and Paging 23
5.1 Physical Addresses 23
5.2 Virtual Addresses 23
5.3 LCL Memory Interface 24
5.4 Physical Memory Map 25
5.5 Memory Map Hardware 25
5.6 Virtual Memory System Tables 25
5.7 Memory Map Data 25
5.8 Memory Map Status Codes 27
5.9 Physical Page Data Table 29
5.10 Reverse Map Table 29
5.11 Second Level Memory Map Block Allocation 30
5.12 PDL Buffer Handling 30
5.13 Page Aging Process 30
5.14 Virtual Page Management 31
6. Internal Storage Formats 37
6.1 Q Format 37
6.2 Structure Headers 41
6.3 Invisible Forwarding Pointers 41
6.4 Symbols 41
6.5 Arrays 42
6.6 Self Reference Pointer Format 47
6.7 PDL Format 48
6.8 FEF Formats 48
6.9 Floating Point Formats 48

6.10 Bignum Format 49
6.11 Special PDL 49
6.12 CLOSURE Formats 51
7. Storage Management 55
7.1 Areas 55
7.2 Regions 56
7.3 Standard Areas 56
7.4 Systems Communication Area 59
7.5 Address Space Map Area 60
7.6 Extra PDL Area 60
7.7 Linear PDL Area 60
7.8 Special PDL Area 61
7.9 Working Storage Area 61
7.10 Macro Compiled Program Area 61
7.11 CONS 61
8. Garbage Collection 63
8.1 In the Machine 63
8.2 The Read Barrier 63
8.3 The Write Barrier 63
8.4 Incremental GC 63
8.5 Ignore 65
9. Function Calling 67
9.1 Functional Objects 67
9.2 PDL Layout 68
9.3 FEF Layout 72
9.4 Calling Conventions 78
9.5 Closure Call 78
9.6 Select-Method Call 78
9.7 Instance Call 78
9.8 Entity Call 78
9.9 ADI Formats 79
9.10 LEXPR Funcall 79
9.11 FEXPR Funcall 79
9.12 Multiple Value Returns 80
9.13 Catch, Throw, Unwind Protect, and Stack Unwinding 80
9.14 Support Vector 81
9.15 Instance Invoke 81
10. Multiprocessing 83
10.1 The Stack Group Data Structure 83
10.2 SG-State Q 85
10.3 SG Instruction Dispatch 85
10.4 SG States 85
11. Error Signalling 89
11.1 Microcode Error Conditions 89
11.2 Microcode Error Table 90
11.3 ILLOP (Illegal Operation) 95
12. Macro Instructions 99
12.1 General Format 99

12.2 Indicators 102
12.3 Kernel Macroinstructions 102
12.4 Main Instructions (Class I) 102
12.5 Non-Destination Instructions (Class II) 104
12.6 Branch Instructions (Class III) 108
12.7 Miscellaneous Instructions (Class IV) 110
12.8 Array Reference Immediate (Class V) Instructions 150
13. Flavors 153
13.1 Instance Data Structure 153
13.2 Instance Descriptor Data Structure 153
13.3 Self Mapping Table 154
13.4 Method Decode Table 155
13.5 Calling an Instance 155
13.6 Instance Variable Accessing 155

1. Introduction

Unsatisfied with the structure of normal computers, they are building at MIT's AI lab a computer whose native language is LISP. It will have 32 bits with virtual memory, and execute LISP like a bat out of hell.

In a refreshing reversal of trends, it will be for one user at a time. "Time sharing is an idea whose time has gone," chuckles one participant. (Project MAC, where time-sharing grew up, was there.)

Ted Nelson, *Computer Lib, Dream Machines*

Here comes a man
To lead you to
Your very own machine!

the Who from *Tommy*, also quoted in UCADR

This document is intended to be the definitive document both explaining and controlling the Lisp Machine virtual machine architecture for TI's line of Lisp computers. This virtual architecture draws heavily on the MIT Lisp Machine. The close resemblance is intentional. Indeed, in most cases, compatibility with the MIT Lisp Machine virtual architecture will be motivated and justified as a way of discouraging gratuitous changes.

This document describes the Lisp Machine virtual architecture as it applies to the Explorer System, Low Cost Lisp, and NuBus hardware. The microcode and software versions are as noted in "Stability", below.

1.1 Required Reading

It is assumed that any reader attempting to understand this document is familiar with the hardware architecture of Explorer System and of the Low Cost Lisp processor. At minimum, the processor general description document should be read.

In addition, extensive familiarity with Lisp Machine Lisp is also expected of the reader. No amount of background in this area can be excessive. The language concepts manual should be familiar to the reader as well as the language reference manual.

1.2 Organisation

This document is organized in a basically bottom-up fashion. The lowest levels of the Lisp Machine will be described and each chapter will generally describe a higher level of the system. The first chapter, therefore, is bootstrap loading which explains how the system is started. Next are interrupts and devices as these form the basis for the next level, virtual memory. Next will come Lisp data object formats, then storage management. Next is function calling which is the basis of the higher level functions. Next is multiprocessing suport and error signaling which uses it. Finally, we introduce the macro-instruction set. Last support for instances of flavors is reviewed.

Of course, a complex system such as this is seldom organized as a strict hierarchy. Levels are often mixed and circular and pseudo-circular dependencies arise. The Lisp Machine system is not hierarchical, but may be described as hierarchical. The main contributor to non-hierarchical organization seems to be level flattening which is the squashing together of several levels. Level flattening contributes to performance, and therefore, is a useful tool for constructing high performance systems.

1.3 Stability

The Lisp Machine Virtual Architecture is subject to change as the system evolves and matures. This document matches Lisp Machine System TI Lisp System HAL 1.0 which was released on Jun 1, 1985. It also matches system microcode Control 182, which was released on Feb 15, 1985. This document will be re-released whenever the Lisp Machine Virtual Architecture changes in a Lisp Machine System.

For historical perspective, the previous system was TI Experimental 0.0 which was released on Oct 26, 1984. The previous microcode was URaven 0.0 which was released on June 2, 1984.

2. Bootstrap Loading

The unpleasant sensations of the start were less poignant now. They merged at last into a kind of hysterical exhilaration. I remarked, indeed, a clumsy swaying of the machine, for which I was unable to account. But my mind was too confused to attend to it, so with a kind of madness growing upon me, I flung myself into futurity.

H. G. Wells, *The Time Machine*

NOTE: This portion of the architecture is undergoing change. This chapter should be contributed by someone associated with the changes.

Bootstrap loading the Explorer Systeminvolves 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 chapters. The three kinds of loads are cold start, warm start and diagnostic load. These will be covered separately below.

All loads involve setting up the writable control store and other memories of the processor and then transferring to the microcode just loaded. The microprogram responsible for this task is in the boot PROM which is enabled upon power-on or by a boot request from the keyboard.

Bootstrap loading is from disk. Since the paging system[1] requires a disk, requiring a disk for bootstrap loading does not exclude any already excluded system configureations. The first module loaded is always a microload[2] which loads writable control store and other processor memories. This module may do additional loading of main and virtual memories from disk or other source.

2.1 Disk Format

The first-level format of the disk is described by the disk label. The disk label gives information specific to this media such as physical sector size and defines regions of the disk known as partitions. Logical sectors are called disk blocks. Blocks are always 1024 bytes (which is 256 words). Block size is the same as a virtual memory page.

The disk label always begins in logical block 0 of the disk and may also use successive even blocks of the disk. For each even block used, the corresponding odd block reserved. The boot PROM saves a page of memory into the odd disk block corresponding to each block of the label before reading the label into that page of memory. This is required for proper warmstart operation.

The label has the format shown in *Table 2-1*. ***This is the CADR style format and has not yet been altered for the Explorer System.***

Each partition is a contiguous region of the disk, specified by its starting block address and its length. Partitions are described in the label with a partition table entry of the form shown in *Table 2-2*.

There are several naming conventions in force for partitions. All are 4 character long names in the LISP Machine character set.[3] Several partition names are used by the software for special purposes. These are shown in *Table 2-3* for reference.

---

[1] See section on paging.

[2] See section on mcr.

[3] The LISP Machine character set is the same as ASCII if restricted to alphanumerics and the eigth bit is set to 0.

Word 0:	"LABL", four characters stored little endian (as are all strings)
Word 1:	label version number, must match version that PROM can read: this is format of version 1
Word 2:	total number of cylinders
Word 3:	number of data heads (tracks/cylinder)
Word 4:	number of blocks (1K byte) per track
Word 5:	blocks per cylinder (*heads* · *blocks*)
Word 6:	name of default microload partition
Word 7:	name of default load band partition
Words 8-15:	name of this drive ** how is this different than "name of this pack"**
Words 16-23:	name of this pack
Words 24-48:	label comment
Words 128:	number of partitions
Words 129:	number of words per partition descriptor
Words 130:	start of partition table

Table 2-1 Disk Label Format

• Word 0:	partition name
• Word 1:	partition start address
• Word 2:	partition size
• Word 3-*entrysize*:	partition comment

Table 2-2 Partition Table Entry Format

PAGE	virtual memory swapping storage
FILE	file system storage, local directories and files are within this partition
METR	if using metering system, microcode meter events are recorded here

Table 2-3 Dedicated Partition Names

In addition, there are several names that are used by the loader. If a net load is requested, the **NETB** partition is microloaded. The normal names of the system microloads are "**MCRn**" where "*n*" is a digit. The boot PROM does not enforce this naming convention. The normal names of system loads (bands) are "**LODn**" and "**LDnn**" where "*m*" and "*n*" are digits. The convention is also not enforced.

2.2 Kinds of Loads

There are four kinds of loads, cold start, warm start, trial load, and diagnostic load. Each has fairly different requirements, but they are largely the same.

2.2.1 Cold Start

A cold start is started after power-on or when simulating a power-on restart of the system software. If possible, a cold start is initiated automatically after power-on.[4]

A cold start performs kernel processor selftest, performs system test, loads the writable control store and internal memories of the processor with system microcode, and starts the

[4] This feature is known to some as "self bootstrapping".

1	control store (I-Mem)
2	dispatch memory (D-Mem)
3	main memory
4	A M memories
5	tag classifier memory (T-Mem)
6	entry control

Table 2-4 Section Type Codes

system microcode. The system microcode then proceeds to start the virtual memory system, initialize virtual memory to a load band, start Lisp storage management, and begin Lisp by running the initial Lisp function. The initial Lisp function is responsible for initializing the other higher-level software systems such as the file system and the window system.

2.2.2 Warm Start

A warm start occurs when the user requests that a halted system continue. A warm start performs kernel processor selftest, reloads the writable control store and internal memories of the processor with system microcode, and restarts Lisp pretty much where it left off. Notice that a warm start must not alter any page of memory used by Lisp.

2.2.3 Trial Load

A trial load is a way for a microload and load band to be loaded temporarily. A trial load is initiated by Lisp. It loads writable control store and internal memories of the processor with system microcode, and starts Lisp in the same way as cold start. The microload and load band for a trial load are specified by Lisp and not by the disk label *** or is it NVRAM ***. The selected microload and selected band are never set by a trial load so that if a trial load proves fatal, a cold start of a good microload/band is easy.

2.2.4 Diagnostic Load

A diagnostic load is a stand-alone microload. It is assumed that the purpose of a diagnostic load is largely diagnostic. After kernel selftest and system test, a microload is loaded and started. These microloads will usually be for diagnostic or maintenance purposes.

2.3 Microloads

The writable control store and other internal memories of the 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 Low Cost Lisp microassembler produces an output file in the microload format. The format of a microload file is the same as the format of a "MCR" partition on disk, and this compact representation can be placed directly into a disk or tape partition and loaded by the standard bootstrap PROM. It provides for the loading of I-Mem, A/M memories, D-Mem, and main memory.

2.3.1 Microload Format

A microload file is organized into 32-bit words and is divided into sections. Each section specifies one of the above memories, or some control information.

Each section begins with a word specifying the section type. The codes are given in *Table 2-4*. The formats of each kind of section is given below.

If the section is for I-Mem, D-Mem, A/M-Mem, or T-Mem, the next word of the section contains the starting location in the selected memory, and the third word contains the number of locations of the selected memory to be loaded. The second and third words of Main-Memory and Entry Control sections have other meanings described below.

2.3.1.1 I-Mem Section

For each I-Mem location represented in the section, there are two 32-bit words in the microload. The first of each pair is the high order part of the I-Mem word which is made to appear on the A side during the I-Mem write. The second word of the pair is the low order part which is made to appear on the M side during the I-Mem write.

2.3.1.2 D-Mem Section

For each dispatch memory location represented in the section, there is a single 32-bit word in the microload. The 17-bit D-Mem word is stored right-justified in the 32-bit microload word.

2.3.1.3 Main Memory Section

The second word of a main memory section in the microload contains the number of blocks to load. The third word of the section is the relative block number of the data within the microload file. In a disk partition, this is the block number of the data relative to the beginning of this microload partition. The fourth word of the section is physical memory address where the data should be loaded. The PROM program uses this data to request that the disk controller load the data at the appropriate memory address.

If the number of blocks in a main memory section is zero, the third and fourth words are ignored. This is used in microloads which are microcode augmentation files to indicate the microcode version number for which this augmentation is valid. The required version number is stored in the fourth word of the section. *** check if incremental assembler does this ***

*** Main Memory section has some mistake in it — if only I could remember what it is ... ***

2.3.1.4 A/M Memory Section

Words in the MCR are loaded into consecutive locations in A and M memories. Locations less than 64 are stored into both A and M memories. Other locations load A memory only.

The memories are not actually loaded when this section is processed, but an image of them is stored in the PDL buffer. This image is copied into the A and M memories just before entering I-Mem. Several A/M sections can reside in a single microload. The last value loaded for a particular register remains in effect. *** What values do unloaded locations get? Zero I think. How is info passed from PROM to loaded module? some A-Mem locations are used ***

2.3.1.5 Tag Classifier Memory Section

The tag classifier RAM is loaded from the microload by transferring the contents of each word into 32 successive locations in T-Mem. Each word represents the boolean vector for a class. The bits of the word correspond to the datatypes. Bit 0 corresponds to datatype 0. If the corresponding bit in the microload word is a 1, this datatype is a member of the class. The location is the class number to begin loading with. The count is the number of classes to load.

2.3.1.6 Entry/Control Section

The entry/control section is to specify other random information and to set the I-Mem entry address. The first word of the section is a count of how many words follow. If the count is 1, the second (and last) word of the section contains the address to enter in I-Mem. Control is immediately transferred there. This is the last section in the microload.

If the count is not 1, the third word is not interpreted as an entry address. Instead, count is the number of words in this section. Processing of the microload continues with the next section.

Currently, there is no use or format specified for this control record.

2.4 Bootstrap PROM

The bootstrap PROM is read-only, non-volatile memory in the Low Cost Lisp processor that contains selftest and loader microprograms. When the system is powered-on or a system-wide hardware reset is issued, the boot PROM receives control. It also receives control when a load request is made from the keyboard and provides entries for a software requested reload.

2.4.1 Power-On Selftest

When entered from power-on, the boot PROM is responsible for performing a kernel selftest to assure the processor is functional. If it detects a failure, it indicates the nature of the problem in the processor fault lights. It tests internal memories and datapaths. It will also test main memory. The memory test should indicate progress on the console. Memory test cannot be performed until after NuBus configuration.

2.4.2 Preliminary NuBus Configuration

It is necessary to search NuBus for a module claiming to contain each of: memory, console, non-volatile memory, and disk controller. This should find all local memory, one NVRAM, one console, and one disk controller. These resources are needed for loading.

The non-volatile memory is searched for first. Slots with consecutively lower ID's are searched starting with the slot occupied by the processor.[5] This allows several processors to coreside on a NuBus, each having its own resource. The NVRAM, if found, might contain the disk controller slot ID to use. The ID found in the NVRAM is tried, and if the device in that slot proports to be a disk controller, it becomes the controller for the boot disk. The NVRAM also contains the unit number of the boot device. If this corresponds to a working device, that becomes the boot device. If this unit is not valid or the indicated unit failed selftest, the lowest numbered working unit is selected as the boot device.

If the non-volatile memory is not found or the disk controller ID is invalid, the NuBus is searched again in the same manner for the disk controller, and the boot device is selected as the lowest numbered working unit on that controller.

The console is also found by searching the NuBus in the same manner, starting again with the slot containing the processor. The console found will be used to display messages and menus during booting. Errors encountered after this point should attempt to display a terse message to the console.

This is a good point at which to display a herald indicating the selected devices.

Some memory must be found. NuBus is searched in the same manner for memory. The first memory found must provide at least 2KB of contiguous read/write memory. With this much configuration performed, all devices required for loading have been located.

2.4.3 Read Boot Request

Next the keyboard port is read. If it is not ready, then no request has been made and this is a power-on restart. Power-on should cause a "default cold boot" request for autoload. If the boot request is a warm boot, copy memory page 0 to the boot disk block 1 so as not to destroy any of virtual memory when reading disk blocks.

2.4.4 Bootload Notification

Next the user should be notified that a load of the selected type is proceeding from the selected device.

---

[5] Starting with the slot containing the processor allows for some resources to be provided by the processor board in high density implementations.

2.4.5 Power-On Memory Test

If a power-on load, run a fast memory test. Test should use no more than a few seconds per megabyte. Should update a progress indication every few seconds. Test only local memory, if there are other processors, they may also be testing their local (or non-local memory).

2.4.6 Boot Type

Select one of Diagnostic Menu, System Load Menu, Network Load, Warm Start or Default Load based on boot request.

2.4.6.1 Diagnostic Menu

If diagnostic boot request, present menu of devices and get boot device choice. Then show a menu of the diagnostic microloads on that device. Get a choice and if valid proceed to microloading. One menu entry returns to the menu of devices, one gets the system load menu.

2.4.6.2 System Load Menu

If verbose system load, present menu of devices and get boot device choice. Then show a menu of the system loads on that device and get a choice. Next show a menu of microloads that will run with that system load. Get a choice. If choices valid proceed to microloading.

2.4.6.3 Network Load

If request is for network load, the microload will be NETB partition, which contains the network loader. The network loader is loaded and started. It performs the network load function. Proceed to microloading.

2.4.6.4 Warm Start

Ascertain what microload was running from a special A-memory location *** location 64? ***. It will be reloaded. Virtual memory will not be reloaded. Proceed to microloading.

2.4.6.5 Default Load

If NVRAM is valid, it specifies the current system load device which must be a disk. If the NVRAM is not valid, proceed to System Load Menu. The label of this device indicates the selected microload and system load. Proceed to microloading.

2.4.7 Microload

The processor internal memories are loaded from the selected partition of the selected device. Then control is passed into the WCS (I-Mem).

2.4.8 PROM Entry Vector

*** give something on what PROM provides and how it can be accessed. ***

2.5 Lisp Start

If the microload loaded is Lisp system microcode, when it is started it must start Lisp. The first task is to initialize the paging tables and start virtual memory. Then, **LISP-INITIAL-FUNCTION** is called. The initial function forms the base of the initial process. **LISP-INITIAL-FUNCTION** invokes **LISP-REINITIALIZE** which is responsible for performing any initializations that are applicable *** boot-initializations? ***. *** more here ***

2.5.1 Virtual Memory Start

This topic will be covered in section on paging.

2.5.2 LISP Initial Function

LISP-INITIAL-FUNCTION is the first lisp function called in a newly booted machine. It forms the the base function of the initial stack group. **LISP-INITIAL-FUNCTION** calls **LISP-REINITIALIZE** to reset various variables and perform initializations. **LISP-REINITIALIZE** calls **PRINT-HERALD** to indicate version information. *** blah blah blah ***

3. Interrupts

"Don't interrupt!", said Gandalf. "You will get there in a few days now, if we're lucky, and find out all about it."

J.R.R. Tolkein, from *The Hobbit*

This chapter describes interrupt handling on the Low Cost Lisp, and will be of particular interest to those who require real-time response to hardware events. An interrupt handler must written in microcode to service the interrupt. This chapter describes how interrupts are detected, how the system data structures are used, and the programming constraints and conventions.

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[1], garbage collected memory, or macro instructions.[2]

An interrupt is caused by an I/O device requesting service, interprocessor communicationanother processor signalling an event, and 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 wired[3] 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. *** it would be much nicer if there were, even a very long one like 20mS *** 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. Mostly, 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.

3.1 Interrupts on the LCL Processor

The Low Cost Lisp processor has hardware to ease the detection and processing of interrupts. Since the Explorer System 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

---

[1] Implementation Note: Virtual memory service is divided into two levels: map handling and page handling. Map handling is regarded as VERY low level and permitted in the interrupt context. Both levels are described together in the Virtual Memory Section.

[2] Note: In Lisp Machine Lisp, the term "interrupt" is used to refer to a process switch. This unfortunate choice of terms creates some confusion where asynchronous hardware event signals (which are conventionally called "interrupts") must be discussed. In the microcode and in this document, "interrupt" will be used to discuss hardware event signals and "sequence break" will be used to discuss Lisp-level process switching.

[3] See section on wired memory.

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) Bootrequest
FsE00000	0	(Preemptive) Powerfail

Fig. 3-1 LCL Control Space

space of the Low Cost Lisp processor. A map of the interrupt locations and the priority of each is shown in *Fig. 3-1*.

Interrupt pending is a condition testable individually and in combination with the page fault[4] and sequence break[5] 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 *Fig. 3-1*. The highest priority level with an interrupt pending is indicated by a special field in the machine control register (MCR) of the Low Cost Lisp. Since there are a moderately large number of interrupt levels, many levels can be allocated to a single device. 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. shown in *Fig. 3-2*. This figure shows the basic structure of the block and the details required by the interrupt handler. Device descriptor blocks are described in more detail in chapter DEVICE HANDLING (Steve, how do I do this reference).

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 (0) indicates the null link or empty list. The interrupt vector table is shown in figure *Fig. 3-3*.

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 coresponds to the interrupt level of the device. Interrupt levels 0 and 1 are dedicated to Powerfail and Bootrequest 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

[4] See section on paging.

[5] See section on multiprocessing.

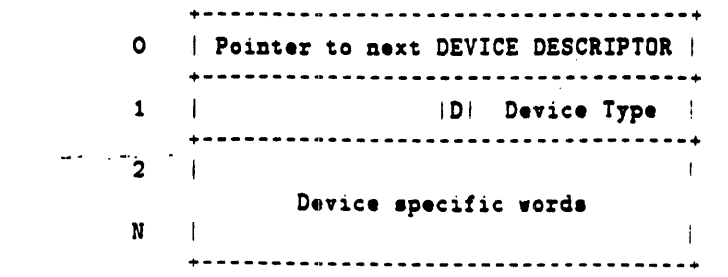


Fig. 3-2 Device Descriptor Block

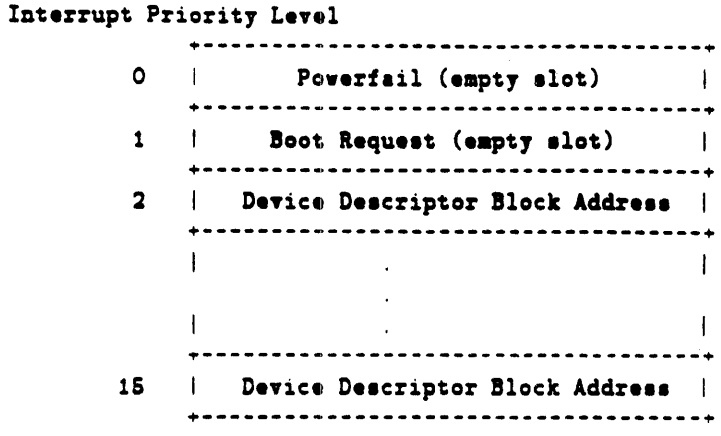


Fig. 3-3 Interrupt Vector Table

events then the processor failed to trap. The interrupt processor will consider this a hardware fault and cause the machine to halt.

For interrupt 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 device descriptor blockdescriptor 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 then the interrupt processing is complete and the interrupt service routine returns to its caller. If an interrupt is pending then the entire process is repeated.[6]

---

[6] Note that this scheme can leave an interrupt pending in a polled level that was set by a device after the pending interrupt was cleared (before polling the devices on that level in response to an interrupt by some other device on that level), but before this device was polled. This device will recieve service on the first scan of all devices but leave an interrupt pending. Thus, the polling routine should not error if it finds no devices needing service when it polls in response to a pending interrupt.

A-INTR-VMA	saved VMA
A-INTR-MD	saved MD
A-INTR-A	saved M-A
A-INTR-B	saved M-B
A-INTR-T	saved M-T
M-FLAGS	machine flags
M-DEV-A	General interrupt use
M-DEV-B	General interrupt use
M-DEV-C	General interrupt use
M-DEV-D	General interrupt use

Table 3-1 Registers Used by Interrupt Processing

Note: There should be some limit on this loop in case the interrupt status won't clear. The limit should be fairly large (100) and cause an illop. This is so the machine doesn't just go autistic. Any service request flags in the device itself must also be reset by the device specific interrupt handler.

3.2 Special Programming Conventions

Since interrupt processing is the lowest level in the system there are many restrictions on interrupt processing code. Those restrictions are called out here.

3.2.1 Register Usage

Interrupt processing must not clobber registers that are used by higher levels. It may only alter the registers listed *Table 3-1*. Several M-memory registers are saved in these to allow the use of some M registers by interrupt handlers. Because interrupt processing must touch memory. VMA and MD must also be saved. The saved registers must be restored before resuming higher level processing. Several M registers are defined for general use during interrupt and device handling operations. The conventions and uses of these registers is defined by the interrupt handler and the associated device handler. These registers will not be preserved between calls, and no higher level routines can expect these registers to be preserved accross interrupt handling.

3.2.2 Memory Usage

Interrupt processing is below the level of virtual memory and therefore must not encounter a page fault. To insure this, interrupt handlers are restricted to use only wired memory and I/O space in virtual address space and physical NuBus addresses. I/O space is a part of the virtual address space that is not paged, but refers to blocks of physical memory used for I/O. Currently I/O space contains only the video (TV) buffer.

Interrupt handler accesses to virtual memory are restricted to wired pages and I/O space. These accesses will never cause a reload from disk but may require a map reload. A flag in the M-FLAGS register indicates that interrupt processing is in progress. Virtual memory swap handling will check this flag to enforce this restriction. Interrupt handler accesses to virtual memory should be checked with

```
CHECK-PAGE-READ-NO-INTERRUPT
```

and

```
CHECK-PAGE-WRITE-NO-INTERRUPT
```

3.2.3 Recursive Invocation

No interrupt may be recognized inside of an interrupt handler, although an interrupt handler may test for another interrupt before exiting and also process that interrupt. There is, therefore, no preemption in interrupt processing. This in not a serious problem since interrupt handlers do little work and therefore complete in less time than the variability between polling for interrupts in non-interrupt processing.

3.2.4 Higher Level Events

The handling of an interrupt is indivisible to any higher level. Interrupt processing, therefore, must not invoke any higher level event such as page swap page fault or sequence break, though a flag may set to indicate that the latter service is required at the next convenient time.

4. Device Handling

Don't touch me there!

the Tubes

This chapter explains device handling in the Explorer System. It explains the handling of the disk, keyboard, serial communications, ... The graphics display (TV) and the mouse are not described here as they are handled specially to make the user interface more powerfull.

The device handling features and conventions on the Explorer System have been designed for simple but flexible operation, with few restrictions, relating mostly 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.[1] 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 in the Explorer Systemsystem.

4.1 Device Decoding

Each device in the system that requires interrupt processing maintains a Device Descriptor Block (see section on interrupts). This descriptor block contains all the information that the system needs for processing requests and interrupts for this device. The Device Descriptor Block maintains information about the device state. A separate structure, the Request Block (RQB) is used to transfer request information. The first word in the Device Descriptor Block is used as a link word for the interrupt decoder and points to the next device descriptor on the same interrupt level. The second word is called the device information word. This word contains information needed to determine what type of processing in needed when we want to initiate an I/O request on 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 is used. The parameters are the device descriptor address 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 (the value of the fixnum) specifies the operation being requested. The second form is as an array. The request block is an array (a block of contiguous storage space). 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 we may like to be able 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 the interrupt handler automatically starts the next request. This requires that the interrupt handlers for a device maintain the queue as part of its duties.

---

[1] Devices that do not require interrupt handling need only to be known at the Lisp level. The conventions here do not apply to devices that are operated entirely by Lisp. The Lisp code that operates the device defines the interface and conventions for use.

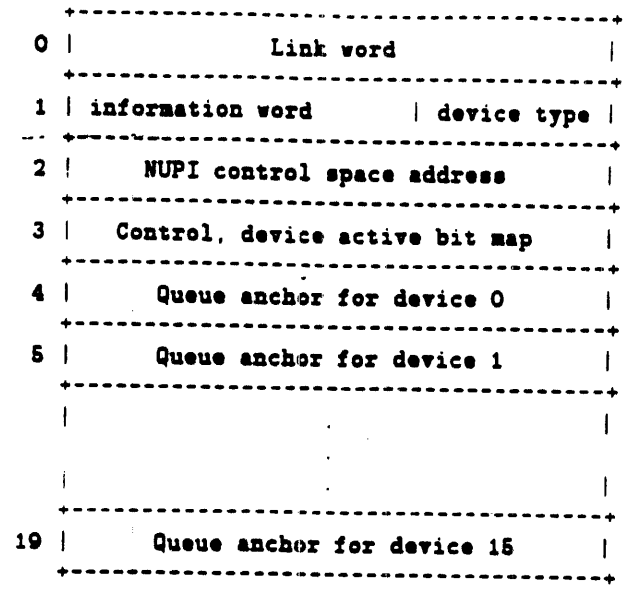


Fig. 4-1 NUPI Device Descriptor Block

The remainder of this chapter describes specific devices and their IO/Interrupt handlers.

4.2 The NuBus Peripheral Interface Board

The NuBus Peripheral Interface (NuPI) is used for interface to disk and tape devices. The NuPI receives operation requests by writing the physical NuBus address of a NuPI Command Block to a special address in the NuPI address space designated as the Commmand Register. The NuPI then processes this command asynchronously from the Low Cost Lisp 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 then, if specified in the request block, posts an event to the Low Cost Lisp processor to signal completion of a command. By convention all requests to the NuPI in this system will post the completion event. This event posting is fielded by the Low Cost Lisp processor as an interrupt. The device interrupt handler is called to process the completion the current request and initiate the next one if required.

Fig. 4-1 shows the format of the Device Descritor 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 then 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 we must check all devices with requests in process to find which ones have completed. The bit map allows polling of only those units for which it is possible to need processing.

4.3 The Keyboard

When a keyboard interrupt occurs, the keyboard character is copied from the hardware register into the keyboard buffer. The keyboard process will notice that a character is available and handle it.

4.4 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 handler does the timer too.

4.5 The RS232 Serial Port

**** write something here. ****

4.6 The Parallel Port

**** write something here ****

5. Virtual Memory and Paging

Swap read error. You lose your mind.

COOKY, a fortune cooky program

Virtual memory is the simulation of a large fast primary store by the use of a fast but smaller primary store and a large but slow secondary store. *Denning70* Blocks, called pages, are moved between primary and secondary store according to a page management strategy.

A page management strategy that moves a page into primary store when it is used but not present in primary store 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 store, it must be written to secondary store 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 Lisp Machine system, semiconductor memory is used as primary storage and disk is used as secondary storage. Pages are moved into primary store when used and not present — demand paging. Every attempt is made to replace a page which is not dirty so that a write to secondary store is not needed.

A page exception occurs when for some reason the virtual to physical address mapping could not be completed with a valid memory operation without microcode support. There are many reasons for this as discussed below. 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 occured. This destinction is made so that page exception rates and page fault rates can be put in perspective.

Below, the details of the Lisp Machine paging system are described. Paging is below everything except interrupts in the Lisp Machine heirarchy. They are depended on by everything except interrupts.

5.1 Physical Addresses

Memory is accessed by presenting it with a physical address. A physical address is a system-wide name for some storage. The Explorer System is based on the NuBus. NuBus is a 32-bit, high-speed bus. All NuBus addresses are byte addresses. Words are aligned so that the low order 2 bits are zero. The bus incorporates fair bus arbitration and supports block transfers of up to 16 words. NuBus also supports 8-bit and 16-bit accesses. The Low Cost Lispprocessor cannot be a NuBus block transfer master. It can perform 8-bit and 16-bit memory accesses but not using the memory map.

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 necessary because the 25-bit virtual address is smaller than the 32-bit NuBus address space.

5.2 Virtual Addresses

An address in the Explorer System is the size of the 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 the page offset is the low order 8 bits of the virtual address. Thus, each page contains 2^8 words of storage.

The Low Cost Lisp 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 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 map to produce the 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 (used to indicate various per-region attributes defined later), and 2 garbage collector volatility bits.

5.3 LCL Memory Interface

*** fix this *** The LCL 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 LCL's own private bus to this memory reduces the NuBus traffic.

There are some special microcode accessible registers in the LCL that are associated with memory. The Memory Address Register (called VMA, meaning *Virtual Memory Address*) holds up to a 32-bit address. Addresses to the NuBus are 32 bits. Addresses to the Virtual Memory subsystem are 25 bits. The Memory Data Register (MD) contains the data to be written to memory, or the data that was read from memory, depending on the operation performed. These are the two main physical registers.

There are several more "logical" registers in the LCL. All of these to be mentioned here physically coincide with the VMA and MD registers mentioned above, but have some important side effects. To start a memory operation the microcode programmer need only to reference the appropriate logical register (as a destination). The registers are as follows:

1. VMA — store a 32-bit value in the memory address register. No other effects.
2. MD — store a 32-bit value in the memory data register. No other effects.
3. VMA-Start-Read — store a 32-bit value in the memory address register and start a virtual memory read operation using the least significant 25 bits as the virtual address. At the completion of the operation the memory word referenced will be found in the MD register. The virtual memory operations are discussed later.
4. VMA-Start-Write — store a 32-bit value in the memory address register and start a virtual memory write operation using the least significant 25 bits as the virtual address. The data word to be written to memory is contained in the MD register.
5. MD-Start-Read — store a 32-bit value in the memory data register and start a virtual memory read operation.
6. MD-Start-Write — store a 32-bit value in the memory data register and start a virtual memory write operation.
7. VMA-Start-Read-Unmapped — store a 32-bit value in the memory address register and start a physical NuBus read operation.
8. VMA-Start-Write-Unmapped — analogous.
9. MD-Start-Read-Unmapped — analogous.
10. MD-Start-Write-Unmapped — analogous.

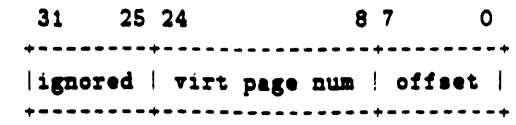


Fig. 5-1 VMA Register Format

11. also byte and halfword forms.

Please note that the concept of mapped and unmapped references, and the concept of local and NuBus memory (address space) are independent. The virtual memory mapping includes the entire 32-bit NuBus address space. The local memory bus is accessed from the LCL if the address referenced falls in the NuBus address space assigned to local memory.

A Low Cost Lisp virtual address consists of 25 bits. The remaining 7 bits of a word are used for type and control information. This additional information is not of consequence for this discussion.

The symbol VMA is used to represent the Virtual Memory Address, a hardware register, that is the address input register to the memory map system. The VMA register is 32 bits wide, however, the virtual memory system is concerned with only the least significant 25 bits. The 25-bit VMA is divided into a 17-bit virtual page number and 8 bits of word address within the page (see *Fig. 5-1*.)

5.4 Physical Memory Map

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.

If Lisp macrocode functions need access to the data items that reside in physical memory, the microcode will provide Misc-Ops to return the data in a virtual memory system compatible format.

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 second, or main, level consists of 128 blocks of 32 registers each. The first level is indexed by the high 12 bits of the virtual page number and specifies the block number in the second level. The remaining 5 bits of the virtual page number select a register within that block.

5.6 Virtual Memory System Tables

There are some additional tables associated with paging: 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. The exact data and function of these tables will be described later.

5.7 Memory Map Data

When a page exception occurs the microcode can read the status of the memory operation from the memory map hardware. The microcode reads the data from special registers in the machine. The first level map is read from the functional source Memory-Map-Level-1 (see *Fig. 5-2*).

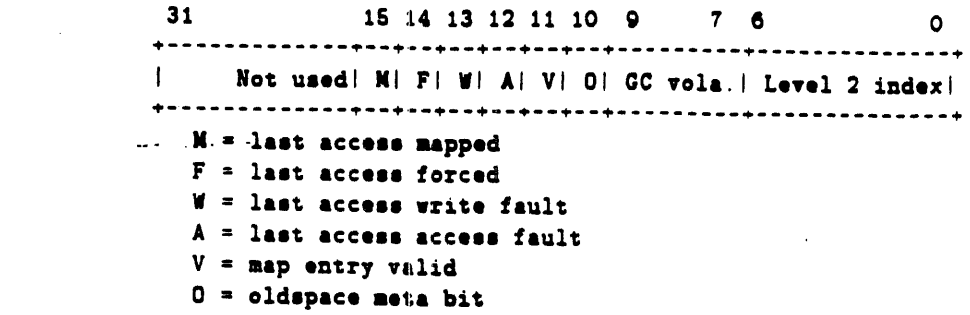


Fig. 5-2 Memory-Map-Level-1 Register Format

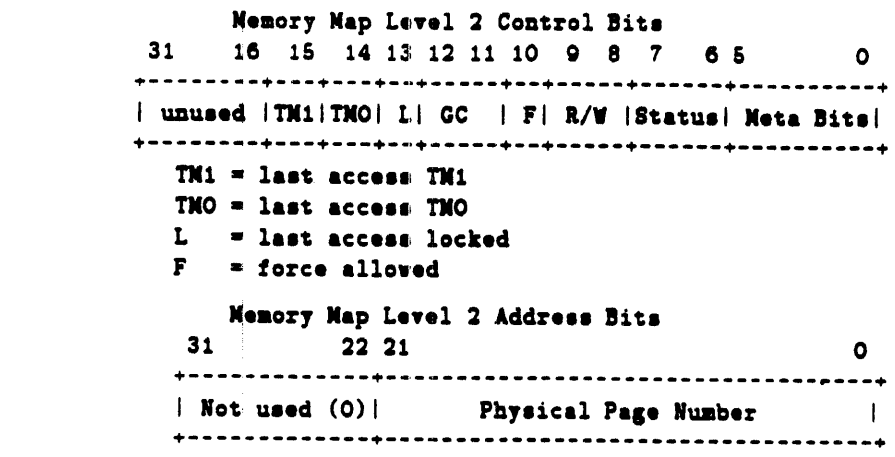


Fig. 5-3 Virtual Memory Map Data Format

The second level map is read in the as two separate functional sources since the data field is greater than 32 bits.[1] The field is separated into Memory-Map-Level-2-Control-Bits and Memory-Map-Level-2-Address-Bits (see *Fig. 5-3*).

To handle a page exception a check must first be made to see if the first level map entry addressed by VMA is valid. A bit in the Memory-Map-Level-1 register indicates this. If the first level map entry is not valid, a block of second level map must be allocated[2] and initialised with "map not valid" entries. The first level map must be set up to point to it. From here the page exception is handled as a second level map miss, described later.

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

The discussion below refers to data fields in the second level map control register.

On the LCL microprocessor, two flags from the map can be substituted into the result of the byte extraction part of the **DISPATCH** instruction. The flags are the GC volatility fault flag

[1] The hardware documentation numbers the bits 34 to 0 as well as describing them as functional sources and destinations as they are described here.

[2] See section on mapl2alloc.

Value	Access Rights
0	no access
1	no access
2	read only
3	read/write

Table 5-1 Access Bits Values

Bit field	Use
0 - 1	Not used.
2	Oldspace bit.
3	Extra PDL bit.
4 - 5	Region representation type.

Table 5-2 Meta Bits Values

Status	Meaning	Access	Swapped-In?
0	Map Miss	None	maybe
1	Meta Bits Only	None	maybe
2	Read Only	R	yes
3	Read/Write First	R	yes
4	Read/Write	RW	yes
5	Page might be in PDL Buffer	None	yes
6	Possible MAR Trap	None	yes
7	not used		

Fig. 5-4 Memory Map Status Codes

and the oldspace bit from the level one map. This feature is used by the transporter and GC-write-test dispatches (which are associated with garbage collection).

Bits 8 and 9 taken together are called the access bits. The values of the access bits are shown in table *Table 5-1*.

The Meta Bits are defined in table *Table 5-2*.

Bits 6 - 8 (note that bit 8 is shared between the status and access fields) form the map status code. The status bits are dispatched on by the page exception handler to find out how to handle the exception. The possible map status values and their interpretations are shown in *Fig. 5-4*. The access status specifies what memory operations. read. read/write, or none, are permitted by the hardware. If access is permitted the memory system performs the operation. If access is denied then no memory operation is performed and the page fault condition is set. Later sections detail the processing required for each case in the map status table.

5.8 Memory Map Status Codes

In this section each memory map status code is examined in detail for the possible causes of a trap with this code and the appropriate actions for handling this case.

5.8.1 Map Miss: code 0

Any reference to a page with a Map Miss status code will cause a page exception. To handle the map not valid case the following steps take place. Check to see if the page is in the A-memory map space. If so. the operation is simulated. Otherwise. consult the page hash table (PHT, see

below) entry for this virtual address to get information about the page. The action taken from here depends on data in the PHT table.

The data in the PHT will indicate if the page is in physical memory. The PHT contains an entry for each virtual page that is resident in physical memory. If an entry is found in the PHT for this virtual address then the information in that entry describes the status and location in physical memory for this page. The memory map is set up with this data and the memory reference is restarted

If there is no entry in the PHT for the virtual page referenced then this page is not in physical memory. It must be brought in from secondary storage. The mapping of a virtual page to a disk address, the physical memory page frames to be used, and the disk operations are refered to as the swapping process. The details of this process are described later. For now, assume that the virtual page is read from the disk into physical memory, the map is set up to refer to this physical page frame and the memory reference is restarted. The details of the PHT are described in a later section.

5.8.2 Meta-Bits-Only: Code 1

A page exception will be generated for any page that has a map status of Meta Bits Only. This code indicates that this map entry contains meta bits information but does not contain page-location information. This type of map entry is created when a pointer to an object is used but the object itself is not referenced. The meta bits in such a map entry are needed by the garbage collector. An attempt to access the storage associated with the object will be treated like a map miss.

5.8.3 Read Only: Code 2

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

If the operation is a regular write then the operation is declared illegal and an error is signaled.

5.8.4 Read/Write First: Code 3

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

If the page that is being set as dirty is currently assigned to a read-only page band then it will be reassigned to a read/write page band when the page needs to be swapped out of physical memory.

5.8.5 Read/Write: Code 4

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

5.8.6 Page might be in PDL Buffer: Code 5

Certain areas which are used to contain PDLs arrange to get the map set up this status for their pages (instead of 4, read/write). The microcode has to decide, on every reference, whether the page is in the PDL buffer or in main memory, and simulate the appropriate operation. It may be that only part, or none, of the page is in the PDL buffer on a particular reference. Thus the page exception handler must test the virtual address to see if it falls in the range which is really in the PDL buffer right now. If not, temporarily turn on read/write access, make the reference, and turn it off again. (Note: if the page is memory resident maybe physical addressing would

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

Table 5-3 MAR Status Codes

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.

Table 5-4 Physical Page Data Area word format

be simpler.) Pages may be swapped out without regard for whether they are in the PDL or not. This works because the normal course of swapping out invalidates the 2nd level map. If the page is then referenced as memory, it will be swapped in normally and its map status restored from the **REGION-BITS** table, in the normal fashion. This will then restore the Maybe-PDL map status. Otherwise, the addressed word is in the PDL buffer. Translate the virtual address to a PDL buffer address and make the reference.

5.8.7 Possible MAR Trap: Code 6

The memory address register (MAR) facility allows any word or contiguous set of words to be monitored constantly, and cause a trap if the words are referenced in a specified manner. The name MAR is from the similar device on the ITS PDP-10's. The MAR trap status is set for all pages that are in the range of addresses being monitored. When this trap occurs the virtual address is checked to see if it falls in the range. If so, a sequence break occurs. It should be noted that sequence breaks are not allowed during stack group switches, so if a MAR monitored address is referenced a sequence break flag is set and the break will occur at the next appropriate time. Two A memory locations are associated with the MAR break feature. The register **A-MAR-HIGH** contains the highest virtual address to be monitored and the register **A-MAR-LOW** contains the lowest monitored address. If the address falls within this range then a dispatch is executed on the variable **M-FLAGS-MAR-DISP**. The action taken for various flag word values is shown in *Table 5-3*.

5.9 Physical Page Data Table

The Physical Page Data Table is a physical memory resident table with one word for each page of main memory. When the system is booted, the microcode determines the size of main 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 5-4*.

All the 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 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.

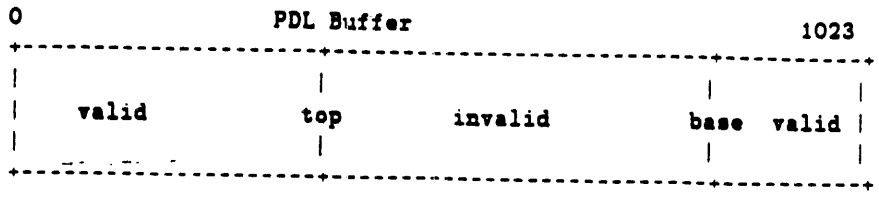


Fig. 5-5 PDL Buffer Wrap Around

5.10 Reverse Map Table

For each block of 32 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 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.

The Reverse First Level Map information is used when allocating map level 2 blocks. See Second Level Memory Map Block Allocation.

5.11 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 then 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 is 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) is set then this entry is in use. The 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.

5.12 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. For instance see *Fig. 5-5*.

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

5.13 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 that a needed virtual page can use this physical memory page frame.

Virtual Memory Size = 32M Words

Physical Memory Size	PHT Size
512K Words. 2048 pages	8192 words
1M Words. 4096 pages	16384 words
2M Words. 8192 pages	32768 words

Fig 5-6 Page Hash Table Sizes

The page age process, refered 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 Virtual Page Data Table entry swap status 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 commiting 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 that no disk operations are required. The status is returned to normal to reflect the fact that the page has been recently referenced.

5.14 Virtual Page Management

This section discusses a technique for dealing with the management of virtual pages. The issues involved are related to requirements for performance improvement and increased functionality in the Explorer virtual memory system. The increased functionality includes the ability to map the paging related backing store across different bands, perhaps on different physical units.

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

5.14.0.1 Page Hash Table

The size of the page hash table is related to the size of physical memory. Since a hashing 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 *Fig. 5-6* 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 show in *Fig. 5-7*.

The Virtual Page Number field is the hash key indicating the virtual page that this entry describes. The field corresponds to the virtual address field in the VMA register for convenience.

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 5-5* 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.

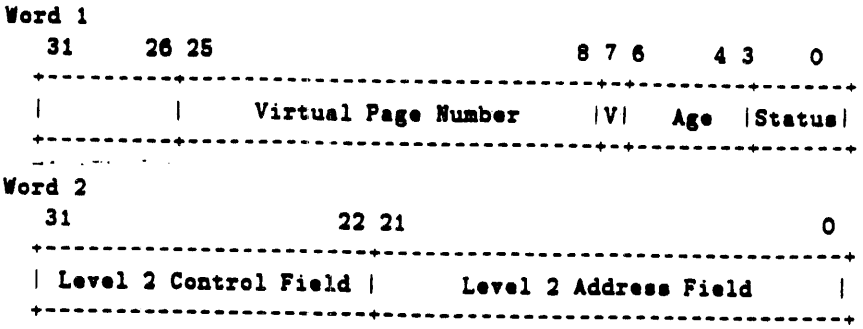


Fig. 5-7 Page Hash Table Entry Format

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, 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-

Table 5-5 Page Hash Table: Swap Status Codes

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 rehash, 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.

5.14.1 Disk Page Mapping Scheme

This section describes the mechanism by which given a virtual page number we may find the disk address assigned to it.

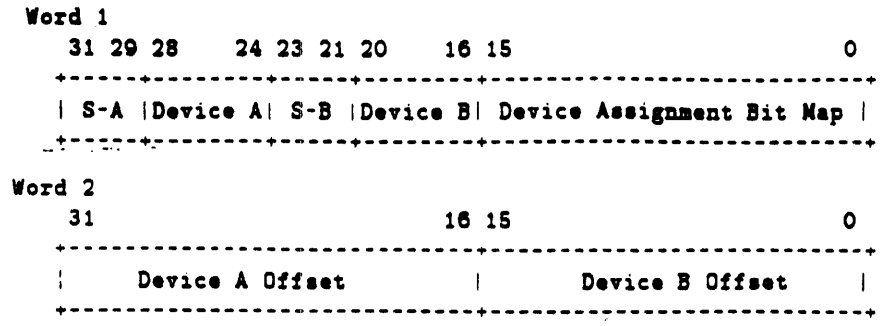


Fig. 5-8 Disk Page Mapping Table

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 assigned.
Status 4 - 7:	Unused.

Fig. 5-9 Device Status Codes

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 not practical to have a one-to-one correspondence between virtual pages and DPMT entries. Therfore 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 will be indexed by the most significant 13 bits of the virtual page number. Disk space is allocated in blocks of sixteen pages. Each block corresponds to one physical page and is 1024 bytes (256 words).

Each entry of the DPMT specifies one or 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 *Fig. 5-8*.

The fields S-A and S-B are the device assignment status fields for device A and device B respectively. The values and for these fields are described in *Fig. 5-9*.

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 we may have several paging bands on a single device or on several devices.

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.

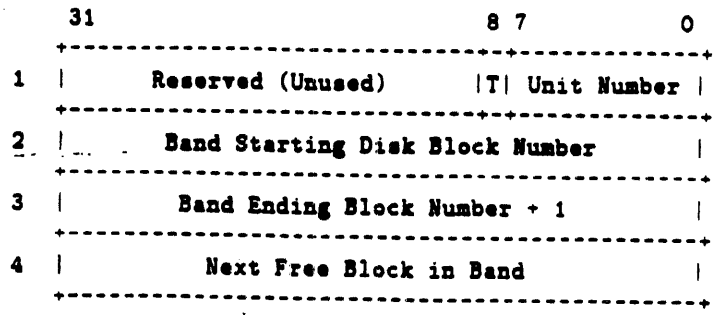


Fig. 5-10 Logical Page Device Information Block

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).

5.14.1.1 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. (See pre-paging)

2. Find memory page frames for the pages being read by scanning the PPD table. If a page is dirty then it must be written to the swap partition.

3. Issue the read operation.

When a virtual page has never been 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.

5.14.1.2 Logical Paging Devices

A logical paging device defines a contiguous set of 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 *Fig. 5-10*, the information maintained is as follows:

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.

2. Starting block number of the page band. Indicates the block number of the first block that may used in this page band.

3. End of page band. This word indicates the block number of the first block that is outside of the band. (Perhaps this should point to the last block actually inside the band.)

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 sequentially. Initially word 3 of the device information block is the same as the starting block number. When a disk block allocation is requested word 3 is checked to see that it less than word 2, the end of the page band. If the value is OK, it is returned as the allocated block. Word 3 is then incremented reflecting the fact that this block has been assigned.

6. Internal Storage Formats

It takes all types.

truism. spouted by Phil Mueller

Here the formats of the data objects in the Lisp Machine system will be described. This agrees with SYS:QCOM version ≈ 595.

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

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

Since some objects require more storage than the fixed size, one of these objects is represented as a pointer to a block of storage where its state is stored in memory. Objects that can be represented completely in a storage cell are called INUM (for "immediate number") types. Other types are called pointer types. This pointer based organization yields very flexible data structuring as will be explained as the structuring data types are explained.

6.1 Q Format

The storage cell in the Lisp Machine is called a Q or quantum. The format of a Q is shown in *Fig. 6-1*. Every LISP object as a Q.

A Q is also sometimes refered to as boxed storage. Words which are not in Q format are refered to as unboxed storage. Unboxed storage can only be interpreted in some special context. For example, the macroinstructions for a function are stored as part of a FEF structure.

The fields of a Q are the **CDR CODE**, **DATA TYPE**, and **POINTER** fields. These are explained below.

6.1.1 CDR Code

The **CDR CODE** field is a 2-bit field in the Q. If this Q is in a list structured region or in a structure that is treated as a list (a stack list or ART-Q-LIST array),[1] 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 pointers form of **CONS** used in most Lisps. **CDR ERROR** means that it is an error to take the **CDR** of this

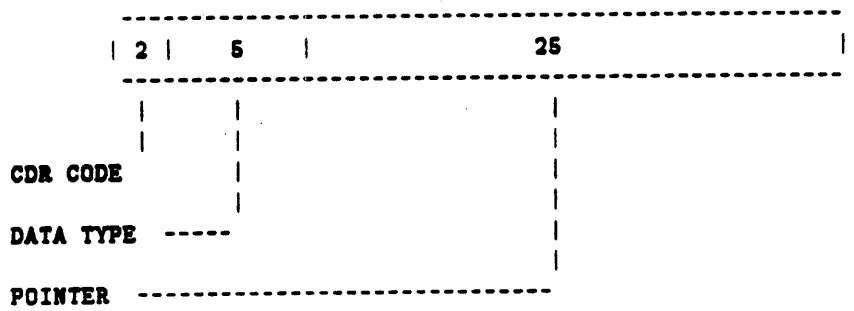


Fig. 6-1 Q Format

[1] See section on areas or something.

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

Table 6-1 CDR Codes

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. CDR NEXT indicates that the CDR is at *this address* − *1*.

The codes are set up this way so that a list of N elements can be stored in N consecutive Q's using CDR NEXT and CDR NIL. This results in high storage density. The functions APPEND and LIST form these compact lists. CONS and friends as of now 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).

6.1.2 Data Type

The 5-bit DATA TYPE field determines the data type of the Q. The datatypes are shown below. Note that some of the datatypes are useful mostly for their meaning in "function context".[2]

6.1.2.1 Data type 0 - DTP-TRAP

Any attempt to reference this cell will cause a trap. This is mostly for error checking. Cannot be "in the machine".[3]

6.1.2.2 Data type 1 - DTP-NULL

This datatype is used for various things to mean "nothing". For example, an unbound atom has one of these as its value. The pointer field points back at the atom. for ease in debugging. Cannot be "in the machine".

6.1.2.3 Data type 2 - DTP-FREE

This cell is free unallocated storage. The pointer field of this word has its own virtual address. The user should not see this. Cannot be "in the machine".

6.1.2.4 Data type 3 - DTP-SYMBOL

This is a non-numeric atom. The pointer points to a five Q "symbol header". Allowed to be "in the machine".

6.1.2.5 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). Allowed to be "in the machine".

6.1.2.6 Data type 5 - DTP-FIX

A FIXNUM (fixed point number). The pointer is not really a pointer; it is the actual value of the number, so FIX numbers with the same value will always be EQ, unlike PDP-10 Maclisp. Of course, allowed to be "in the machine".

6.1.2.7 Data type 6 - DTP-EXTENDED-NUMBER

Any type of number other than a FIXnum or small flonum. It points to a DTP-HEADER word (see below). Allowed to be "in the machine".

---

[2] See section on funcalling.

[3] See section on in the machine.

6.1.2.8 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. Cannot be "in the machine".

6.1.2.9 Data type 8 - DTP-GC-FORWARD

The forwarding address left behind 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. The address that got to this object is altered to point to the object's new location (the **GC-FORWARD** is "snapped out"). Cannot be "in the machine".

6.1.2.10 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. See section on closure. Cannot be "in the machine".

6.1.2.11 Data type 10 - DTP-ONE-Q-FORWARD

This is a simple kind of invisible pointer used to "invisiblize" 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. Cannot be "in the machine".

6.1.2.12 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.

6.1.2.13 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)

6.1.2.14 Data type 13 - DTP-LOCATIVE

This is a pointer to a single cell of memory, which is not "invisible" to anything. 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. Allowed to be "in the machine".

6.1.2.15 Data type 14 - DTP-LIST

The pointer points to a cons cell. The format of storage of LISTs is explained above, under **CDR CODE**. Allowed to be "in the machine".

6.1.2.16 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. Allowed to be "in the machine".

6.1.2.17 Data type 16 - DTP-FEF-POINTER

Points to a macro-compiled function. It points to a word of **DTP-FEF-HEADER**, which is the first of a block of words at least 8 long. Allowed to be "in the machine".

6.1.2.18 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. Allowed to be "in the machine".

6.1.2.19 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. Before 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. Cannot be "in the machine".

6.1.2.20 Data type 19 - DTP-STACK-GROUP

This is a stack group. The word just like an **ARRAY-POINTER**. it points to an array header of array type ART-STACK-GROUP-HEAD. The format of a stack group is explained in section on multiprocessing Allowed to be "in the machine".

6.1.2.21 Data type 20 - DTP-CLOSURE

This is a closure. It points to a block of storage $2N + 1$ long, where N is the number of cells closed over. For details see documentation of closures. Allowed to be "in the machine".

6.1.2.22 Data type 21 - DTP-SMALL-FLONUM

A small floating point number. The pointer is not really a pointer; instead it is a 24 *** 25? not yet! *** bit floating point number. There is a 7 bit excess-100 exponent (10^{-19} to 10^{+19}, approximately) and and a 17 bit 2's complement normalized mantissa (5 digits, approximately). Allowed to be "in the machine".

6.1.2.23 Data type 22 - DTP-SELECT-METHOD

Method table for a message handling functional object. These aren't used much. *** No further documentation. There needs to be. *** Allowed to be "in the machine".

6.1.2.24 Data type 23 - DTP-INSTANCE

A flavor instance. Points to a word of **DTP-INSTANCE-HEADER**. See section on flavors for more information. Allowed to be "in the machine".

6.1.2.25 Data type 24 - DTP-INSTANCE-HEADER

The header word of an **INSTANCE**. Pointed to by a **DTP-INSTANCE**. See section on flavors for more information. Cannot be "in the machine".

6.1.2.26 Data type 25 - DTP-ENTITY

A closure with also binds **SELF** if it is called (similar to **DTP-INSTANCE**). This type is to be considered obsolete. Use at your own risk. Allowed to be "in the machine".

6.1.2.27 Data type 26 - DTP-STACK-CLOSURE

A funarg. Can only be stored shallower on the stack than where it is. If stored anywhere else, must be copied out of the stack. *** More info is required on this ***

6.1.2.28 Data type 27 - DTP-SELF-REF-POINTER

Special form for referencing instance variables. Transporter notices and accesses self via self-mapping table or not. Also used to monitor variables. See Self Reference Pointer, below.

6.1.2.29 Data type 28 - DTP-CHARACTER

A character. Primarily for Common Lisp compatability (for now). Can be used in arithmetic like a FIXNUM. *** does it have fields for font and "bits"? What is layout? ***

6.1.2.30 Data type 29 - DTP-FEP-HEADER

Header for a FEF.

6.1.2.31 Data types 30 through 31

Data types 30 through 31 are unused. They are treated the same as **DTP-TRAP**. They are not allowed to be "in the machine".

TI Internal Data

CODE	SYMBOL	*Meaning*
0	Q-HEADER-TYPE-ERROR	Bad header. not used
1	Q-HEADER-TYPE-UNUSED-1	not used
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-3	not used
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 BIGNUM's

Table 6-2 Header Type Codes

6.1.3 Pointer

POINTER (25 bits) - The use is determined by the datatype of the Q. Usually it points to some other object in memory. Sometimes it just contains miscellaneous data as described for INUM types above.

6.2 Structure Headers

A word of **DTP-HEADER** is the first word of a number of structure types. Other kinds of structures have a special data type for their headers. When **DTP-HEADER** is used, the type of the structure is indicated by the **HEADER TYPE** field. Header types are shown in *Table 6-2*.

6.3 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 the indirection is specified by the reading instruction, instead it 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 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**. Even **DTP-SELF-REFERENCE-POINTER** gives behavior similar to an invisible pointer. These perform additional work beyond following the invisible pointer. Even though an SRP is an INUM type, when one is used for a mapped or unmapped instance variable reference, it indirects the reference to an instance variable, much as an invisible pointer.

6.4 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*.

OFFSET	CELL NAME
0	PRINT-NAME-CELL
1	VALUE-CELL
2	FUNCTION-CELL
3	PROPERTY-CELL
4	PACKAGE-CELL

Table 6-3 Qs of Symbol

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 section on 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 in contains **DTP-NULL**. See Functional Objects below.

The **PROPERTY-CELL** contains the property list. Nothing in the basic system requires that symbols have properties, so this might be **NIL**. On the other hand, many subsystems and features make heavy use of the property list, so it is likey 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 is 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**, etc., 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.

6.5 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.[4] 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

[4] The rank of an array is the number of dimensions it has.

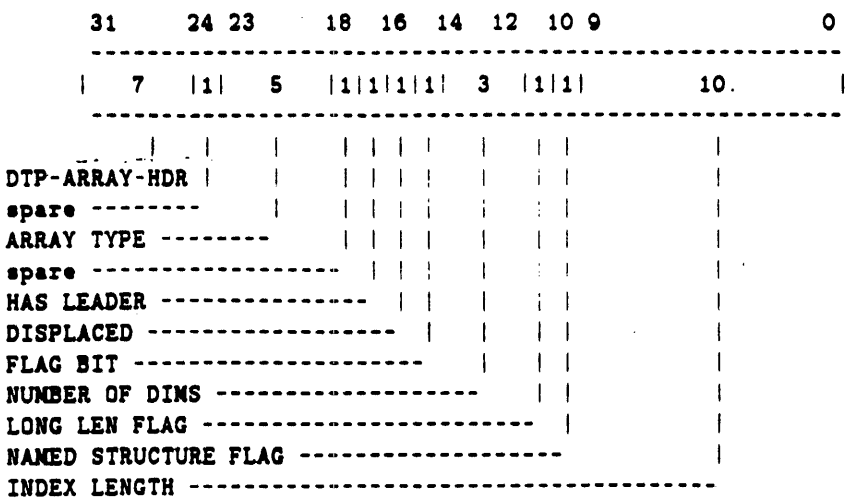


Fig. 6-2 Array Header Word

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 always can 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**. It's pointer field has the format shown in *Fig. 6-2*.

6.5.1 Has Leader

The array may optionally have an array leader which is formed 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 collecter which scan through memory in the usual direction. The storage layout of an array with leader is shown in *Fig. 6-3*.

6.5.2 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[5] (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

[5] See section on areas.

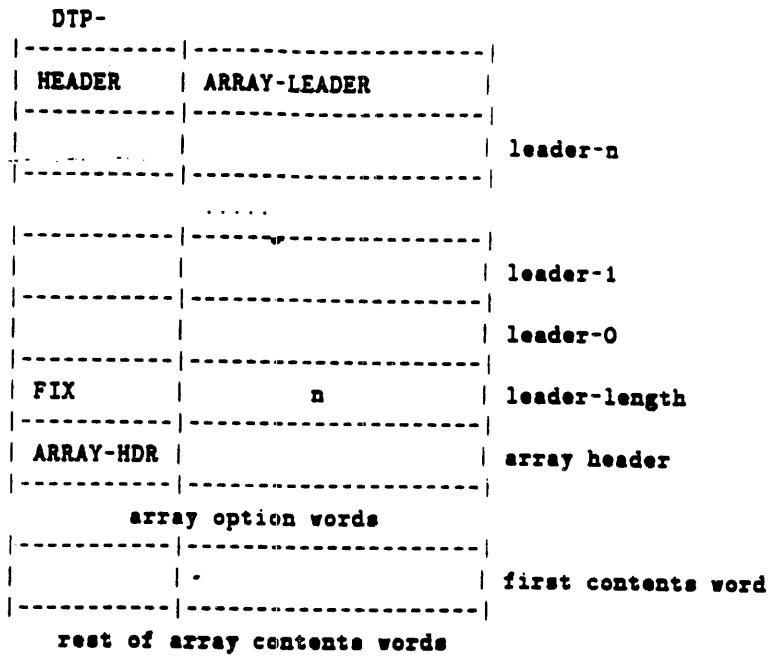


Fig. 6-3 Array with Leader

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 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.

6.5.3 Number Of Dims

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

6.5.4 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.

6.5.5 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

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

Table 6-4 Array Types

that a function to apply. ACTOR like, to perform the primitive. One can see that there is some potential...

6.5.6 Index Length

The **INDEX LENGTH** of an array is the number of items which is the maximum value that the index can take on in a one dimensional array. 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 (see above).

6.5.7 Array Type

The array type indicates the type of the array. The array type indicates how the data should 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 below.

Note: the elements of arrays (those which are smaller than 32 bits) are stored right-to-left (i.e., the first element of an **ART-4B** array would be stored right-justified, including the least significant bit).

6.5.7.1 ART-ERROR

This is not used and it is always an error to access an array of this type. This is mainly to insure the robustness of the implementation. Bad array header words have a chance of having this as their array type.

6.5.7.2 ART-1B

This is an array of 1-bit numbers. Accessing this array type always returns either FIXNUM 0 or FIXNUM 1. Storing an even FIXNUM into this type of array sets the cell to 0, odd to 1. 32 cells are stored per word.

6.5.7.3 ART-2B

This is an array of 2-bit numbers. Accessing this array type always returns a FIXNUM from 0 to 3. Storing a FIXNUM into this type of array sets the cell to the least significant 2 bits of the FIXNUM. ART-2B arrays are stored with 16 cells per word.

6.5.7.4 ART-4B

This is an array of 4-bit numbers, analogous to ART-2B arrays. ART-4B arrays are stored with 8 cells per word.

6.5.7.5 ART-8B

This is an array of 8-bit numbers, analogous to ART-2B arrays. ART-8B arrays are stored with 4 cells per word.

6.5.7.6 ART-16B

This is an array of 16-bit numbers, analogous to ART-2B arrays. ART-16B arrays are stored with 2 cells per word.

6.5.7.7 ART-32B

ART-32B arrays have 32 bits per element. Since fixnums only have 25 bits anyway, these are the same as ART-Q arrays except that they only hold fixnums. They are not compatible with other "bit" array types and generally should not be used.

6.5.7.8 ART-Q

This is the commonly used type of array. Each cell holds a Lisp object of any type.

6.5.7.9 ART-Q-LIST

This is similar to ART-Q in that its elements may any Lisp object. The difference is that the ART-Q-LIST array doubles as a list: there is a miscellaneous instruction (G-L-P) which will return the elements of the array in a list. Furthmore, the elements of the list and the cells of the array share storage so that RPLACA'ing an element of the list will change the contents of the coresponding array cell. Such lists cannot be RPLACD'ed: an attempt to RPLACD an element of an ART-Q-LIST array will get a **WRONG-REPRESENTATION-TYPE** error.

6.5.7.10 ART-STRING

This array is a character string. This type acts similarly to the ART-8B. its elements must be fixnums or characters, of which only the least significant eight bits are stored. However, many important system functions treat ART-STRING arrays very differently from the other kinds of arrays. These arrays are usually called strings. See also ART-FAT-STRING.

6.5.7.11 ART-STACK-GROUP-HEAD

Stored the same way as an ART-Q array. This is pointed at by DTP-STACK-GROUP. The format of a stack group is explained in section on multiprocessing.

6.5.7.12 ART-SPECIAL-PDL

This is the array type used to implement the special binding stack (PDL). Stored the same way as an ART-Q array.

6.5.7.13 ART-HALF-FIX

This is an array of signed 16-bit FIXNUM's. Two cells are stored per word. Note that this differs from ART-16B in that the 16-bit numbers are sign extended to make FIXNUM's when read.

6.5.7.14 ART-REG-PDL

This is the array type used to implement the regular (main) stack (PDL). Stored the same way as an ART-Q array.

6.5.7.15 ART-FLOAT

This is an array of flonums. When storing into such an array the value (any kind of number) will be converted to a flonum. The advantage of storing flonums in an ART-FLOAT array rather than an ART-Q array is that the numbers in an ART-FLOAT array are not true Lisp objects. Instead the array remembers the numerical value, and when it is read, creates a flonum Lisp object to hold the value.

Because the system does special storage management for extended numbers that are intermediate results, the use of ART-FLOAT arrays can save a lot of work for the garbage collector and hence greatly increase performance. ART-FLOAT arrays also provide a locality of reference advantage over ART-Q arrays containing flonums, since the flonums are contained in the array rather than being separate objects probably on different pages of memory.

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)

Table 6-5 Self-Reference Pointer Fields

6.5.7.16 ART-FPS-FLOAT

This is also an array of flonums. The internal format of this array is compatible with the PDP-11/VAX single-precision floating-point format. The primary purpose of this array type is to interface with the FPS array processor, which can transfer data directly in and out of such an array.

6.5.7.17 ART-FAT-STRING

This is a character string like ART-STRING, but with wider characters, containing 16 bits rather than 8 bits. The extra bits are ignored by string operations, such as comparison, on these strings; typically they hold font information. Just as ART-STRING is similar to ART-8B, ART-FAT-STRING is similar to ART-16B.

6.5.7.18 ART-COMPLEX-FLOAT

This is an array whose elements are numbers whose real and imaginary parts are both floating point numbers. If a non-floating-point number is stored into the array, its real and imaginary parts are converted to floating point. This provides maximum advantage in garbage collection if all the elements stored into the array are numbers with floating point real and imaginary parts.

6.5.7.19 ART-COMPLEX

This is an array whose elements are arbitrary numbers which may be complex numbers.[6] As compared with an ordinary ART-Q array, ART-COMPLEX provides an advantage in garbage collection similar to what ART-FLOAT provides for floating point numbers.

6.5.7.20 ART-COMPLEX-FPS-FLOAT

*** combined ART-COMPLEX-FLOAT and ART-FPS-FLOAT ***

6.6 Self Reference 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, it 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.

The RELOCATE-FLAG, when set, says to use the 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 is 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.

---

[6] Other numeric arrays can only hold real numbers.

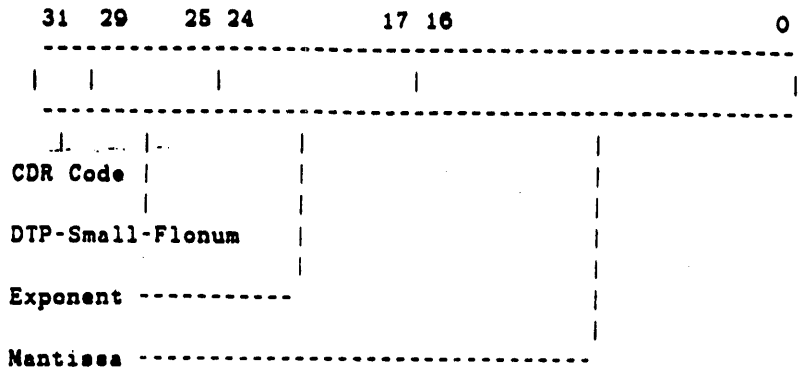


Fig. 6-4 Small Flonum Format

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

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

6.7 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 datatypes. It always contains boxed (typed) data.

6.8 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 section on function calling.

6.9 Floating Point Formats

There are several floating point formats supported in the Lisp Machine. 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 is also a three word internal representation of a flonum.

The format of a small flonum is shown in *Fig. 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 *Fig. 6-5*.

The Flonum Header has the format shown in *Fig. 6-6*.

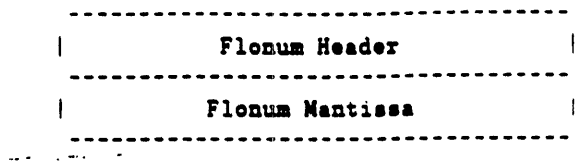


Fig. 6-5 Flonum Structure Format

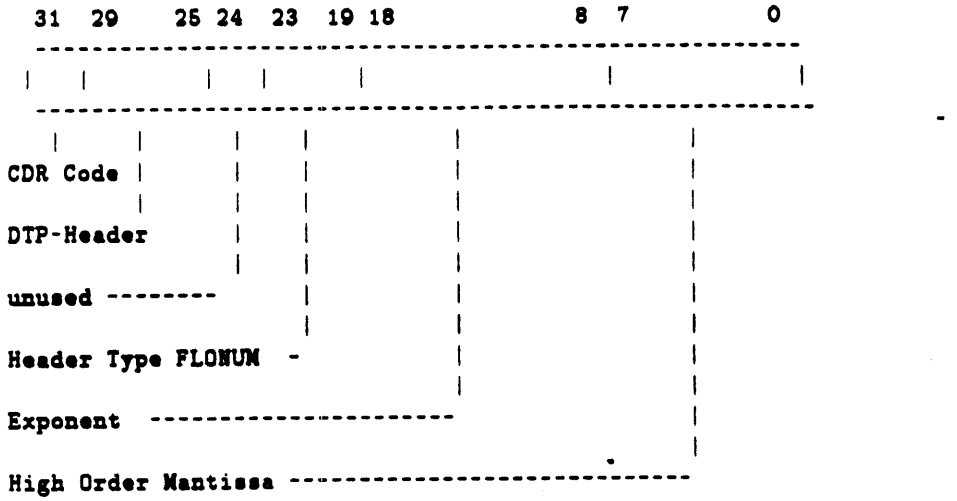


Fig. 6-6 Flonum Header Format

The exponent is stored in excess-4000 (octal) form. The high order mantissa has the most significant eight bits of a twos 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 *Fig. 6-7*.

6.10 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 *Fig. 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 *Fig. 6-9*.

The LENGTH field gives the length of the bignum in words; this is the length of the bignum structure minus one.

Each word of the bignum has the format shown in *Fig. 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. Storing the high order bit as 0 allows for easy multiprecision arithmetic since the hardware does not have carry save.

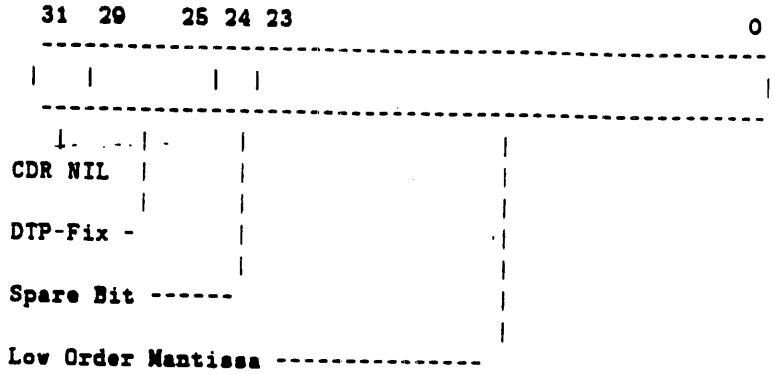


Fig. 6-7 Flonum Mantissa Format

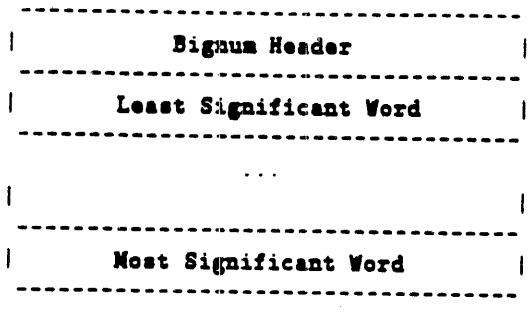


Fig. 6-8 Bignum Structure

6.11 Special PDL

The Special PDL[7] 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.

The Special PDL also serves some other functions. When a *micro-to-macro* call is made, the "micro-PDL" of the machine is stored there (this is needed because the hardware micro-PDL is of a small fixed size).

The Special PDL is block oriented. The blocks are delimited by setting the **SPECPDL-BLOCK-START-FLAG** in the first binding made in a block. The data type of the top word (last pushed) of a block determines what kind of block this is, as shown in *Table 6-6*.

SPECPDL-BOCK-START-FLAG and **SPECPDL-CLOSURE-BINDING** are stored in the CDR-Code field of Q's on the Special PDL. The CDR-Code is not otherwise used on the Special PDL. **SPECPDL-CLOSURE-BINDING** indicates that this binding was made "before" entering the function (ei. by closure binding, or by the binding of **SELF** for a method). **SPECPDL-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

[7] also known as the linear binding PDL (LBP)

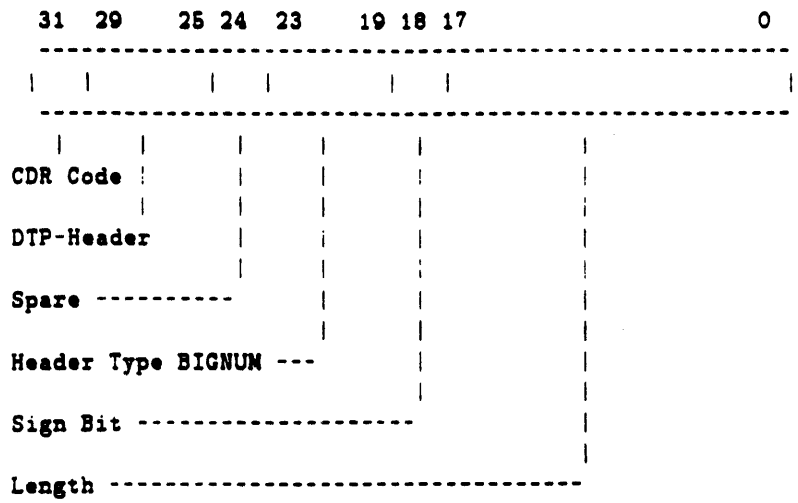


Fig. 6-9 Bignum Header Format

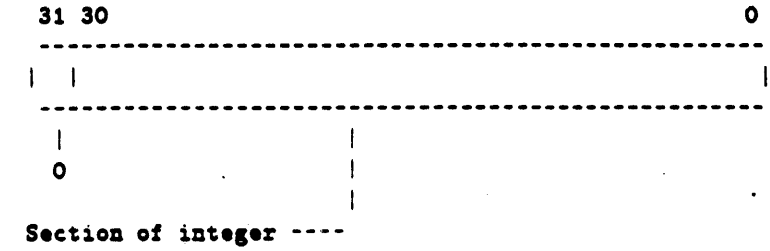


Fig. 6-10 Flonum Mantissa Format

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 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 (see calling conventions) on the regular PDL in the exit state word (see section on PDL formats). It is restored when returning to a call block. This assured 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 even 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. This is all very hairy; anyone who is very, very interested is invited to read UCONS and/or LMI. *** figure out and explain this hair ***

The Special PDL is pointed to by the location **SG-SPECIAL-PDL-POINTER** in the stack group and by **A-QLBNDP** when the stack group is executing on processor. There is an area devoted to storing the Special PDL's called **LINEAR-BIND-PDL-AREA**.

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.

Table 6-6 Special PDL Block Type

6.12 CLOSURE Formats

Lisp Machine LISP, like MACLISP, uses shallow binding; each symbol contains a " value cell" which contains its current binding. For this discussion, the value cell of a symbol will be known as the "internal value cell". An advantage of shallow binding is that the time needed to access the value of a symbol is a very small constant: only that of a single memory reference. When a symbol is bound, a pointer to its internal value cell and its current binding are pushed on the binding PDL.

Some implementations of LISP us deep binding, in which accessing a variable requires an **ASSOC**, and takes time proportional to the number of bindings on the A-list. In Lisp Machine LISP, we desired to keep the short, constant access time of shallow binding but still be able to deal with "funargs" where a binding environment is remembered with a function.

To acomplish this, we introduce a new data type, **DTP-EXTERNAL-VALUE-CELL-POINTER**. This object is treated in the usual way by the **BIND** and **UNBIND** operations, but is treated as an "invisible pointer" by **SET** and **SYMEVAL**. (**SYMEVAL** is the primitive function for accessing the value of a symbol, and **SET** is the function for updating the value of a symbol). The word pointed to by the **DTP-EXTERNAL-VALUE-CELL-POINTER** is called the "external value cell". Thus, **SET** and **SYMEVAL** operate on the external value cell, while **BIND** and **UNBIND** refer to the internal value cell.

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** **CDR**'s 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 $2N + 1$ 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 pointer to the external value cell. Finally, **CLOSURE** returns an object of datatype **DTP-CLOSURE** which points at the block. This is the closure itself.

When a closure is invoked as a function, the first thing that happens is that the saved environment is restored; that is, the current contents of the internal value cells of — are saved on the binding PDL, and the **DTP-EXTERNAL-VALUE-CELL-POINTER**'s are restored from the closure. Then, the function is invoked with the same arguments as were passed to the closure.

Here is another example, in which the closure feature is used to solve a problem presented in "LAMBDA - The Ultimate Imperative" [Steele 77?]. The problem is to write a function called **GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE** which is to take one argument, which is the factor

by which the tolerance is to be increased. You are given a function **SQRT** which makes a free reference to **EPSILON**, which is the tolerance it demands of the trial solution.

```
(DEFUN GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE (FACTOR)
    (CLOSURE '(FACTOR)
     (FUNCTION
      (LAMBDA (X)
      ((LAMBDA (EPSILON) (SQRT X))
       (* EPSILON FACTOR))))))
```

7. Storage Management

This doesn't look like Kansas!

Dorothy in *The Wizard of OZ*

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

Storage allocation for Lisp objects and structures is implemented on top of a large uniform address space provided by the Virtual Memory System (see section on virtual memory). 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 can not 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 in two levels into smaller pieces. The first level breaks the address space into areas. An area is a collection of regions, the second level of storage management. Areas are created by explicit commands. Creation merely defines an abstract entity called an area. Actual allocation of the virtual address space occurs when storage space is requested. The virtual address space is assigned to a region. The region is then divided up into blocks as per storage allocation requests.

7.1 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 1 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 tables.

Five kinds of information about areas are kept in the Area Descriptor Table. The five word entries contain this information:

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 in 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.

Actually, the Area Descriptor Table doesn't exist. It is implemented 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 the entry size.

The set of attributes that an area has 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 described in detail in section on section on regions.

Areas can be created by user commands. The area in which consing (storage allocation) 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 performance. Also virtual memory paging can be controlled on an area basis.

The list of regions associated with an area is a linked list. The last element in the list is the area number with the sign bit set. The free regions are also in a linked list. This is needed because garbage collection frees a set of regions when it "flips", therefore the set of free regions grows and shrinks dynamically.

7.2 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. 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 each region. Each region has a 6 word entry in the table. Like the (Area Descriptor Table), the (Region Descriptor Table) is 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 length of virtual address space allocated to this region in words.

Region Bits: Specifies the properties of this region. See *Fig. 7-1*.

Region Free Pointer: Offset into this region of the next free word to be allocated. The virtual address of the next free word in this region is the region origin plus the region free pointer.

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

Region List Thread: region number of the next region in the linked list.

Regions can store two types of data. The first type is List data. The second type is structure data. 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*. *** more here ***

The region bits word defines the properties of the region. The fields within the region bits word are shown in *Fig. 7-1*.

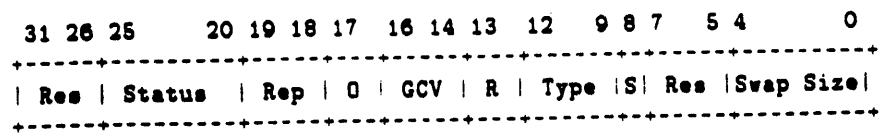
7.3 Standard Areas

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 of these 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 (QCOM) specifies these areas 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 QCOM file.

The fixed areas each contain a single region. 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, the 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:

TI Internal Data



Res:	Reserved, unused.
Status:	Access and status bits to be used in the level 2 memory map. See virtual memory
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 s
GCV:	GC Volatility for this region.
R:	Reserved, unused.
Type:	Space Type.
S:	Savenger Enable. Value: 1 = Savenger can touch this area.
Res:	Reserved, unused.
Swap Size:	Number of pages the Virtual Memory System should try to swap at a time

Fig. 7-1 Regions Bits Area Entry Description

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

Table 7-1 Space Type Codes

Resident Symbol Area (wired): contains the important symbols T and NIL.

System Communication Area (wired): This area contains various groups of words used by I/O routines and systems utilities. See section section on SYSCOM.

Scratch Pad Init Area (wired, read only):

Micro Code Symbol Area (wired, read only): Contains the microcode entry points for the miscops.

Region Origin Area (wired): Contains the starting address of 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 free pointer for each region. index by region number.

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

Disk Page Map Area (wired): Contains the Disk Page Map Table for the Virtual Memory System.

Page Table Area (wired): Contains the Page Hash Table for the Virtual Memory System.

Physical Page Data Area (wired): Contains the Physical Page Data Table for the Virutal Memory System.

Address Space Map Area (wired): Contains the Address Space Map for Storage Allocation and the Virtual Memory System.

Region GC Pointer Area (fixed): Contains the GC pointer 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):

Area Region Size Area (fixed):

Area Maximum Size (fixed):

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): Number consing area.

Microcode Entry Area (fixed):

Microcode Entry Name Area (fixed):

Microcode Entry Args Info Area (fixed):

Microcde Entry Max PDL Usage (fixed):

Microcode Entry Arglist Area (fixed):

Microcode Symbol Name Area (fixed, read only):

Linear PDL Area (fixed):

Linear Bind PDL Area (fixed):

Init List Area (fixed, read only):

TI Internal Data

(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

Fig. 7-2 Map of Systems Communication Area

Working Storage Area: Default (general purpose) consing 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 (static, read only):

PDL Area:

FASL Table Area:

FASL Temp Area:

7.4 Systems Communication Area

The Systems Communication Area contains miscellaneous words that are needed to be basic, i.e., not rely on the rest of the machine operating. The Systems Communication Area is wired and at the fixed address of 400 (octal).

See figure *Fig. 7-2*.

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 bufferred devices. This list is searched when a interrupt occurs and there is not 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

7.5 Address Space Map Area

The Address Space Map Area contains the address space map. This table is 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 alloated 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.

7.6 Extra PDL Area

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". 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 a pointer to the copy is written.

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.)

7.7 Linear PDL Area

The Linear PDL Area contains the Linear PDL (Push Down List i.e., stack) for each process. The Linear PDL (usually just called PDL) is the run time stack for the process. The currently executing process will have the top part of its PDL cached in the PDL Buffer.

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

7.8 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 the process.

7.9 Working Storage Area

The Working Storage Area is the default cons area, that is most objects created by the user are created in this area.

7.10 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.

7.11 CONS

*** explain how cons finds a region in the area and makes an allocation there ***

8. Garbage Collection

**** something from Oscar on Sesame St. ****

Oscar on Sesame St.

This chapter explains how the Lisp system recovers storage that is no longer in use. The collection algorithm used is based on the famous Baker algorithm.[1]

8.1 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 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. Certain values that are "active" may not be stored "inside the machine". Since there are no active values inside the machine, tests for active values are not needed to access data inside the machine.

Clearly, this efficiency is important for certain machine registers and (maybe less clearly) for the PDL buffer. These places are declared to be inside the machine. The certain registers include the lettered and numbered registers (eg. M-A, and M-1) and *** which other registers ***.

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

Many of the data types are not allowed "inside the machine". These include the illegal data types (DTP-TRAP and DTP-NULL), the header data types (DTP-HEADER, DTP-SYMBOL-HEADER, DTP-ARRAY-HEADER, DTP-FEF-HEADER, and DTP-INSTANCE-HEADER), the forwarding types (DTP-ONE-Q-FORWARD, DTP-GC-FORWARD, DTP-HEADER-FORWARD, DTP-BODY-FORWARD, and DTP-EXTERNAL-VALUE-CELL-POINTER) and some special types (DTP-SELF-REFERENCE-POINTER). In addition to data types not allowed inside the machine, pointer type objects that point to oldspace are also not allowed inside the machine.

Attempts to read something into the machine that is not allowed cause some action to take place. The specific action depends on the type of the object. In any case, the prohibited object is not allowed to pass into the machine.

8.2 The Read Barrier

The read barrier is implemented by 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 made based on the data type of the Lisp object read from memory and the **OLDSPACE** property of the region that the object points to, if it is a pointer type object.

8.3 The Write Barrier

The write barrier is implemented by GC Write Test. Mainly it detects writing of pointers to the extra PDL. Every Lisp object written is tested with GC write test.

---

[1] See *Baker78* for a description of this algorithm.

8.4 Incremental GC

Should be familiar with Baker's paper. Also reference Hewitt's paper. Need to understand the basics of storage allocation, previous chapter.

The goal of incremental GC is to perform garbage collection without any long and embarrassing pauses. Instead, some garbage collection is done whenever storage is consed. During collection, storage that is being collected is divided in old space and other spaces that will be discussed below. Old space is dynamically allocated storage that is in the process of being purged of garbage. The goal of this garbage collection cycle is to copy everything useful out of old space and to reclaim the storage used by old space.

Of course, if every object is copied out of oldspace, every useful object will be preserved and when oldspace is reclaimed, there will be no net savings. The goal is to copy only the useful objects out of old space. It is important to guarantee that there is no useful object remaining in oldspace when it is reclaimed. The scavenging algorithm assures this.

New space is where new objects are allocated. Nothing in new space is allowed to refer to old space. Objects that are copied from old space are moved to copy space. 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.

8.4.1 Scavenging

Scavenging is the operation of "cleaning" the other spaces of references to old space. Words are examined and if a pointer to old space is found, the object is copied out of oldspace into copy space. When none of the other spaces contain pointers into old space, there are no more useful objects remaining in old space and it may be reclaimed.

Notice that new space contains no pointers to old space so there is no need to scavenge it. Static space must be scavenged.

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. 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 modify any structure (the tranporter does not allow it). 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 refered 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 Transporting

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 as described above). This means that objects that are "in the machine" may not point to old space. It is not possible therefore to store a pointer to old space since it must first be read into the machine and transported.[2]

8.4.4 Areas

The description above is incomplete as it does not take into account the division of the address space into Areas. Every object is in some region that is part of some 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. New allocation in an area goes into a new space region.

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 in which area an object is stored.

8.4.5 Flipping

The remaining phase of incremental GC that has not been 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 Ignore

When garbage collection begins, all regions are marked as either old space or static space. Garbage collection then proceeds to copy useful data out of regions designated as old space into regions designated as copy space. This copying action is known as transporting. All new storage allocation is done in regions designated as new space.

The transporter does not copy an object from one area into another area. That is, objects in an old space region are copied to a copy space region in the same area.

Transporting occurs whenever a pointer into old space is read into the machine. Thus, all of the pointers in the machine point to either copy space or new space. Since all of the pointers written into new space come from in the machine, new space can not contain any pointers into old space.

When objects are copied by the transporter, just the top level of the object is copied. Thus, objects in copy space may contain pointers into old space. The scavenger goes through all of copy space looking for pointers into old space. When the scavenger finds a pointer into old space it copies it into copy space using the transporter. The rate at which the scavenger scavenges copy space is proportional to the **CONS** rate. When all of the copy space has been scavenaged then there are no more pointers to old space, i.e., there is no useful information in old space and old space can now be reused.

At some point (when?) a flip occurs and all new space and copy space regions are designated old space *** this is a lie *** 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.

*** Previous attempt. ***

---

[2] Actually can be done with **%MAKE-POINTER** and a fixnum for the pointer argument. Be extremely careful when using **%MAKE-POINTER**.

Garbage collection is performed by copying useful data to another place in memory. This occurs on a per area basis. i.e.. each area has it's own regions to which useful data items are copied. These regions are designated as copy space. When storage is allocated in an area it is allocated in a region designated as new space. Whenever a reference is made to storage that is not in new space or copy space it must be copied into copy space. This copying action is known as transporting.

Objects that have been transported may also contain references to old space within them. All of the data items in old space that are referenced must be transported to copy space before a garbage collection pass may be deemed complete. However, all these references can not be copied at the time the first object is transported as this would require enormous computational power and this method of garbage collection would degenerate into a *stop and collect* scheme rather than the incremental scheme described here.

Instead. a process known as scavenging scans all copy space regions at a **CONS** related speed, that is. scavenging occurs at a rate related to the rate at which storage is allocated. When scavenging is complete no references to old space exist. This means that no references exist to any items contained in old space. i.e.. this space is garbage and may be used over.

When the scavenger has completed a pass storage is reclaimed by a process know as flipping. Flipping means that all old space regions are marked free and all new space and copy space regions are now marked as old space. Now the transport/scavenge process starts over.

9. Function Calling

You never call me

the Judys

This chapter explains Lisp functions and how they are called in Explorer System. There are many kinds of Lisp functions ...

9.1 Functional Objects

There are many kinds of functions in Lisp Machine Lisp. Here each of them is described. 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 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[1], or the first argument to **APPLY**, its **FUNCTION CELL** is likewise examined and the contents considered in functional context.

Functional 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 **COMPILE** or by loading an **XFASL** 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 a 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 functional 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.[2]

9.1.2 DTP-SYMBOL Functions

When a symbol is encountered in a functional 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 fetched and the function interpretation mechanism is reinvoked (tail recursively) to interpret this object in a functional context.

---

[1] See section on macroinstructions.

[2] See section on support vector.

9.1.3 DTP-FEF-POINTER Functions

This function is macro-compiled, so use the Function Entry Frame (FEF) pointed to. Microcode interprets FEF's. 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, either from hand coding or the microcode compiler. A routine in microcode is called for this function.

The 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 microcode function. That entry in the microcode symbol area contains the address of the first microinstruction of the microcode function. If the selected entry in the microcode entry area is not a fixnum, the function is not a currently installed microcode function, and the entry is the functional 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 Funcallable Hash Arrays

If an array (DTP-ARRAY-POINTER) is called that is a named structure and has a leader, leader 0 is ******

9.1.7 DTP-STACK-GROUP Functions

Stack groups can be used as functions. The actual action taken when a stack group is called depends on the state of the called stack group and is described below. 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.[3]

9.1.8 DTP-INSTANCE Functions

An instance is a message receiving object that has both state and a table of message-handling functions, 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 bound and the selected method is applied to the arguments.[4]

9.1.9 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.[5]

9.1.10 DTP-ENTITY Functions

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

9.1.11 DTP-STACK-CLOSURE Functions

Almost just like DTP-CLOSURE. **** what goes here ****

9.1.12 DTP-LOCATIVE Functions

Treated like a list, i.e., invoke the interpreter to deal with it. *** not likely to win ***

---

[3] For more on stack groups, resumption, and transmitted values see section on multiprocessing.

[4] See section on instance.

[5] See section on closure.

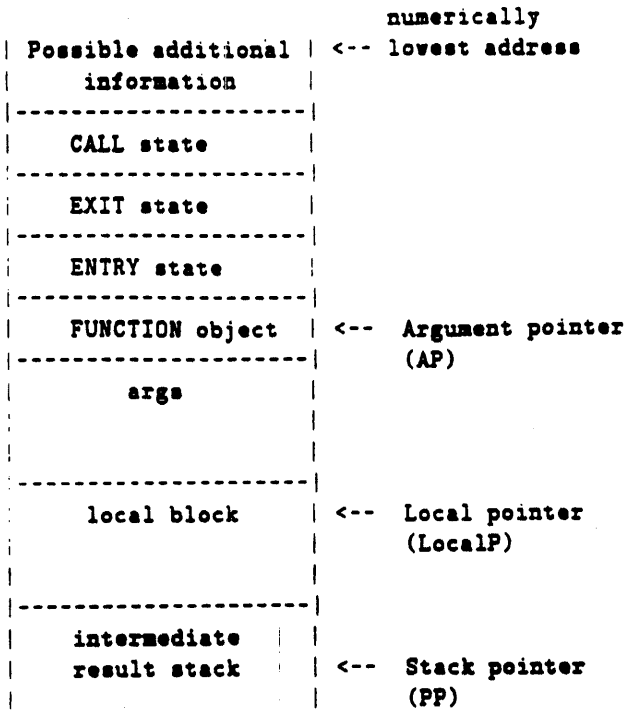


Fig. 9-1 Call Block Layout

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 *Fig. 9-1*.

The "possible additional information" (ADI) is used by certain complex types of calls such as multiple-value calls which need to convey more information. (See ADI, below.)

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., **PROG** and **DO** variables).

9.2.1 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. And 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.

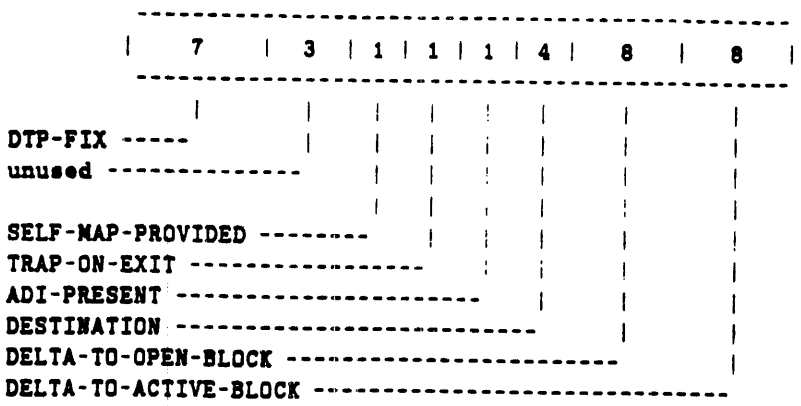


Fig. 9-2 CALL State Bits

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.[6] Also stored in the EXIT state word is the BINDS-PUT-ON-BINDING-PDL bit.[7]

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.

The stack and macro-instruction set are set up so that a function refers to its arguments relative to AP rather than SP. Thus, a function of five arguments refers to its second argument by 2(AP) instead of -3(SP). If fact, all functions refer to their second argument by 2(AP) regardless of the total number of arguments the function takes.

There are also some other useful bits among the CALL state, EXIT state, and ENTRY state words, which are not necessarily related to calling, exiting, or entering; they were basically put wherever they fit. Here are the exact formats of the words:

9.2.1.1 CALL State

The CALL state word (%%LP-CLS-) is shown in *Fig. 9-2*.

DELTA-TO-ACTIVE-BLOCK is the distance back to the previous active block on the PDL. The previous active block is the one that was active when this one was called. Because the size of any block on the PDL is limited to 256 words, this distance fits in the 8-bit field provided.

DELTA-TO-OPEN-BLOCK is the distance back to the previous open block on the PDL. For the same reason as above, this distance also fits in the 8-bit field provided. This field is used

[6] See section on FEF format.

[7] See section on binding PDL format.

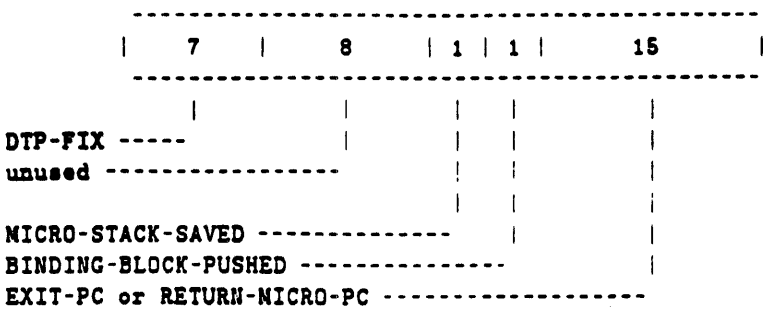


Fig. 9-3 EXIT State Word

to restore the pointer to the innermost open call block when this function is returned from. It is also used when looking for CATCH'es or UNWIND-PROTECT's.

DESTINATION holds a code indicating where to store the value that will be returned. This is the same as the 2-bit **DESTINATION** field of the CALL macroinstruction that built the CALL block. An additional code **D-MICRO** is used by some calls originated by microcode to indicate that the value is to be returned to microcode.

ADI-PRESENT if set indicates that there is additional information (ADI) present. ADI is used for multiple-value calls, Lexpr/Fexpr calls, and certain other unusual calls. The normal CALL instruction never sets this bit.

TRAP-ON-EXIT if set will cause an error before popping the frame. This is used by *** - which *** debugging tools.

SELF-MAP-PROVIDED the function (presumed to be a method) does not need to compute the SELF-MAPPING-TABLE because the caller has done so. This is set by the **%SET-SELF-MAPPING-TABLE** instruction.

9.2.1.2 EXIT State

This is information stored when this frame completes a call. It saves state needed to restore its state on return. The EXIT state word (%%LP-EXS-) is shown in *Fig. 9-3*.

EXIT-PC (15 bits) is saved whenever another block is activated while this is the active block. The EXIT-PC is in halfwords and is relative to the beginning of the FEF.

$$EXIT - PC = LC - (2 \times FEF)$$

The same field is used instead for RETURN-MICRO-PC if this block is for microcode. Only the low 14 bits of the field are used for RETURN-MICRO-PC.

BINDING-BLOCK-PUSHED is used to save the **QBBFL** bit in **M-FLAGS**. **QBBFL** is set if the block does any binding. Just before returning, **QBBFL** is checked and if set, the bindings recorded in the top block of the special PDL are undone.

MICRO-STACK-SAVED is set if a microstack frame has been pushed onto the special PDL.

9.2.1.3 ENTRY State

This is stored when the frame is entered (activated). The ENTRY state word (%%LP-ENS-) is shown in *Fig. 9-4*.

LCTYP is a 1-bit field. This is 1 for a lexpr/fexpr function call.

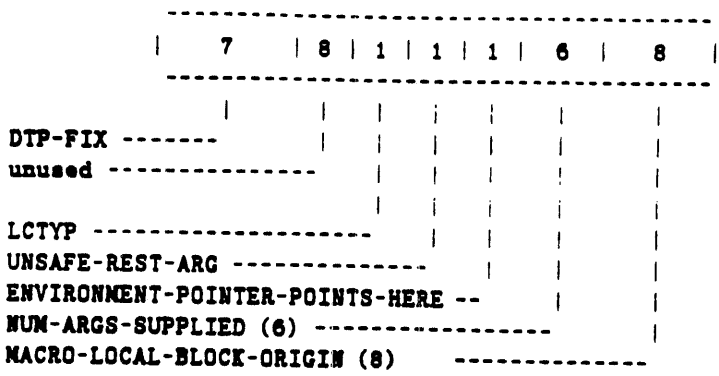


Fig. 9-4 Entry State Word

UNSAFE-REST-ARG is a flag indicating that this frame has a rest arg living on the stack. If tail recursion is attempted on a frame with this flag set, the frame will not be flushed from the PDL.

ENVIRONMENT-POINTER-POINTS-HERE *** what does this mean? ***

NUM-ARGS-SUPPLIED is a 6-bit field indicating the number of args passed to this frame. It is set on function entry. *** do not know that this is actually used by anyone after it is set up *** *** think that this number may be different than the number of arguments received by the function due to receiving some arguments as &REST or &OPTIONAL. *** *** might this be used by the error handler? ***

MACRO-LOCAL-BLOCK-ORIGIN is the distance from the argument pointer (AP) to the beginning of the local block. The locals pointer (LOCALP) can be computed from MACRO-LOCAL-BLOCK-ORIGIN by

$$LOCALP = AP - MACRO - LOCAL - BLOCK - ORIGIN$$

9.2.2 Argument Passing

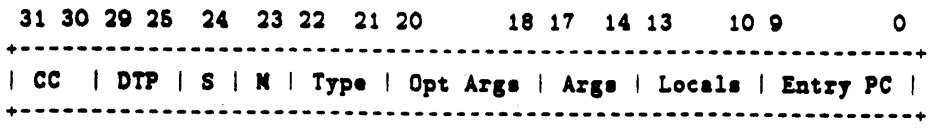
One of the things which must be kept handy is the manner in which the function interprets its arguments.

Function entry for compiled functions supports the complex argument specification supported by Lisp Machine Lisp. There are provisions for storing very complex specifications, such as whether each arg is REQUIRED, OPTIONAL, or REST, whether it is SPECIAL or LOCAL, etc. However, for simple functions a great deal of efficiency would be lost if such a general and complex format were always used. The solution to the problem is that three forms of description are present in a FEF.

9.3 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.[8]

[8] It may prove hard to understand the macrocode instruction set without first understanding the FEF format, and vice-versa; they are very closely related. It is assumed that the reader has read the section on macroinstructions and is at least somewhat familiar with the workings of the Low Cost Lisp.



CC	CDR Code, unused.
DTP	Data Type, DTP-FEF-HEADER.
S	Special Variable Bind Flag 0 means no specials to bind 1 means specials to bind.
M	Get Self Mapping Table 0 means no self mapping 1 means get self mapping table.
Type	Call type. 0: No Locals. 1: Some Locals, 2: Rest Arg (with or without other locals), 3: Long or Complex Call.
Opt Args	Number of optional arguments.
Args	Number of required arguments.
Locals	Number of locals.
Entry PC	entry PC in words.

Fig. 9-5 FEF Header Fields

9.3.1 FEF Header

The FEF begins with a word of DTP-FEF-HEADER. the format of which is shown in *Fig. 9-5*.

9.3.2 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 Q's occupied by the FEF. This is used for garbage collection and other system primitives that require the length of a structure.

9.3.3 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).

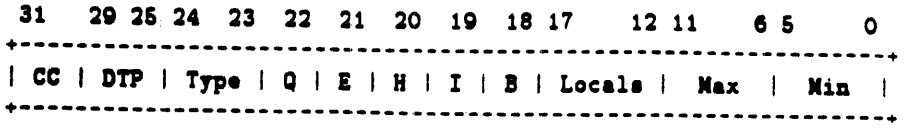
9.3.4 Numeric Argument Descriptor

The fourth word of the FEF is the FAST-ARG-OPT word, the format of which is shown in *Fig. 9-6*. This word contains the numeric argument descriptor.

9.3.5 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 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. etc. *** This is semi-inconsistant with usual convention. but probably not worth changing. *** If the bit for a pdl-slot is set, then that pdl-slot corresponds to a special variable.



CC	CDR Code, unused.
DTP	Data Type. DTP-FIX.
Type	Call type, 0: No Locals, 1: Some Locals, 2: Rest Arg (with or without other locals). 3: Complex Call.
Q	Quoted Rest, has a quoted rest arg
E	Evaled Rest, has an evaluated rest arg
H	Complex Quoting, must use ADL
I	Interpreted, no information available
B	Complex binding, must use ADL
Locals	Number of Locals
Max	Maximum Number of required plus optional args, not including the rest arg
Min	Minimum number of required args, not including the rest arg

Fig. 9-6 FEF Fast Argument Option Fields

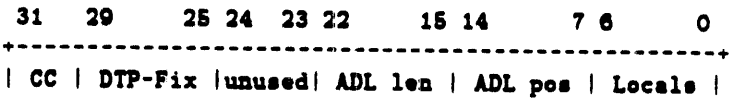


Fig. 9-7 FEF Misc Word

9.3.6 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 *Fig. 9-7*. *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 section on Arg Descriptor List.

9.3.7 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.

9.3.8 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.[9] The ADL is at the offset stored in the Misc word.

[9] See section on Arg Descriptor List.

After this come 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.

9.3.9 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. It 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.

9.3.10 Arg Descriptor List

*** I don't think this first paragraph is correct PTM 3/4/85 ***

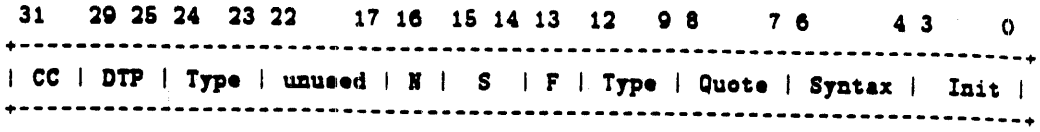
When the ADL [10] is used: If the **FEF-QUOTE-HAIR** bit is set, or the **FEF-BIND-HAIR** bit is set, or if the "S.V. bit map active" bit is clear and the Special Variables Bind bit is set, then the ADL must be present. (It may be present anyway for debugging purposes.) Also, there is a random bit in the FEF called **FAST-ARGUMENT-OPTION-ACTIVE** (or **FEFH-FAST-ARG**) which is semi-historical. If it is set, it is a guarantee that the ADL can be safely ignored.

Also, note that 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.

---

[10] Historically the ADL was, confusingly, sometimes called the "Binding Descriptor List" or BDL.



CC	CDR Code. unused.
DTP	Data Type. DTP-FIX.
N	Name Present. there is a second word containing the name of this variable.
S	Specialness. whether this variable is local. special, or remote.
F	Functional, this function has no side effects.
Type	Desired data type for this variable.
Quote	Desired quotage of this argument.
Syntax	Desired syntax of this argument.
Init	Desired initialization of this variable.

Fig. 9-8 ADL First Q

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

Table 9-1 FEF Specialness

The fields of the first Q are shown in *Fig. 9-8*.

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 variables 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 (see way below), 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).

9.3.11 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 FEF Specialness are shown in *Table 9-1*.

9.3.12 Desired Data Type

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

9.3.13 Quote Status

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

9.3.14 Argument Syntax

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

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)

Table 9-2 Desired Data Type

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	

Table 9-3 Quote Status

Value	Name	Meaning
0	FEF-ARG-REQ	Required.
1	FEF-ARG-OPT	Optional. May be initialized if arg not present.
2	FEF-ARG-REST	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.

Table 9-4 Argument Syntax

9.3.15 Init Option

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

9.3.16 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

Value	Name	Meaning
0	FEF-INI-NONE	Do not initialize (All required args have this)
1	FEF-INI-NIL	Initialize to NIL (Default for locals)
2	FEF-INI-PNTR	Initialize variable to 3rd Q
3	FEF-INI-C-PNTR	Initialize variable to the object pointed to by the 3rd Q
4	FEF-INI-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-INI-COMP-C	Variable initialized by compiled code. Initialization too hairy to be done by above mechanisms.
6	FEF-INI-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-INI-SELF	Initialize to self. used for (LAMBDA (&OPTIONAL (FOO FOO)) ..) which isn't reasonable unless FOO is special.

Table 9-5 FEF Init Option

page. or to create a new constant in the FEF and reference it instead.

9.4 Calling Conventions

nothing here — want to write it?

9.5 Closure Call

When a closure is called the bindings of the closure are reinstated and the function of the closure is called in the usual way. The bindings in a closure are to External Value Cells. Several closures may share a binding environment by sharing External Value Cells.

9.6 Select-Method Call

When a select-method is called ...

9.7 Instance Call

When an instance is called SELF is bound to the instance: then, the method is decoded by hash lookup of the first (message selector) argument in the method decode table of the flavor; the function in the stack frame (which has to now been the instance) is replaced by the method found; if GET-SELF-MAPPING-TABLE is set and SELF-MAP-PROVIDED is clear the self mapping table is looked up in the flavor: finally. the method is entered.

TI Internal Data

Code	Name
0	ADI-ERR
1	ADI-RETURN-INFO
2	ADI-RESTART-PC
3	ADI-FEXPR-CALL
4	ADI-LEXPR-CALL
5	ADI-BIND-STACK-LEVEL
6	ADI-UNUSED-6
7	ADI-USED-UP-RETURN-INFO

Table 9-6 ADI Kinds

Code	Name
0	ADI-ST-ERR
1	ADI-ST-BLOCK
2	unused-2
3	ADI-ST-MAKE-LIST
4	ADI-ST-INDIRECT

Table 9-7 ADI Storing Options

9.8 Entity Call

When an entity is called it is treated like a closure. The closure bindings are reinstated to External Value Cells and the function is called in the usual way.

9.9 ADI Formats

ADI words are additional information prepended to a call block by the caller. Several things are indicated by ADI words: multivalue return. LEXPR funcall, and FEXPR funcall all provide ADI words. Multiple ADI words may pertain to a single call block. Restart PC is used in *CATCH frames to indicate where THROW should resume execution.

Each ADI is two words. The one with the highest PDL address is the one that contains the ADI type and controls the meaning of the ADI. This word is always a fixnum. The other word can contain any lisp object and its meaning is dependant on the type of the ADI.

If there are any ADI words, the %%CLS-ADI-PRESENT flag is set. Whenever ADI are examined, they must be searched while %%PREVIOUS-ADI-FLAG is true in each ADI word. This flag is bit 30, which is part of the CDR code and is not otherwise used in ADI words.

The ADI kinds are shown in *Table 9-6*.

When the ADI is ADI-RETURN-INFO, the storing options are as listed in *Table 9-7*. Uses of RETURN-INFO ADI are discussed below in section on multiple value returns.

9.10 LEXPR Funcall

LEXPR-FUNCALL works like a cross between APPLY and FUNCALL. Its first argument is a function to call. Its last argument is a list of arguments to pass. The arguments between are also passed to the function. When a function is entered that was LEXPR called, enough arguments are spread off the rest arg to fill in the spread args. *** note will illop rather than listify spread args into rest arg ***

Tag	Meaning
NIL	CATCH-ALL
T	UNWIND-PROTECT Always continues throwing

Table 9-8 Special Catch Tags

Tag	Meaning
T	Return from function (like destination-return) Throw all the way out of the top of the stack-group. In this case we bypass CATCH-ALL's Must be used with non-null catch action.
NIL	*CATCH returns NIL as the tag if no throw or return operation occurred.

Table 9-9 Special *Throw Tags

9.11 FEXPR Funcall

If an ADI-FEXPR-CALL is present, the caller .../ *** What ??? ***

9.12 Multiple Value Returns

If a caller wants to receive multiple values, a special calling form is used that uses an ADI to indicate that multiple values are expected and how to store them. The ADI is a RETURN-INFO ADI. If the ADI has the storing option ADI-ST-BLOCK the caller has a block on the stack that the values are to be stored into. If the ADI has the storing option ADI-ST-MAKE-LIST, the value is CONS'ed with NIL into a full (two word) list node, and that list is RPLACD'ed into the previous tail of the returned values list. The result of this is that the returned values end up in order in the list of returned values. If the ADI has the storing option ADI-ST-INDIRECT allows a frame to indirect its multiple values to another frame on the same PDL *** don't understand this ***. Indirect storing is obsolete.

**** write some more ****

9.13 Catch, Throw, Unwind Protect, and Stack Unwinding

*** needs some work still ***

A CATCH is represented as an open call block. The function in the call block is the special function of DTP-U-ENTRY and entry index 0. This is the function *CATCH. The catch tag is stored as the word following the catch block, which is where the first argument would go. Several catch tags are special and are shown in *Table 9-8*. These have the meaning shown when encountered as a catch tag. If the function *CATCH is ever entered, its second argument is returned as its value.

*THROW takes two arguments, the throw *tag* and the *value* to pass to the catch. Throw searches backward on the stack for a *CATCH with whose tag is EQ to *tag*. Each open frame is examined. The throw succeeds with the first frame which is *CATCH and has a tag that is either a special catch tag or is EQ to *tag*. Certain values of *tag* have special meanings as shown in *Table 9-9*.

If the *THROW is successful, the PDL is unwound to there, and control is passed to that *CATCH as described below. This style of exit is sometimes called non-local exit. If no catch tag is found matching the throw tag, no unwinding occurs, instead a THROW-TRAP is signalled.

TI Internal Data

***UNWIND-STACK** is a generalized ***THROW** used by the error handler and by **UNWIND-PROTECT**. It takes the same first two arguments as ***THROW**. It also takes a third argument which is a count: if the count is **NIL** things are the same as ***THROW**, otherwise if this may frames are passed we resume as if a catch had been found. The fourth argument, if non-**NIL**, means that instead of resuming when we find the point to throw to, we call that function with one argument, the second arg to ***UNWIND-STACK**.

If a ***THROW** or ***UNWIND-STACK** succeeds control passes back to the frame containing the ***CATCH**. Usually the the open block for ***CATCH** has a RESTART-PC ADI. In this case, the restart PC from the ADI is used as the exit PC for the active block containing the ***CATCH**. If the open block for ***CATCH** does not contain a RESTART-PC ADI, it must be D-RETURN so that it returns from the active block containing the ***CATCH**.

To complete the throw, ***CATCH** returns up to four values: trailing null values are not returned for reasons of microcode simplicity, but the vlaues not returned will default to **NIL** if they are received with **MULTIPLE-VALUE** or **MULTIPLE-VALUE-BIND** special forms. If the catch completes normally, the first value is the value of the body of the catch and the second is **NIL**. If a ***THROW** occurs, the first value is the second argument to ***THROW** (the *value*), and the second value is the first argument to ***THROW** (the *tag*). The third and fourth values are the third and fourth arguments to ***UNWIND-STACK** if that was used in place of ***THROW**; otherwise these values are **NIL**. ***CATCH** does not propagate multiple values back from the last form.

UNWIND-PROTECT is a special form of **CATCH** that catches all stack unwinding. If the stack frame containing an **UNWIND-PROTECT** is popped for any reason, the **UNWIND-PROTECT** catches it and executes its unwind form. After the form executes, unwinding proceeds.

When the PDL is unwound, each block is examined. If it is an open block for ***CATCH** (DTP-U-ENTRY 0) then it is checked to see if it is an **UNWIND-PROTECT**. If the block is active, it is checked for special bindings, if it has bound any specials, the bindings are undone and the binding block is popped from the special PDL. Notice that it is easy to find all open and active blocks by following the chain of DELTA-TO-OPEN-BLOCK offsets. The special DELTA-TO-OPEN-BLOCK offset of 0 is placed in the first block of a PDL to indicate that the end has been reached. This is how it is noticed that all blocks have been searched and no catch tags match the throw tag.

9.14 Support Vector

The support vector is a vector in wired storage that microcode uses to reference Lisp functions. Microcode uses Lisp for hard cases of common instructions, to call the evaluator, for arithmetic on rational or complex arguments, etc. The support vector is shown in *Table 9-10*.

DEFSTRUCT-DESCRIPTION is used for **TYPEP** of named structures. **APPLY-LAMBDA** is the interpreter for DTP-LIST functions. **EQUAL** is used when a hard case of EQUAL is encountered by microcoded **EQUAL**. **PACKAGE** does not seem to be used. **EXPT-HARD** is used by EXPT when it encounters a hard case. **NUMERIC-ONE-ARGUMENT** and **NUMERIC-TWO-ARGUMENTS** are used to perform arithmetic on complex or rational numbers. The "unbound marker" does not seem to be used. **INSTANCE-HASH-FAILURE** is called if the hash lookup of a method for an instance or in a funcallable hash array is not found or if the array needs rehashing. **INSTANCE-INVOKE-VECTOR** is described below. **EQUALP** is called if a bad recursive case is encountered and **EQUALP-ARRAY** is called if attempt **EQUALP** of a non-string array.

9.15 Instance Invoke

Several of the basic operations of the macroinstruction set deal with instances by sending a message. This works by sending the instance a message with the keyword for the operation being attempted. For example, **CAR** when applied to an instance, sends **:CAR** to the instance and uses the result as the value of **CAR**. The keyword symbols for the supported operations are stored in

Index	Function
0	PRINT
1	CALL-NAMED-STRUCTURE
2	DEFSTRUCT-DESCRIPTION
3	APPLY-LAMBDA
4	EQUAL
5	PACKAGE
6	EXPT-HARD
7	NUMERIC-ONE-ARGUMENT
8	NUMERIC-TWO-ARGUMENTS
9	unbound marker
10	INSTANCE-HASH-FAILURE
11	INSTANCE-INVOKE-VECTOR
12	EQUALP
13	EQUALP-ARRAY

Table 9-10 Support Vector

Index	Operation
0	:GET
1	:GETL
2	:GET-LOCATION-OR-NIL
3	:CAR
4	:CDR
5	:SET-CAR
6	:SET-CDR

Table 9-10 Instance Invoke Vector

the instance invoke vector which microcode accesses via the support vector. The assigned slots of the instance invoke vector are shown in *Table 9-10*. If the instance does not handle the message, the unhandled message error is signalled as usual.

10. Multiprocessing

If only I had three hands!

This chapter explains multiprocessing in the Explorer System. It explains stack groups, the building blocks of multiprocessing and ...

A stack group is the data structure behind the implementation of a process in the Explorer System. 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 betwen users on the Explorer System, it is still useful for to support multiple processes; for example, when a message is received from a netword, some other stack group could be activated to handle it. stack groups are also useful for certain control structures; a solution to the *Same Fringe* problem was written using them.

10.1 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.[1] 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 , and the array leader holds many other relevant data including a pointer to another array holding the Linear Binding PDL (q. v.) for the stack group, the PDL pointers for both PDLs, various microcode registers, etc.

A useful feature is that by binding appropriate special variables, the default cons area, etc, and error and invoke 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 the stack group name, and size and limits on the PDLs. This data is set up when the stack group is created and doesn't normally change during the course of system operation. This information is loaded when the stack group is entered, but since it doesn't change the data needn't be saved when the stack group is left. The *debugging* section has information that the error handler needs to determine what error has occured and how to restart the stack group. This information is read directly by the error handler as needed and will not be loaded or saved on stack group entry or exit. The *high level* section contains state information used to determine which operations are valid on this stack group.

The elements of the array leader are:

10.1.1 Static Section

SG-NAME Name of this stack group for conversing with user about it.

---

[1] See **shallow binding**.

SG-REGULAR-PDL This is the array that is serving as the regular PDL of this stack group. It must be an ART-REG-PDL array.

SG-REGULAR-PDL-LIMIT Maximum PDL pointer value before overflow.

SG-SPECIAL-PDL Array which is the special PDL for this stack group. It must be an ART-SPECIAL-PDL array.

SG-SPECIAL-PDL-LIMIT Maximum special PDL pointer value before overflow.

SG-INITIAL-FCTN-INDEX Position in regular PDL of the topmost function pointer cell. This is AP for the top-level function of this stack group. This is normally 3, but may differ if ADI is present.

SG-UCODE Used somehow (**how??**) to indicate what microcode packages this stack group requires to be loaded.

10.1.2 Debugging Section

SG-TRAP-TAG Symbolic tag corresponding to **SG-TRAP-MICRO-PC**. Gotten by looking up the trap PC in the microcode error table. Properties of this symbol drive error reporting and recovery.

SG-RECOVERY-HISTORY Available for complex SG debugging routines to leave tracks in for debugging purposes. *** No known uses? ***

SG-FOOTHOLD-DATA Structure which saves dynamic section of "real" SG when executing in the foothold.

10.1.3 High Level Section

SG-STATE The STACK-GROUP state. This has fields describing the high level state of this stack group. See Stack Group State.

SG-PREVIOUS-STACK-GROUP Pointer to SG which called be or was interrupted "for me". (so that this SG could be run)

SG-CALLING-ARGS-POINTER Pointer to argument-block which last called this SG.

SG-CALLING-ARG-NUMBER Number of args in above block.

SG-TRAP-AP-LEVEL Used for stepping. When stepping compiled functions. will cause a STEP-BREAK trap when PDL pointer is below this virtual address.

10.1.4 Dynamic Section

SG-REGULAR-PDL-POINTER Saved PDL pointer, stored as a fixnum offset from **SG-REGULAR-PDL**.

SG-SPECIAL-PDL-POINTER Saved special PDL pointer, stored as a fixnum offset from **SG-SPECIAL-PDL**.

SG-AP Points to current active block on the stack in this stack group, stored as a fixnum offset to **SG-REGULAR-PDL**.

SG-IPMARK Points to current open block on the stack. Stored the same way.

SG-TRAP-MICRO-PC Micro-address from which a trap was signalled. This is used as a key to lookup the TAG-TAG in the microcode error table.

SG-SAVED-QLARYH The last array referenced.

SG-SAVED-QLARYL The last element of an array referenced.

TI Internal Data

SG-SAVED-M-FLAGS Saved processor flags. The flags are shown in *Table 10-1*. It unwise to change most of these. Only **METER-STACK-GROUP-ENABLE**, **CAR-NUM-MODE** and **CDR-NUM-MODE** are likely to be safe to change. In particular, changing the symbol modes will break much system code.

SG-AC-K Accumulator (register) M-K

SG-AC-S Accumulator (register) M-S

SG-AC-J Accumulator (register) M-J

SG-AC-I Accumulator (register) M-I

SG-AC-Q Accumulator (register) M-Q

SG-AC-R Accumulator (register) M-R

SG-AC-T Accumulator (register) M-T

SG-AC-E Accumulator (register) M-E

SG-AC-D Accumulator (register) M-D

SG-AC-C Accumulator (register) M-C

SG-AC-B Accumulator (register) M-B

SG-AC-A Accumulator (register) M-A

SF-AC-ZR Accumulator (register) M-ZR

SG-AC-2 Accumulator (register) M-2, pointer field in a fixnum. The data type is stored in **SG-VMA-M1-M2-VMA-TAGS**. Since this register is saved in this way, it need not contain a valid data type.

SG-AC-1 Accumulator (register) M-1, pointer field in a fixnum. The data type is stored in **SG-VMA-M1-M2-VMA-TAGS**. As with M-1, the data type need not be valid.

SG-VMA-M1-M2-TAGS The datatypes from M-1, M-2 and VMA are packed into this fixnum.

SG-SAVED-VMA VMA address register. The data type is stored in **SG-VMA-M1-M2-VMA-TAGS**. Like M-1, VMA need not contain a properly typed pointer; however, since when it is restored MD will be restored by rereading from VMA, it must contain a valid virtual address of a properly typed Q.

SG-PDL-PHASE PDL index of PDL buffer head. This assures that when this PDL is reloaded into the PDL buffer on resuming this stack group, each PDL word will be at the same word of the PDL buffer it previously occupied.

10.2 SG-State Q

The stack-group state Q has the format shown in *Fig. 10-1*.

10.3 SG Instruction Dispatch

SG-INST-DISP is a two bit field indicating which macroinstruction dispatch is in effect. It may take on the values shown in *Table 10-3*.

10.4 SG States

The encoding of stack group states is shown in *Table 10-4*.

Bit(s)	Name	Meaning
0	QBBFL	This frame has binding block on special PDL
1-2	CAR-SYM-MODE	CAR of symbol mode: 0: is an error 1: is an error except (CAR NIL) is NIL 2: is NIL 3: unused (an error)
3-4	CAR-NUM-MODE	CAR of number mode: 0: is an error 1: is NIL 2: unused (an error) 3: unused (an error)
5-6	CDR-SYM-MODE	CDR of symbol mode: 0: is an error 1: is an error excetp (CDR NIL) is NIL 2: is NIL 3: the property list
7-8	CDR-NUM-MODE	CDR of number mode: same as CAR-NUM-MODE
9	DONT-SWAP-IN	temporary flag used in creating "fresh" pages
10	TRAP-ENABLE	ILLOP if try to trap
11-12	MAR-MODE	temporary flag indicating read or write access on MAR trap
13	PGF-WRITE	temporary flag indicating current page fault is writing
14	INTERRUPT-FLAG	in interrupt handler no page faults
15	SCAVENGE-FLAG	in scavenger no sequence breaks
16	TRANSPORT-FLAG	in transporter no sequence breaks
17	STACK-GROUP-SWITCH-FLAG	switching stack groups no sequence breaks
18	DEFERRED-SQUENCE-BREAK-FLAG	remember to sequence break INHIBIT-SCHEDULING-FLAG is cleared
19	METER-STACK-GROUP-ENABLE	metering enabled
20	TRAP-ON-CALLS	trap on activating stack frame

Table 10-1 Processor Flags

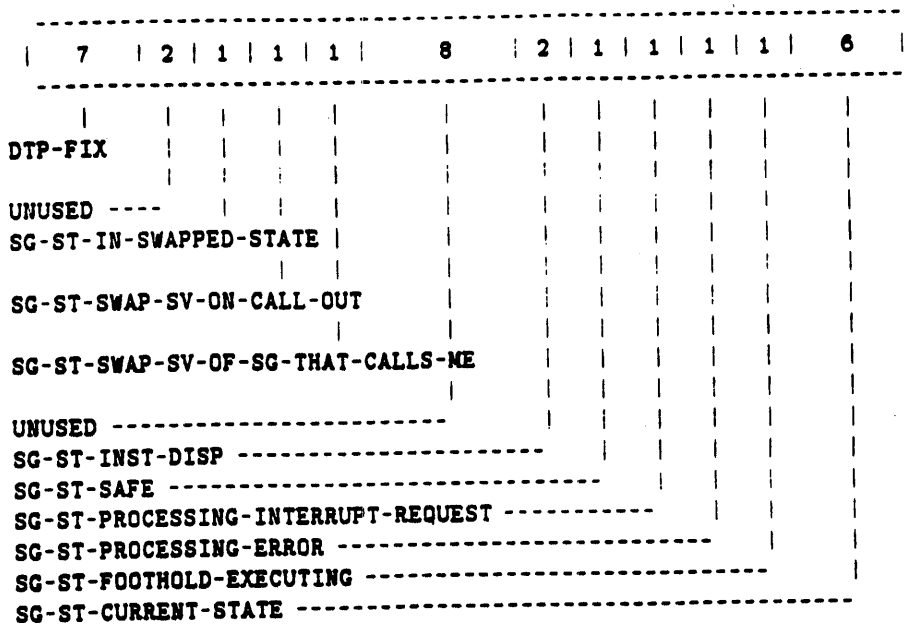


Fig. 10-1 Stack Group State Q

Field Name	Meaning
SG-ST-IN-SWAPPED-STATE	
SG-ST-SWAP-SV-ON-CALL-OUT	
SG-ST-SWAP-SV-OF-SG-THAT-CALLS-ME	
SG-ST-INST-DISP	
SG-ST-SAFE	
SG-ST-PROCESSING-INTERRUPT-REQUEST	
SG-ST-PROCESSING-ERROR	
SG-ST-FOOTHOLD-EXECUTING	
SG-ST-CURRENT-STATE	

Table 10-2 Stack Group State Fields

Value	Name	Meaning
0	SG-MAIN-DISPATCH	Main instruction dispatch.
1	SG-DEBUG-DISPATCH	Debugging dispatch.
2	SG-SINGLE-STEP-DISPATCH	Dispatch once, and then return.
3	SG-SINGLE-STEP-TRAP	For sequence breaks out of trapping instructions.

Table 10-3 Stack Group Instruction Dispatch

Value	Name	Meaning
0	SG-STATE-ERROR	Should never get this.
1	SG-STATE-ACTIVE	Actually executing on the machine.
2	SG-STATE-RESUMABLE	Reached by interrupt or error recovery completed.
3	SG-STATE-AWAITING-RETURN	After doing a "legitimate" SG-CALL.
4	SG-STATE-INVOKE-CALL-ON-RETURN	To resume this, reload SG, then simulate a store in D-LAST. The error system can produce this state when it wants to activate the foothold or perform a retry.
5	SG-STATE-INTERRUPTED-DIRTY	Get this if forced to take an interrupt at an inopportune time.
6	SG-STATE-AWAITING-ERROR-RECOVERY	Immediately after error, before recovery.
7	SG-STATE-AWAITING-CALL	
8	SG-STATE-AWAITING-INITIAL-CALL	
9	SG-STATE-EXHAUSTED	

Table 10-4 Stack Group States

11. Error Signalling

Whoops

finger on the button

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

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

11.1 Microcode Error Conditions

There are a number of conditions that are signalled by the system microcode. Listed below are the condition flavors, all of which are built directly or indirectly upon ERROR. A later section covers the names of the microcode traps themselves. All of the symbols listed are in the **ERROR-HANDLER** ("EH") package unless another package name is present.

ARG-TYPE-ERROR an argument to an operation is not of an acceptable type.

ARRAY-NUMBER-DIMENSIONS-ERROR attempt to access an array with greater or fewer dimensions than the array has. Built on BAD-ARRAY-ERROR.

BAD-ARRAY-ERROR generic array problems that lack their own condition.

CALL-TRAP-ERROR break on entry to a function or on normal exit from a ***CATCH**.

CANT-INITIATE-ON-THIS-DEVICE-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 losing.

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**.

TURD-ALERT-ERROR attempt to draw on a sheet which has not been prepared.

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

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

UNIMPLEMENTED-HARDWARE-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.

11.2 Microcode Error Table

This chapter is a description of all existing error-table entries. as of Explorer Systemmicrocode 182 and error-handler files **SYS:EH;EH.LISP#359**, and **SYS:EH;EHF.LISP#213**. Information presented here is mostly from the code in **SYS:EH;EHF**.

The error table, read from **SYS: UBIN; CONTROL TBL nnn** into **MICROCODE-ERROR-TABLE**, describes the symbolic meaning of certain microcode PC's. Its data is rearranged into other variables below. The **MICROCODE-ERROR-TABLE** relates the micro PC to a symbolic description of error. called an ETE (for Error Table Entry), whose CAR will then have properties saying how to construct a condition instance. The properties are defined in **SYS: EH; EHF** and are described below.

Each place in the microcode where **TRAP** can be called is followed by an **ERROR-TABLE** pseudo-op. This should appear 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. So, the *first* element is the name of the error, and the *second* is the first "argument" to that error's associated routines.

All ETE's should be a list whose CAR is a symbol. That symbol should be defined in a **DEF-UCODE-ERROR** in this file.

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 important to us.

In the "argument" lists below, an argument with suffix "-location" should be one of **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** meaning the top of the pdl, **VMA, RMD** meaning **MD**, or **(PP** *number*) where *number* is a negative index into the PDL. These values are used by **EH:SG-CONTENTS** to access values saved in stack groups for the purposes of examination, replacement, and restarting.

These registers are the ones saved in stack groups. That means they will be pushed on the PDL. written to memory, **GC-WRITE-TEST**'ed, and have other nasty things done to them. 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 with suffix "-tag" is a pseudolabel defined by a **RESTART** entry. See section on restart below.

TI Internal Data

11.2.1 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 had 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 pseudolabel 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; it doesn't matter, though: there aren't any such entries, anyway.

11.2.2 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-ERROR**s 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, eg, (fixnum bignum) for "fixnum or bignum."

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 (that's how all 32 bits are preserved). **VMA** is also not allowed.

Arg-number is obvious, zero origin.

Restart-tag is the label to restart from, if other than current pc.

Function-name is the erring function, if not obvious.

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 array called as function."

BAD-ARRAY-DIMENSION-NUMBER array-location dimension-number-location **"The dimension number S is out of range for S."** Is dimension-number the dimension or its position in the order?

BAD-ARRAY-TYPE array-header-location The array type of this array was not one of the legal types (see **ARRAY-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 FEF's.

CALL-TRAP Looks to be 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 Allocate zero storage? No documentation. Is used.

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 Looks to be 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. From ur-flonum: "ARGTYP not usable, I think I lost the arg."

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."**

TI Internal Data

ILLEGAL-INSTRUCTION Illegal macroinstructions that aren't unimplemented miscops. "**There was an attempt to execute an invalid instruction: O.**"

INDIVIDUAL-SUBSCRIPT-OOB array-location dimension-number-location restart-tag Dimension-number is the location of the offending dimension's index. "We assume that the current frame's args are the array and the subscripts. and find the actual losing subscript that way."

INSTANCE-LACKS-INSTANCE-VARIABLE var-location instance-location

''Signaled 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 currently unsupported 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.**" (formerly TURD-ALERT).

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 overflown.**" Pdl-type is either **REGULAR** or **SPECIAL**.

RASTER-WIDTH-TOO-WIDE "**The raster width of a font passed to equal to 32 pixels.**"

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." From ur-return: "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 ~@[(in ~A)~]."

‡TURD-ALERT sheet-location

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**.

TI Internal Data

USER-NUBUS-ERROR high-address-location low-address-location nubus-tms-type-location **"User NuBus Error of type S, at address #x 16R 16,6,48R. %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-ARGS-TO-SELECT-METHOD select-method-location **" S was applied to no arguments."**

*** much more to come: explain the error handler stack group, what levels can signal an error, restrictions on the error signaller, restarting, ... ***

11.3 ILLOP (Illegal Operation)

YOU ARE HITTING THE MOON AT EXACTLY 0276.20 MILES PER HOUR. BLOOD, GUTS. TWISTED METAL

Space War, a famous video game

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 worsten the situation or complicate it beyond analysis. Instead, ILLOP will make notes about the error in a crash record in the NV RAM 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 *** list assumptions: basically that the processor kernal is well and that A-Zero, M-Zero, A-Ones, and M-Ones are set up, and that the NuBus is available and that the NV RAM 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 failure in the crash record.

11.3.1 Crash Record

ILLOP stores some of the machine state in the non-volatile RAM so that next successful startup can explain the cause of the crash, or 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 should be kept in a circular buffer. 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.

NV-RAM-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

Table 11-1 Crash Record Allocation Registers

11.3.1.1 Allocation

Allocation of the crash record is controlled by 4 16-bit numbers that are stored at a known place within the NV RAM. These are shown in *Table 11-1*.

All of the allocation registers are 16-bit byte offsets into the NV-RAM. NV-RAM offset will be expressed in hexidecimal. Also since NV-RAM is 8-bit memory stored one-per-word, addresses which are multiples of four are used (eg. 0, 4, 8, C, 10, ...). A 16-bit quantity (ie. 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, $F4B2_{16}$ would be stored in NVRAM-CRASH-BUFF-POINTER with "B2" in NV-RAM-BASE+90_{16} and "F4" in NV-RAM-BASE+94_{16}.

NVRAM-CRASH-BUFF-FORMAT-PROCESSOR is the code of the processor that is to use these records. It is currently always 0. NVRAM-CRASH-BUFF-FORMAT-REV is the format revision of the crash record format in use. Currently revision 0 is the most recent revision.

NVRAM-CRASH-BUFF-POINTER is the offset into the NV RAM (in bytes) of the beginning of the currently selected crash record. The currently selected crash record describes the last 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 backwards 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.

To view the previous crash record: The record pointer is the NVRAM-CRASH-BUFF-POINTER minus the NVRAM-CRASH-BUFF-REC-LEN. If this is less than NVRAM-CRASH-BUFF-BASE, it should be set to NVRAM-CRASH-BUFF-LAST.

11.3.1.2 Format

The crash record format is shown in *Table 11-2*.

Halt kinds are listed in *Table 11-3*.

NVRAM-CRASH-BUFF-POINTER plus (hex)	Contents
0	Progress, how far gotten
	about load
4	Disk Controller slot number
8	Disk Device Number for microcode
C	Disk Device Number for Load Band
10	Microload Name (4)
20	Load Band Name (4)
30	Microload Version (2)
38	Load Band Version (2)
40	Load Band Revision (2)
	date and time of boot
48	Month
4C	Day
50	Year
54	Hour
58	Minute
5C	Report Flags
	about shutdown
60	Halt/Crash Microaddress (2)
68	Halt Kind
6C	Boot Kind
	saved data
70	contents of M-1 (4)
80	contents of M-2 (4)
90	contents of MD (4)
A0	contents of VMA (4)

Table 11-2 Crash Record Format

Code	Meaning
0	Did not halt. Hang or non-crash restart.
1	Microcode halt. Crash table decodes meaning.
2	Hardware halt. Currently unused.
3	Lisp halt. Software halted machine. If Halt-Addr is zero then this is normal system shutdown.

Table 11-9 Halt Kinds

12. Macro Instructions

When the metal is hot.
the engine is hungry
and we're all about to see the light

Jim Steinman. in Meatloaf's *Bat Out of Hell*

In this chapter. each of the Lisp Machine instructions is described. These are called macro instructions at most places in this document to avoid confusion with microinstructions. In this chapter. "instruction" will suffice.

12.1 General Format

Instructions are always sized in 16-bit units. Almost all instructions are 16 bits long, a few branch instructions are 32 bits long. This means that usually two instructions are stored per word. 32-bit instructions need not be word aligned.

Instructions are stored in a part of the FEF[1] that does not have a data type on each word. This leaves the entire 32 bits for data add allows two instructions to be stored in each word.

The basic instruction format is the main instruction (class I) format. Certain values in the opcode field of the class I instruction indicate that it is of some other class. The format of a class I instruction is shown in *Fig. 12-1*. The mapping of the opcode field into instruction class is shown in *Table 12-1*. The formats of the other instruction classes will be discussed in their respective sections. but some field of main instruction format is shared with each of the others so the fields of the main instruction will be described here.

12.1.1 Destination Field

The destination field of the instruction is used in Class I. Class IV, and Class V instructions. It is a 2-bit field used to specify where the result of the instruction is to go. Instructions in Class II have an implicit definition of where the destination is stored (usually, on the stack). Instructions in Class III do not have a result.

The encoding of the destination field of the instruction is shown in *Table 12-2*. D-IGNORE means that the result is not stored anywhere. but it does still. set the indicators (see below). D-PDL (or D-STACK) indicates that the result is to be pushed onto the PDL and also set the indicators.

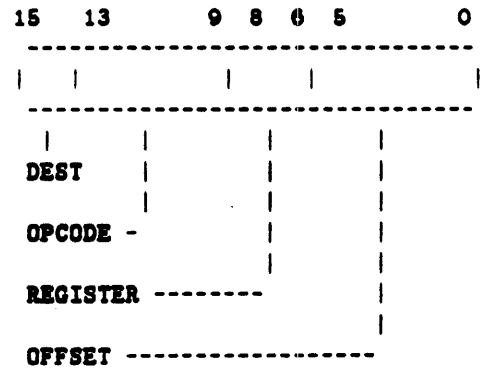


Fig. 12-1 Main Instruction Format

[1] See section on FEF format.

Opcode	Class
0-8	I
9-11	II
12	III
13	IV
14	II
15	unused
16	V
17-24	unused
25-27	II
28	III
29	IV
30	II
31	unused

Table 12-1 Opcode to Class Map

Code	Destination
0	**D-IGNORE**
1	**D-PDL**
2	**D-RETURN**
3	**D-LAST**

Table 12-2 Destination Field

D-RETURN indicates that the result is to be the result of the current function. That result is returned to the caller according to the destination of the **CALL** instruction that invoked this function. **D-LAST** indicates that the result is to be pushed onto the PDL (like in destination PDL) as the last argument to a pending call. Destination LAST completes the calling of the function and transfers control to it.[2]

D-PDL and **D-LAST** work together to make the passing of **&REST** arguments easy and inexpensive. As the arguments to a pending call are pushed with **D-PDL** the **CDR-CODE** of each is set to **CDR-NEXT**. When the last argument is pushed with **D-LAST** its **CDR-CODE** is set to **CDR-NIL**. This forms a CDR-coded list on the stack. When the function is entered, if an **&REST** argument is expected a **DTP-LIST** pointer is made to the argument that is the **CAR** of the **&REST** arg.

12.1.2 Register Field

The register and offset fields together specify the effective address of the operand. These fields are used in Class I and Class II instructions. The effective address is formed by using the register specified in the register field as the base register and adding offset to it.

The register field is encoded as shown in *Table 12-3*.

The **FEF** and **FEF+?** registers are for addressing symbols and constants used in the function. The **FEF+64**, **FEF+128**, and **FEF+192** forms increase the range of the FEF that may be used for symbols and constants. The function object of the currently executing stack frame (which will be a FEF since only FEF functions contain macroinstructions) is used as the base address. Offset is added to the base to produce an address of a Q in the FEF. This Q is read normally (it is normally transported).

[2] See section on argument passing.

Code	Base Register
0	FEF
1	FEF+64
2	FEF+128
3	FEF+192
4	CONSTANTS
5	LOCALS
6	ARGS
7	PDL/IVAR

Table 12-3 Register Field Encoding

Offset	Object Type	Object
0	Symbol	NIL
1	Symbol	T
2	Fixnum	0
3	Fixnum	1
4	Fixnum	2
5	Fixnum	3
6	Fixnum	4
7	Fixnum	5
8	Fixnum	6
9	Fixnum	7
10	Fixnum	8
11	Fixnum	9
12	Fixnum	10
13	Fixnum	-1
14	Fixnum	-2
15	Fixnum	-3
16	Fixnum	-4

Table 12-4 Constants Table

For references to the value or function cells of symbols, the addressed location in the FEF contains an **EXTERNAL-VALUE-CELL-POINTER** (EVCP) to the **VALUE-CELL** of the symbol (if used as a special variable) or the **FUNCTION-CELL** of the symbol (if used as a function).

The **CONSTANTS** register addresses a table in the constants area[3] in wired memory[4] that contains commonly used constants. The constants area is sometimes refered to as the quote vector. The constants table is shown in *Table 12-4*.

The **LOCALS** register provides access to the locals block on the PDL of the currently active function.[5] The **ARGS** register provides access to the argument block on the PDL of the currently active function. The address of argument n is computed as $AP+n+1$, where AP is the argument pointer.

The **PDL/IVAR** register is used for two distinct purposes. If **OFFSET** is 63, then the effective address is the top of the PDL. If **OFFSET** is less than 56, it is an instance variable (**IVAR**) reference. If **OFFSET** is in the range of 0 to 31, the effective address is to the unmapped instance variable

[3] See section on areas.
[4] See section on paging.
[5] See section on PDL layout.

at the slot number indicated by offset. If OFFSET is in the range 32 to 55, the effective address is to a mapped instance variable. The effective address is calculated by reading the self mapping table indexed by *OFFSET* - 32. The result is the slot number of the instance variable in SELF. The effective address of an IVAR slot is the slot number plus one addressed from a base of SELF (the instance).

All of the base registers address storage that is "inside the machine"[6] except for FEF and IVAR references. These then are the only references that need to be transported on read. These references can cause a TRANS-TRAP to be signalled if the Q read contains an invalid object or is unbound.

Some instructions such as POP use the effective address as a destination in which to store the result. Not all registers are supported as the base for a store. Only FEF (and FEF+), ARG, and LOCAL registers are currently supported as bases for storing. In the MOVEM instruction, PDL is also available as a destination to give an instruction that pushes a second copy of the top of the stack (usually named DUP in stack computers). *** soon will support IVAR for stores too ***

12.2 Indicators

The instruction set uses two indicators: NIL and ATOM. These are testable in branch instructions (Class III). Instructions that produce a result set the indicators (even if the destination is D-IGNORE). The NIL indicator is set only if the result is the symbol NIL. The atom indicator is set only if the result is not a list (an object of DTP-LIST).

Note on implementation: The indicators are not actually physical flags in the Low Cost Lisp processor. Instead, the result is remembered and the saved result is only tested when an indicator is checked. This saves hardware complexity or increases speed. The "virtual indicators" register is M-T. Every macroinstruction (except for the branch instructions which have no result) must leave its result in M-T where it can be tested if needed.

12.3 Kernel Macroinstructions

The macroinstructions can be broken into a kernel group which are needed to execute compiled Lisp, advanced kernel group which supports exended functions of the runtime architecture (eg. multiple value return), and high-speed functions which need only be instructions to allow for high speed implementations (eg. MEMQ).

The basic architecture of the machine is defined by the kernel macroinstructions. This group includes CALL, CALLO, MOVE, CAR, and CDR from the Class I macroinstructions. The arithmetic and logical instructions of non-destination group 1 are also kernel macroinstructions, as are the comparison instructions of non-destination group 2 (=, >, <, and EQ) and BINDPOP, PUSHE, MOVEM, and POP from non-destination group 3. These taken with the branch instructions forms the core instruction set for compiled Lisp programs. A full understanding of these few instructions is essential to understanding the functioning of any compiled Lisp function.

12.4 Main Instructions (Class I)

The format of Class I instructions has already been described. There are 8 Class I instructions, which are described below. All have both an effective address and a destination.

CALL *function* (Class I)

Open a call block on the stack, to call *function*. Whatever the function returns will go to the destination. The actual transfer of control will not happen until the arguments have been stored.

---

[6] See section on in the machine.

Function must be a callable object, a FEF, a symbol with a function property, a list (a lambda expression), an instance, an entity, a select-method, an array, a closure, a stack group, or a microcode entry (these are all user visible types except for numbers); if not the instruction which uses **D-LAST** to finish the call will get a trap. If *function* is a number, the trap signalled will be **NUMBER-CALLED-AS-FUNCTION**. If the contents of the effective address is an invalid data type or the unbound marker, **TRANS-TRAP** will be signalled by **CALL** just as other instructions signal **TRANS-TRAP** for unreadable arguments.

CALLO *function* (Class I)

Call a *function* without passing any arguments. Opens a call block on the stack and then transfers control immediately. What the function returns will go to the destination.

Since the call is immediately finished, the trapping on illegal functional objects as described in **CALL** occurs on this instruction.

CALLO is needed to call a function passing no arguments since there is no "last" argument to pass to **D-LAST**.

MOVE *from* (Class I)

Copies the contents of from to the destination. The source is not destroyed unless it is **PDL-POP**.

MOVE is four different instructions depending on the destination: With destination PDL, this is a "**PUSH**" instruction. With **D-IGNORE**, this is a "**CHECK**" instruction that only sets the indicators based on the source. With **D-RETURN**, this is a return instruction. And with **D-LAST**, this is a variation of push that completes the pending call.

CAR *list* (Class I)

Takes the **CAR** of *list* and stores it in the destination. *List* must be an object of a "list-like" data type, otherwise an **ARGTYP** trap is signalled.

CDR *list* (Class I)

Takes the **CDR** of *list* and stores it in the destination. *List* must be an object of a "list-like" data type, otherwise an **ARGTYP** trap is signalled.

CADR *list* (Class I)

Takes the **CADR** of *list* and stores it in the destination. *List* must be an object of a "list-like" data type, otherwise an **ARGTYP** trap is signalled.

CDDR *list* (Class I)

Takes the **CDDR** of *list* and stores it in the destination. *List* must be an object of a "list-like" data type, otherwise an **ARGTYP** trap is signalled.

CDAR *list* (Class I)

Takes the **CDAR** of *list* and stores it in the destination. *List* must be an object of a "list-like" data type, otherwise an **ARGTYP** trap is signalled.

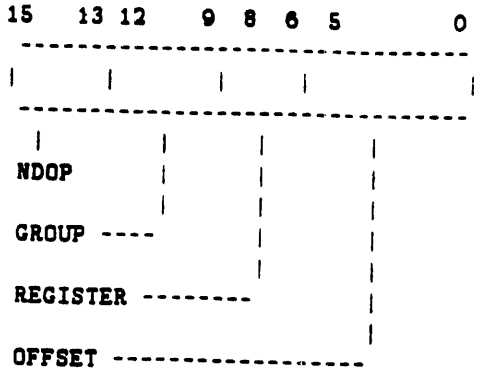


Fig. 12-2 Non-Destination Instruction Format

Code	Group
9	1
10	2
11	3
14	4

Table 12-5 Non-Destination Groups

CAAR *list* (Class I)

Takes the **CAAR** of *list* and stores it in the destination. *List* must be an object of a "list-like" data type. otherwise an **ARGTYP** trap is signalled.

12.5 Non-Destination Instructions (Class II)

Non-destination instructions have an effective address but no destination field. Most produce a result which is stored in an implicit destination (usually the PDL). Non-destination instructions are most useful for stack operations such as arithmetic and POP.

The format of a non-destination instruction is shown in *Fig. 12-2*. The high 3 bits of a non-destination instruction select the operation within the group. This field occupies the same bit positions as the destination field and the high bit of the Class I opcode field. The remainder of the Class I opcode field selects the non-destination group (if it is a non-destination instruction). The non-destination values of this field are shown in *Table 12-5*. The low 9 bits form the register and offset fields of an effective address as in Class I instructions.

12.5.1 Group 1

The **NDOP** of a non-destination group 1 instruction is shown in *Table 12-6*. Those instructions are shown below. All group 1 instructions will fetch from the effective address.

+ stack: *augend* EA: *addend result to stack* (Class II)

Adds *addend* to *augend*. *Addend* and *augend* must both be numeric or an **ARGTYP** trap is signalled.

- stack: *x* EA: *y result to stack* (Class II)

Subtracts *y* from *x*. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

NDOP	Instruction
0	illegal
1	+ (PLUS)
2	- (DIFFERENCE)
3	* (TIMES)
4	/(QUOTIENT)
5	AND
6	XOR
7	IOR

Table 12-6 Non-Destination Group 1 Decoding

NDOP	Instruction
0	= (EQUAL)
1	> (GREATERP)
2	< (LESSP)
3	EQ
4	SCDR
5	SCDDR
6	SETE-1+
7	SETE-1-

Table 12-7 Non-Destination Group 2 Decoding

***** stack: *x* EA: *y* ***result to stack*** (Class II)

Multiplies *x* and *y*. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

/ dividend stack: ***divisor*** EA: ***result to stack*** (Class II)

Divides ***dividend*** by ***divisor***. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

AND stack: *x* EA: *y* ***result to stack*** (Class II)

Computes the bit-wise logical AND of *x* and *y*. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

XOR stack: *x* EA: *y* ***result to stack*** (Class II)

Computes the bit-wise logical exclusive-or of *x* and *y*. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

IOR stack: *x* EA: *y* ***result to stack*** (Class II)

Computes the bit-wise logical inclusive-or of *x* and *y*. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

12.5.2 Group 2

The **NDOP** of a non-destination group 2 instruction is shown in *Table 12-7*. Those instructions are shown below. All group 2 instructions will read from the effective address. and the last four also store back into the effective address.

= stack: *x* EA: *y* *result to stack* (Class II)

Compares *x* to *y*. Returns the symbol T if *x* and *y* are numerically equal, otherwise returns the symbol **NIL**. An integer can be = to a flonum. Both *x* and *y* must be numbers, otherwise an **ARGTYP** trap is signalled.

> stack: *x* EA: *y* *result to stack* (Class II)

Compares *x* to *y*. If *x* is greater than *y* (see Lisp **GREATERP**), result is the symbol T, otherwise it is the symbol **NIL**. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

< stack: *x* EA: *y* *result to stack* (Class II)

Compares *x* to *y*. If *x* is less than *y* (see Lisp **LESSP**), result is the symbol T, otherwise it is the symbol **NIL**. *X* and *y* must both be numeric or an **ARGTYP** trap is signalled.

EQ stack: *x* EA: *y* *result to stack* (Class II)

Compares *x* to *y*. If the two are identical (see Lisp **EQ**), result is the symbol T (true), otherwise it is the symbol **NIL** (false).

SCDR source and destination: *var* (Class II)

Takes the **CDR** of the contents *var* and stores it back in the contents of *var*. (**setq frob (cdr frob)**) translates to this instruction. The contents of *var* must be a "list-like" object, or an **ARGTYP** trap is signalled.

SCDDR source and destination: *var* (Class II)

Takes the **CDDR** of the contents of *var* and stores it back as the contents of *var*. (**setq frob (cddr frob)**) translates to this instruction. The contents *var* and its **CDR** must be "list-like" objects, or an **ARGTYP** trap is signalled.

SETE-1+ source and destination: *var* (Class II)

Increments the contents of *var* and stores the result back as the contents of *var*. (**setq frob (1+ frob)**) translates to this. The contents of *var* must be numeric or an **ARGTYP** trap is signalled.

SETE-1- source and destination: *var* (Class II)

Decrements the contents of *var* and stores the result back as the contents of *var*. (**setq frob (1- frob)**) translates to this. The contents of *var* must be numeric or an **ARGTYP** trap is signalled.

12.5.3 Group 3

The **NDOP** of a non-destination group 3 instruction is shown in *Table 12-8*. Those instructions are shown below. These instructions store into the effective address.

BIND source and destination: *var* (Class II)

Save the current value of *var* on the special binding stack and leave its value as it was.

NDOP	Instruction
0	BIND
1	BINDNIL
2	BINDPOP
3	SETNIL
4	SETZERO
5	PUSHE
6	MOVEM
7	POP

Table 12-8 Non-Destination Group 3 Decoding

BINDNIL source and destination: *var* (Class II)

Save the current value of *var* on the special binding stack and set its value to **NIL**.

BINDPOP source and destination: *var* stack: *newval* *(Class II)*

Save the current value of *var* on the special binding stack and set its value to *newval*.

SETNIL destination: *var* *(Class II)*

Set the value of *var* to symbol **NIL**.

SETZERO destination: *var* *(Class II)*

Set the value of *var* to the fixnum zero.

PUSHE EA: *loc* result to stack (Class II)

Push a locative pointer to *loc* onto the stack.

MOVEM destination: *dest* stack: *copy* *(Class II)*

Copies *copy* from the stack without popping it. The result is stored at the effective address, *dest*.

POP destination: *dest* stack: *source* *(Class II)*

Copies *source* from the stack to the effective address, *dest*. *Source* is removed (popped) from the stack.

12.5.4 Group 4

The **NDOP** of a non-destination group 4 instruction is shown in *Table 12-9*. Those instructions are shown below. These use effective address in several different ways including using those bits of the instruction for something entirely different.

STACK-CLOSURE-DISCONNECT stack: *arg?* EA: *arg?* *result to stack* (Class II)

*** What does this do?? Look in new UC-STACK-CLOSURE ***

NDOP	Instruction
0	STACK-CLOSURE-DISCONNECT
1	STACK-CLOSURE-UNSHARE
2	MAKE-STACK-CLOSURE
3	PUSH-NUMBER
4	STACK-CLOSURE-DISCONNECT-FIRST
5	PUSH-CDR-IF-CAR-EQUAL
6	PUSH-CDR-STORE-CAR-IF-CONS
7	illegal

Table 12-9 Non-Destination Group 4 Decoding

STACK-CLOSURE-UNSHARE stack: *arg?* EA: *arg?* *result to stack* (Class II)

*** What does this do?? Look in new UC-STACK-CLOSURE ***

MAKE-STACK-CLOSURE stack: *arg?* EA: *arg?* *result to stack* (Class II)

Make a new stack closure. *** What does this do?? Look in new UC-STACK-CLOSURE ***

PUSH-NUMBER *number* result to stack (Class II)

Push effective address as a fixnum. This takes the 9-bit effective address field of the instruction and returns it as a positive fixnum.

STACK-CLOSURE-DISCONNECT-FIRST stack: *arg?* EA: *arg?* *result to stack* (Class II)

*** What does this do?? Look in new UC-STACK-CLOSURE ***

PUSH-CDR-IF-CAR-EQUAL stack: *frob* EA: *list* *result to stack* (Class II)

Takes *list*, popped from the stack. if *list* is a cons compares its **CAR** to *frob*: if equal, the **CDR** of *list* is pushed onto the stack. If either *list* is not a cons or the **CAR** of *list* is not equal to *frob*, nothing is pushed onto the stack and the indicators are set to the symbol **NIL**.

*** This is mostly useful for maintaining a loop variable on the stack? ***

PUSH-CDR-STORE-CAR-IF-CONS source and destination: *dest* stack: *frob* *(Class II)*

Takes *frob*. popped from the stack: if *frob* is a cons. the **CDR** of *frob* is pushed onto the stack and the **CAR** of *frob* is stored into *dest*. otherwise nothing is pushed or stored and the symbol **NIL** is left in the indicators.

This is mostly useful for maintaining a loop variable (eg. for **DOLIST**).

TI Internal Data

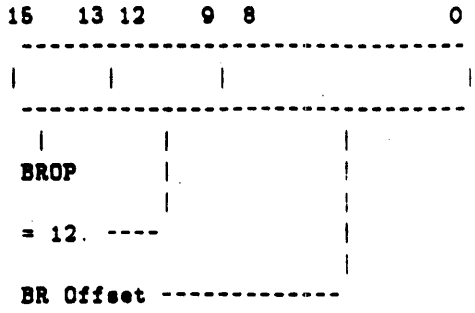


Fig. 12-3 Branch Instruction Format

12.6 Branch Instructions (Class III)

Branch instructions alter program flow within a function but cannot transfer outside of the currently executing FEF. The unconditional branch instruction, **BR**, always transfers to the target. Conditional branch instructions test one of the indicators and transfers if the test is true. Two more complex branch instructions are provided for looping support.

Branch instructions have no effective address nor destination field. Instead, the effective address field is used as a 9-bit branch offset. The branch offset is taken as a signed two's complement quantity to be added to the location counter (LC) after it has be incremented to point to the instruction following the branch instruction. This sum is the LC of the branch target.

If the branch offset is -1.[7] then a long branch is indicated. The next 16 bits of the instruction stream is read and it is used as a signed two's complement quantity to be added to the LC after it has be incremented to point to the instruction following the branch instruction.[8]

If the branch is not taken, execution continues with the instruction following the branch instruction. Note that even if the branch is not taken, the branch offset must be checked for a long branch, and if it is a long branch, the next halfword (the long branch offset) must be skipped.

The format of a branch instruction is shown in *Fig. 12-3*. The high 3 bits of a branch instruction select the particular branch operation. This field occupies the same bit positions as the destination field and the high bit of the Class I opcode field. The remainder of the Class I opcode field is 12 to select Class III instructions. The remaining 9 bits (the Class I Register and Offset fields) are the branch offset. The coding of the selection of branch operation field is shown in *Table 12-10*.

BR ***offset*** **(Class III)**

Always branch to the location indicated by the branch offset.

---

[7] A branch offset of -1 makes the branch target, the branch instruction itself; and since branch instructions do not alter the indicators, that branch, if taken, would close a one instruction long endless loop. The reuse of this value makes this particular kind of endless loop impossible.

[8] Note that a negative (backward) long branch is one greater in magnitude than it would be if it were a normal (short) branch because there is an extra halfword to branch over in the instruction stream for the long branch offset. If the branch is positive (forward) there is no extra halfword to branch over.

BROP	Instruction
0	BR
1	BR-NIL
2	BR-NOT-NIL
3	BR-NIL-ELSE-POP
4	BR-NOT-NIL-ELSE-POP
5	BR-ATOM
6	BR-NOT-ATOM
7	illegal

Table 12-10 Branch Operation Decoding

BR-NIL *offset* (Class III)

Branch if the **NIL** indicator is set. Otherwise, continue with the next instruction sequentially.

BR-NOT-NIL *offset* (Class III)

Branch if the **NIL** indicator is not set. Otherwise, continue with the next instruction sequentially.

BR-NIL-ELSE-POP *offset* (Class III)

Branch if the **NIL** indicator is set. Otherwise, pop the stack and continue with the next instruction sequentially. This is useful in some loops.

BR-NOT-NIL-ELSE-POP *offset* (Class III)

Branch if the **NIL** indicator is not set. Otherwise, pop the stack and continue with the next instruction sequentially. This is useful in some loops.

BR-ATOM *offset* (Class III)

Branch if the **ATOM** indicator is set. Otherwise, continue with the next instruction sequentially.

BR-NOT-ATOM *offset* (Class III)

Branch if the **ATOM** indicator is not set. Otherwise, continue with the next instruction sequentially.

12.7 Miscellaneous Instructions (Class IV)

Miscellaneous instructions have a destination but no effective address field. Most take arguments on the stack. Misc instructions are mostly used for Lisp functions that are microcoded for speed and for very special purpose instructions, eg. %GC-SCAVENGE.

The format of a misc instruction is shown in *Fig. 12-4*. The high 2 bits select the destination, as in Class I instructions. The Class I opcode field is either 13 or 29 to select a Class IV instruction. If 13, this is a Group 0 miscellaneous instruction. If 29, this is a Group 1 miscellaneous instruction. The remainder of the Class I instruction is the miscellaneous operation field. This 9-bit field selects the instruction within the group.

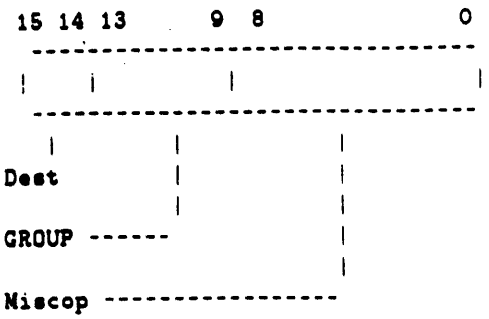


Fig. 12-4 Miscellaneous Instruction Format

12.7.1 Miscops Group 0

Group 0 is the old form of miscellaneous instructions. It is the only group currently in use. *** large task remains to complete this documentation!! ***

*** explain the 0 - 200 miscops for AR-1, AS-1, AP-1, and instance refs *** **** aren't these in group V now ****

Note: Miscops 240, and 241 are free.

M-CAR *list* (Class 4 - miscop 242)

Takes the **CAR** of *list* and leaves it in the destination. Signals an **ARGTYP** trap if *list* is not a list or an allowed **CAR** of a symbol or number.

M-CDR *list* (Class 4 - miscop 243)

Takes the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* is not a list or an allowed **CDR** of a symbol or number.

M-CAAR *list* (Class 4 - miscop 244)

Takes the **CAR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CAR** is not a list or an allowed **CAR** of a symbol or number.

M-CADR *list* (Class 4 - miscop 245)

Takes the **CAR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CDR** is not a list or an allowed **CAR** or **CDR** of a symbol or number.

M-CDAR *list* (Class 4 - miscop 246)

Takes the **CDR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CAR** is not a list or an allowed **CAR** or **CDR** of a symbol or number.

M-CDDR *list* (Class 4 - miscop 247)

Takes the **CDR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CDR** is not a list or an allowed **CDR** of a symbol or number.

CAAAR *list*

(Class 4 - miscop 250)

Takes the **CAR** of the **CAR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CAR** or the **CAR** of its **CAR** is not a list or an allowed **CAR** of a symbol or number.

CAADR *list*

(Class 4 - miscop 251)

Takes the **CAR** of the **CAR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CDR** or the **CAR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CADAR *list*

(Class 4 - miscop 252)

Takes the **CAR** of the **CDR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CAR** or the **CDR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CADDR *list*

(Class 4 - miscop 253)

Takes the **CAR** of the **CDR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CDR** or the **CDR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDAAR *list*

(Class 4 - miscop 254)

Takes the **CDR** of the **CAR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CAR** or the **CAR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDADR *list*

(Class 4 - miscop 255)

Takes the **CDR** of the **CAR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CDR** or the **CAR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDDAR *list*

(Class 4 - miscop 256)

Takes the **CDR** of the **CDR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CAR** or the **CDR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDDDR *list*

(Class 4 - miscop 257)

Takes the **CDR** of the **CDR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list* or its **CDR** or the **CDR** of its **CDR** is not a list or any is a disallowed **CDR** of a symbol or number.

CAAAAR *list*

(Class 4 - miscop 260)

Takes the **CAR** of the **CAR** of the **CAR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CAR**, the **CAR** of its **CAR**, or the **CAR** of the **CAR** of its **CAR** is not a list or any is a disallowed **CAR** of a symbol or number.

CAAADR

(Class 4 - miscop 261)

list Takes the **CAR** *of the* **CAR** *of the* **CAR** *of the* **CDR** *of list and leaves the result in the destination. Signals an*

CAADAR *list* (Class 4 - miscop 262)

Takes the **CAR** of the **CAR** of the **CDR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CAR**, the **CDR** of its **CAR**, or the **CAR** of the **CDR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CAADDR *list* (Class 4 - miscop 263)

Takes the **CAR** of the **CAR** of the **CDR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CDR**, the **CDR** of its **CDR**, or the **CAR** of the **CDR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CADAAR *list* (Class 4 - miscop 264)

Takes the **CAR** of the **CDR** of the **CAR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CAR**, the **CAR** of its **CAR**, or the **CDR** of the **CAR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CADADR *list* (Class 4 - miscop 265)

Takes the **CAR** of the **CDR** of the **CAR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CDR**, the **CAR** of its **CDR**, or the **CDR** of the **CAR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CADDAR *list* (Class 4 - miscop 266)

Takes the **CAR** of the **CDR** of the **CDR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CAR**, the **CDR** of its **CAR**, or the **CDR** of the **CDR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CADDDR *list* (Class 4 - miscop 267)

Takes the **CAR** of the **CDR** of the **CDR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CDR**, the **CDR** of its **CDR**, or the **CDR** of the **CDR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDAAAR *list* (Class 4 - miscop 270)

Takes the **CDR** of the **CAR** of the **CAR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CAR**, the **CAR** of its **CAR**, or the **CAR** of the **CAR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDAADR *list* (Class 4 - miscop 271)

Takes the **CDR** of the **CAR** of the **CAR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CDR**, the **CAR** of its **CDR**, or the **CAR** of the **CAR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDADAR *list* (Class 4 - miscop 272)

Takes the **CDR** of the **CAR** of the **CDR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CAR**, the **CDR** of its **CAR**, or the **CAR** of the **CDR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDADDR *list* (Class 4 - miscop 273)

Takes the **CDR** of the **CAR** of the **CDR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CDR**, the **CDR** of its **CDR**, or the **CAR** of the **CDR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

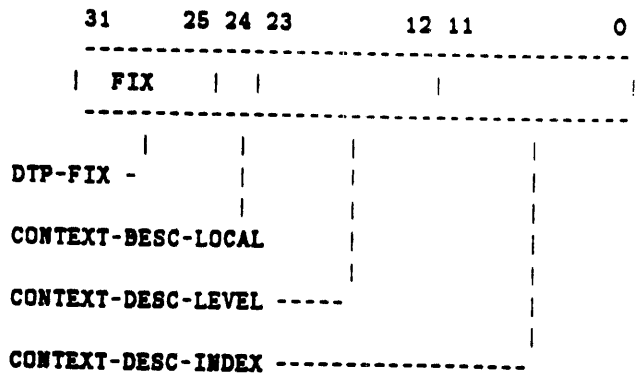


Fig. 12-5 Context Descriptor Format

CDDAAR *list* (Class 4 - miscop 274)

Takes the **CDR** of the **CDR** of the **CAR** of the **CAR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CAR**, the **CAR** of its **CAR**, or the **CDR** of the **CAR** of its **CAR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDDADR *list* (Class 4 - miscop 275)

Takes the **CDR** of the **CDR** of the **CAR** of the **CDR** of *list* and leaves the result in the destination. Signals an **ARGTYP** trap if *list*, its **CDR**, the **CAR** of its **CDR**, or the **CDR** of the **CAR** of its **CDR** is not a list or any is a disallowed **CAR** or **CDR** of a symbol or number.

CDDDAR (Class 4 - miscop 276)

list Takes the **CDR** *of the* **CDR** *of the* **CDR** *of the* **CAR** *of list and leaves the result in the destination. Signals an A*

CDDDDR (Class 4 - miscop 277)

list Takes the **CDR** *of the* **CDR** *of the* **CDR** *of the* **CDR** *of list and leaves the result in the destination. Signals an A*

%LOAD-FROM-HIGHER-CONTEXT *CONTEXTDESC* (Class 4 - miscop 300)

CONTEXTDESC is a context descriptor, which is a fixnum where fields are interpreted as shown in *Fig. 12-5*.

If **CONTEXT-DESC-LOCAL** is zero access an argument, if one access a local in the selected context. **CONTEXT-DESC-LEVEL** is the number of contexts to go up (unsigned). This identifies the context to access. **CONTEXT-DESC-INDEX** is the index of the argument or local in that context. The same format of context descriptor is used in **%LOCATE-IN-HIGHER-CONTEXT** and **%STORE-IN-HIGHER-CONTEXT** instructions.

Loads the argument or local from a higher lexical context as specified by *CONTEXTDESC*. Leaves the result in the destination.

%LOCATE-IN-HIGHER-CONTEXT *CONTEXTDESC* (Class 4 - miscop 301)

CONTEXTDESC is a context descriptor as shown in *Fig. 12-5* and described above. The result is a locative to the argument or local specified by the context descriptor.

%STORE-IN-HIGHER-CONTEXT *VALUE, ENVPTR* (Class 4 - miscop 302)

TI Internal Data

CONTEXTDESC is a context descriptor as shown in *Fig. 12-5* and described above. *VALUE* is stored into the argument or local specified by the context descriptor. Returns a locative to the modified cell as the result ** ?seems to via RPLACA? **.

%DATA ***303 obj*** (Class 4 - miscop TYPE)

Result is the data type code of *obj* as a fixnum

%POINTER *obj* (Class 4 - miscop 304)

Result is the pointer field of *obj* as a fixnum. Note: May be negative.

%MAKE-REST-ARG-SAFE (Class 4 - miscop 305)
;;;; DANGER: Cannot find the code for this one!!!! ;;;;

%PERMIT-TAIL-RECURSION (Class 4 - miscop 306)

Clears the UNSAFE-REST-ARG flag in the ENTRY-STATE word of the active call block. This allows tail recursion if the TAIL-RECURSION flag is T and the function bound no specials.

INTERNAL-FLOAT *NUMBER* (Class 4 - miscop 307)

Same as FLOAT.

%MAKE-POINTER *DTP, ADDRESS* (Class 4 - miscop 310)

Result is a Lisp object constructed from *DTP* and *ADDRESS*. *DTP* is the data type code of the object to be constructed (should be a fixnum but is not checked). The result is *ADDRESS* with its data type field replaced by *DTP*. If *ADDRESS* is not a pointer data type this instruction can produce a pointer that breaks the garbage collector: excercise extreme caution.

%SPREAD *LIST* (Class 4 - miscop 311)

Destination must be D-PDL or D-LAST. The elements of *LIST* are pushed onto the stack (appropriately for function args). If destination is D-LAST, the last element activates the call. If *LIST* is not a list an **ARGTYP** trap is signalled. May also signal **STACK-FRAME-TOO-LARGE** if the stack frame grows to exceed the limit of 256 locations.

%P-STORE-CONTENTS *POINTER, X* (Class 4 - miscop 312)

Replaces the contents of the cell pointed to by *POINTER* with *X*. Does not alter the cell's CDR code. Result is *X*.

%LOGLDB ***ppss, word*** (Class 4 - miscop 313)

Ppss is a field specifier as in LDB. ***word*** is uninterpreted 32 bits. Can load a field up to 25and return it as a fixnum. The result may be negative if the field size is 25. Signals **ARGTYP** if ***ppss*** is not a fixnum or ***ppss*** specifies a field more than 25bits wide.

%LOGDPB ***value, ppss, word*** (Class 4 - miscop 314)

Ppss is a field specifier as in LDB. The low order SIZE bits of ***value*** replace the field of the same size in word. Always returns a fixnum. Does not complain about loading/clobbering the sign; in fact, does not trap at all. *** should be symetric with %LOGLDB ***

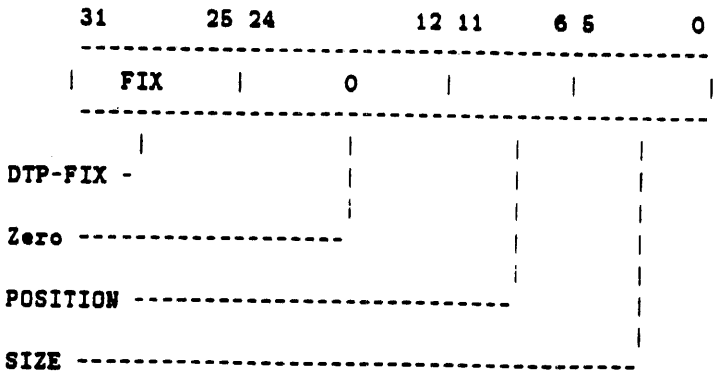


Fig. 12-6 Field Specifier Format

LDB *ppss, num* (Class 4 - miscop 315)

Ppss is a field specifier which is a fixnum with the low 6 bits specifying the size of the field (in bits) and the next 6 bits specifying the position of the field as in *Fig. 12-6*.

The field starts with the bit at POSITION and continues for SIZE bits. For example, the low order 8 bits of a word is specified as 0010_8. The high 7 bits of a 32 bit word are 3107_8. This is called "PPSS" because when expressed in octal, two digits give the position ("PP") and two digits give the size ("SS").

Extracts the field specified by *ppss* from *num* and return as a fixnum. *Num* must be a fixnum or a bignum or an **ARGTYP** trap is signalled. Also signals an **ARGTYP** trap if the field specifier specifies a field whose width is greater than 25-1 bits. Always returns a positive fixnum.

DPB *value, ppss, num* (Class 4 - miscop 316)

Inverse of LDB. The low order SIZE bits of *value* replace the field of the same size in *num*. Never changes the sign of the quantity DPB'ed into. If POSITION is above the current size of *num*, the quantity is sign extended until it is long enough to accommodate the DPB. Always returns either a fixnum or a bignum.

Value must be a fixnum or an **ARGTYP** error is signalled. *Ppss* must be a fixnum and specify a field of less than 25bits. otherwise an **ARGTYP** error is signalled. **ARGTYP** is also signalled unless *num* is a fixnum or a bignum.

%P-STORE-TAG-AND-POINTER *pointer, misc-fields, pointer-field* (Class 4 - miscop 317)

Builds a pointer by depositing the data type and CDR code from the low order part of *misc-fields* into the data type and CDR code fields of *pointer-field*. Then stores the newly built pointer into the cell addressed by *pointer*. Does not do **GC-WRITE-TEST**. Result is the symbol **NIL**.

Pointer and *pointer-field* may be of any type. *Misc-fields* must be a fixnum or an **ARGTYP** trap is signalled. See warning on %MAKE-POINTER.

GET *plist, property-name* (Class 4 - miscop 320)

Looks up *plist's property-name* property. If the property is found, the result is the value that property. Otherwise, the result is the symbol **NIL**. Returns **NIL** if *plist* does not have a property list or is of a type that does not have properties.

Handles objects of type symbol, list, locative, and instance. See section on instance invoke.

TI Internal Data

GETL *plist, indicator-list* (Class 4 - miscop 321)

Same as GET but looks for any property on *indicator-list*. If it finds any indicator then it returns the rest of the property list starting with the first indicator found.

ASSQ *x, alist* (Class 4 - miscop 322)

Alist is an association list where each element is a pair where the **CAR** is the tag and the **CDR** is the associated value. **ASSQ** searches *alist* by comparing the **CAR** of each element for being **EQ** to *x*. Returns the **CDR** of the matching element if a match is found or the symbol **NIL** if no match is found. Signals **ARGTYP** if *alist* is not a list of lists.

LAST *list* (Class 4 - miscop 323)

Returns the last element (last CAR) of *list*. Signals **ARGTYP** if *list* is not a list-like object. This CDRs down *list* and returns the **CAR** of the first pair whose **CDR** is **NIL**.

LENGTH *list* (Class 4 - miscop 324)

Returns the length of *list*. Signals **ARGTYP** if *list* is not a list-like object.

+ *n* (Class 4 - miscop 325)

Same as (PLUS n 1). Increments *n*. *N* must be a number or an **ARGTYP** trap is signalled.

- *n* (Class 4 - miscop 326)

Same as (DIFFERENCE n 1). Decrements *n*. *N* must be a number or an **ARGTYP** trap is signalled.

RPLACA *cons, x* (Class 4 - miscop 327)

Replace the **CAR** of *cons* with *x*. *Cons* must be a list or locative or an **ARGTYP** trap is signalled. Returns the altered *cons*. Does not recopy *cons*.

RPLACD *cons, x* (Class 4 - miscop 330)

Replace the **CDR** of *cons* with *x*. *Cons* must be a list, symbol if **CDR-SYM-MODE** allows, or locative, otherwise an **ARGTYP** trap is signalled. Returns the altered *cons*. Does not recopy *cons*. This handles the tricky hackery of altering a CDR-coded list. **RPLACD**'ing a locative is the same as **RPLACA**'ing it. **CDR-SYM-MODE** allows **RPLACD** to smash the property list of the symbol. Can also signal **BAD-CDR-CODE**.

ZEROP *number* (Class 4 - miscop 331)

Returns the symbol **T** if *number* is equal to the zero of its type; otherwise returns the symbol **NIL**. Signals **ARGTYP** if *number* is not numeric.

SET *symbol, x* (Class 4 - miscop 332)

Sets the value cell of *symbol* to value *x*. Signals **ARGTYP** if *symbol* is not a symbol. In addition, signals **ARGTYP** if *symbol* is **NIL** since it is illegal to set **NIL**.

FIXP *X* (Class 4 - miscop 333)

Returns the symbol **T** if *X* is a fixnum or a bignum. Otherwise returns the symbol **NIL**.

FLOATP *X* (Class 4 - miscop 334)

Returns the symbol T if *X* is a small flonum or a flonum. Otherwise returns the symbol **NIL**.

EQUAL *X*, *Y* (Class 4 - miscop 335)

Returns the symbol T if *X* and *Y* are similar (isomorphic) objects (cf. **EQ**). Two numbers are equal if they have the same value and type. Two conses are equal if their CARs are equal and their CDRs are equal. Two strings are equal if they have the same length, and the characters composing them are the same (see **STRING-EQUAL**). All other objects are equal if and only if they are **EQ**.

%SET-SELF-MAPPING-TABLE *MAPPING-TABLE* (Class 4 - miscop 336)

Set **SELF-MAPPING-TABLE** to *MAPPING-TABLE* and set the bit in the open call block saying we are providing it. Destination should be either D-IGNORE or D-LAST *** not clear what happens with other dests ***. Won't set this flag if the function being called is a symbol because can't be sure what mapping table it wants *** I think this means that the compiler can't tell what's going on and this instruction is badly applied?***.

PDL-WORD *N* (Class 4 - miscop 337)

Returns the value of the word on the PDL that is *N* below the PDL-Pointer after *N* has been popped. *** This should be %PDL-WORD since it does not check that *N* is an integer or that the selected word is in the PDL buffer. *** Assumes that the selected word is in the PDL buffer: never fetches the word from memory.

FALSE (Class 4 - miscop 340)

Result is the symbol **NIL**. *** Eventually flush this, after everything recompiled. New compiled code stopped using this as of 98. ***

TRUE (Class 4 - miscop 341)

Result is the symbol T. *** Eventually flush this, after everything recompiled. New compiled code stopped using this as of 87.27. ***

NOT *X* (Class 4 - miscop 342)

Also NULL. Result is the symbol T if *X* is the symbol **NIL**. Otherwise, the result is the symbol **NIL**.

NULL *X* (Class 4 - miscop 342)

Also NOT. Result is the symbol T if *X* is the symbol **NIL**. Otherwise, the result is the symbol T.

ATOM *X* (Class 4 - miscop 343)

Result is the symbol **NIL** if *X* is a list. Otherwise, the result is the symbol T.

ODDP *NUMBER* (Class 4 - miscop 344)

Result is the symbol T if *NUMBER* is odd. Otherwise, the result is the symbol **NIL**. *NUMBER* must be numeric or an **ARGTYP** trap is signalled.

EVENP *NUMBER* (Class 4 - miscop 345)

Result is the symbol **T** if *NUMBER* is even. Otherwise, the result is the symbol **NIL**. *NUMBER* must be numeric or an **ARGTYP** trap is signalled.

%HALT (Class 4 - miscop 346)

Halts the processor. Is intended to be continuable from the hardware debugger. This should not normally be used to shutdown the system.

GET-PNAME *SYMBOL* (Class 4 - miscop 347)

Returns the print name of *SYMBOL*. The object returned is an array pointer. *SYMBOL* must be a symbol or an **ARGTYP** trap is signalled.

LSH *N, NBITS* (Class 4 - miscop 350)

Logical shift of *N* to the left by *NBITS* bit positions. If *NBITS* is negative, the shift is to the right by (ABS *NBITS*). Zero bits are shifted in at right (left). *N* and *NBITS* must be fixnums. otherwise an **ARGTYP** trap will be signalled.

ROT *N, NBITS* (Class 4 - miscop 351)

Returns *N* rotated left *NBITS* bits if *NBITS* is positive or zero, or *N* rotated right (ABS *NBITS*) if *NBITS* is negative. The rotation considers *N* as a 25-bit number (cf. MACLISP ROT function). *N* and *NBITS* must be fixnums. otherwise an **ARGTYP** trap will be signalled.

***BOOLE** *FN, ARG1, ARG2* (Class 4 - miscop 352).

Performs a boolean function specified by *FN* on the other two arguments. *FN* must be a fixnum. otherwise an **ARGTYP** trap will be signalled. *ARG1* and *ARG2* must each be either a fixnum or a bignum. otherwise an **ARGTYP** trap will be signalled. *FN* specifies the boolean function to be performed. If the binary representation of *FN* is *abcd* (with *a* being the most significant bit) then the truth table for the boolean operation is as follows:

```
          y
      |  0   1
------------------
  0   |  a   c
x     |
  1   |  b   d
```

NUMBERP *X* (Class 4 - miscop 353)

Returns the symbol **T** if *X* is a number, otherwise returns the symbol **NIL**.

PLUSP *NUMBER* (Class 4 - miscop 354)

Returns the symbol **T** if *NUMBER* is positive number, strictly greater than zero; otherwise, returns the symbol **NIL**. Signals **ARGTYP** if *NUMBER* is not a number.

MINUSP *NUMBER* (Class 4 - miscop 355)

Returns the symbol **T** if *NUMBER* is a negative number, strictly less than zero; otherwise, returns the symbol **NIL**. Signals **ARGTYP** if *NUMBER* is not a number.

\ *X*, *Y* (Class 4 - miscop 356)

Returns the remainder of *X* divided by *Y*. *X* and *Y* must both be integers (each may be either a fixnum or a bignum), otherwise an **ARGTYP** trap will be signalled.

MINUS *NUMBER* (Class 4 - miscop 357)

Returns the negative of *NUMBER*. *NUMBER* must be a number or an **ARGTYP** trap will be signalled.

%SXHASH-STRING *string, character-mask* (Class 4 - miscop 360)

Compute the hash code for *string*. Each character is logically AND'ed with *character-mask* before being used in the hash computation. Returns as a fixnum the hash code of the array.

VALUE-CELL-LOCATION *SYMBOL* (Class 4 - miscop 361)

Returns a locative pointer to *SYMBOL*'s internal value cell. There may also be an external value cell. *SYMBOL* must be a symbol or an **ARGTYP** trap will be signalled.

FUNCTION-CELL-LOCATION *SYMBOL* (Class 4 - miscop 362)

Returns a locative pointer to *SYMBOL*'s function cell. If *SYMBOL* is not a symbol an **ARGTYP** trap will be signalled.

PROPERTY-CELL-LOCATION *SYMBOL* (Class 4 - miscop 363)

Returns a locative pointer to the location of *SYMBOL*'s property-list cell. *SYMBOL* must be a symbol or an **ARGTYP** trap will be signalled.

NCONS *X* (Class 4 - miscop 364)

Constructs a list cell (cons) that has a CAR of *X* and a CDR of **NIL**. The storage is allocated in the default consing area specified by the value of the variable **DEFAULT-CONS-AREA**.

NCONS-IN-AREA *X, area* (Class 4 - miscop 365)

Like **NCONS** but the storage is allocated in the area specified by *area*. Signals **ARGTYP** if *area* is not a fixnum or a symbol with a fixnum in its value cell.

CONS *X, Y* (Class 4 - miscop 366)

Constructs a list cell (cons) that has a CAR of *X* and a CDR of *Y*. The storage is allocated in the default consing area specifed by the value of the variable **DEFAULT-CONS-AREA**.

CONS-IN-AREA *X, Y, area* (Class 4 - miscop 367)

Like **CONS** but the storage is allocated in the area specified by *area*. Signals **ARGTYP** if *area* is not a fixnum or a symbol with a fixnum in its value cell.

XCONS *X, Y* (Class 4 - miscop 370)

Reverse **CONS**. Same as (CONS *Y X*).

XCONS-IN-AREA *X, Y, area* (Class 4 - miscop 371)

Reverse **CONS-IN-AREA**. Same as (CONS-IN-AREA *X Y area*).

TI Internal Data

%SPREAD-N *LIST, N* (Class 4 - miscop 372)

Similar to %SPREAD. Pushes the first *N* elements of *LIST* onto the stack. If the destination is D-LAST, activate the call block. Always attempt to take N **CDR**'s of *LIST*, even if it isn't long enough. If appropriate **CDR-SYN-MODE** is set, will continue pushing **NIL**s, otherwise will signal an **ARGTYP** trap. Can also signal **STACK-FRAME-TOO-LARGE** if the stack frame would grow beyond the limit of 256 stack locations.

SYMEVAL *SYMBOL* (Class 4 - miscop 373)

SYMEVAL is the basic primitive for retrieving a symbol's value. The current value of *SYMBOL* is returned. If the symbol is unbound, then a **TRANS-TRAP** error is signalled. If *SYMBOL* is not a symbol, **ARGTYP** is signalled.

POP-N-FROM-UNDER-N *num-pops, num-to-keep* (Class 4 - miscop 374)

Has the same effect as popping and saving the top *num-to-keep* somewhere, then popping the next *num-pops*, and then pushing back the saved *num-to-keep* values. Actually works more like copying the top *num-to-keep* stack locations down by *NUM-POPS* locations. Also removes any open call blocks in that region of the stack.

GET-LEXICAL-VALUE-CELL *env-list, symbol-cell-location* (Class 4 - miscop 375)

Could have been defined as:

```
(Defun GET-LEXICAL-VALUE-CELL (env-list symbol-cell-location)
(GET-LOCATION-OR-NIL (LOCF env-list) symbol-cell-location))
```

except this runs much faster when *env-list* is a stack list in the PDL buffer.

%CALL-MULT-VALUE *FUNCTION, NUM-VALUES* (Class 4 - miscop 376)

Similar to a CALL instruction in that it opens a call block to call *FUNCTION*. The number of values expected as the result is *NUM-VALUES*. Before pushing the call block, reserves a block of *NUM-VALUES* words on the stack for the returned values. Then pushes an ADI that indicates that a multiple value block is present and gives its size. See See section on (.ADI). When the last argument is pushed to D-LAST, *FUNCTION* is activated. When function returns, the values are left on the stack.

%CALLO-MULT-VALUE *VALUES, FUNCTION* (Class 4 - miscop 377)

This is to **%CALL-MULT-VALUE** as **CALLO** is to **CALL**.

%RETURN-2 *VAL1, VAL2* (Class 4 - miscop 400)

Return *VAL1* and *VAL2* as the values of this function; then this function returns. No more values are returned than the caller is expecting.

%RETURN-3 *VAL1, VAL2, VAL3* (Class 4 - miscop 401)

Return *VAL1*, *VAL2*, and *VAL3* as the values of this function; then this function returns. No more values are returned than the caller is expecting.

%RETURN-N *VAL1, VAL2 ... VALN, N* (Class 4 - miscop 402)

The number of values to return, *N*, is the last argument. *VAL1* through *VALN* are returned as the values of this function. No more values are returned then the number expected by the caller. When all values have been processed, this function returns.

RETURN-NEXT-VALUE *X* (Class 4 - miscop 403)

Return *X* as a value of the current function. If the caller is expecting more values, this becomes the next of them, otherwise it is ignored.

RETURN-LIST *VALUES* (Class 4 - miscop 404)

Return the elements of *VALUES* as the values of the current function. It is *always* called with D-RETURN, and so always returns from the current function. The caller never receives more values than it is expecting.

UNBIND-TO-INDEX-UNDER-N *N* (Class 4 - miscop 405)

This removes an index from the PDL, *N* under the PDL-pointer after popping *N*. This index is then used as a Special-PDL index and the Special-PDL is unwound to the level of that index. The regular PDL is copied down to fill in the hole where the index was removed. *** this is wierd ***

BIND *POINTER, X* (Class 4 - miscop 406)

Bind any location to a specified value. Adds the binding to the current stack-frame. This allows you to bind cells other than value cells and to do conditional binding.

UNUSED407 (Class 4 - miscop 407)

Not currently used. Was **%MAKE-LEXICAL-CLOSURE**.

MEMQ *ITEM, LIST* (Class 4 - miscop 410)

If *ITEM* is one of the elements of *LIST*, the sublist of *LIST* beginning with the first occurrence of *ITEM* is returned. If *ITEM* is not in *LIST*, the symbol **NIL** is returned. The comparison is made by **EQ**. Because **MEMQ** returns **NIL** if *ITEM* is not found and something non-**NIL** if it is, it is often used as a predicate.

Note that the value returned by **MEMQ** is **EQ** to the portion of *LIST* beginning with *ITEM*. Thus **RPLACA** on the result of **MEMQ** may be used, if **MEMQ** did not return **NIL**.

N-< *NUM1, NUM2* (Class 4 - miscop 411)

Primitive, two argument **LESSP**. Compares *NUM1* to *NUM2*. Returns the symbol **T** if *NUM1* is less than *NUM2*; returns the symbol **NIL** otherwise. *NUM1* and *NUM2* must both be numeric types or an **ARGTYP** error is signalled.

N-> *NUM1, NUM2* (Class 4 - miscop 412)

Primitive, two argument GREATERP. Compares *NUM1* to *NUM2*. Returns the symbol **T** if *NUM1* is less than *NUM2*; returns the symbol **NIL** otherwise. *NUM1* and *NUM2* must both be numeric types or an **ARGTYP** error is signalled.

N-= *NUM1, NUM2* (Class 4 - miscop 413)

Returns the symbol **T** if *NUM1* and *NUM2* are numerically equal. An integer can be equal to a flonum. *NUM1* and *NUM2* must both be numeric types or an **ARGTYP** error is signalled.

TI Internal Data

INTERNAL-CHAR-EQUAL *CH1, CH2* (Class 4 - miscop 414)

Compare two characters that are either fixnums or DTP-CHARACTER objects. If the character code part of the two are identical, the symbol T is returned. Otherwise, if **ALPHABETIC-CASE-AFFECTS-STRING-COMPARISON** is non-**NIL**, the symbol **NIL** is return. If the characters are different and case doesn't matter, the symbol T is returned if one is the alphabetic uppercase of the other, otherwise the symbol **NIL** is returned

%STRING-SEARCH-CHAR *CHAR, STRING, START, END* (Class 4 - miscop 415)

Search *STRING* from *START* to *END* for *CHAR*. Character comparison is as in **INTERNAL-CHAR-EQUAL**. *STRING* should be a numeric (or ART-STRING or ART-FAT-STRING) array. *CHAR* must be a character or fixnum. *START* and *END* must be fixnums or an **ARGTYP** trap is signalled. Both *START* and *END* must be in bounds on *STRING*.

Searching is always forward. Returns **NIL** if *START* is greater than *END*.

%STRING-EQUAL *STRING1, INDEX1, STRING2, INDEX2, COUNT* (Class 4 - miscop 416)

Returns the symbol T if *COUNT* characters of *STRING1* starting at *INDEX1* match those of *STRING2* starting at *INDEX2*. The comparison ignores case if **ALPHABETIC-CASE-AFFECTS-STRING-COMPARISON** is **NIL**.

NTH *N, LIST* (Class 4 - miscop 417)

Returns the *N*th element of *LIST*. Counting starts from 0, so element 0 is the CAR and element 1 is the CADR, etc.

NTHCDR *N, LIST* (Class 4 - miscop 420)

Discards *N* elements from *LIST*. Same as performing CDR *N* times.

M-+ *NUM1, NUM2* (Class 4 - miscop 421)

MISC version of + (**PLUS**). Adds *NUM1* to *NUM2* to produce a result. *NUM1* and *NUM2* must both be numbers or an **ARGTYP** error is signalled. The result will be a number.

M-- *NUM1, NUM2* (Class 4 - miscop 422)

MISC version of - (**DIFFERENCE**). Subtracts *NUM2* from *NUM1* to produce the result. *NUM1* and *NUM2* must both be numbers or an **ARGTYP** error is signalled. The result will be a number.

M-* *NUM1, NUM2* (Class 4 - miscop 423)

MISC version of * (**TIMES**). Multiplies *NUM1* and *NUM2* to produce the result. *NUM1* and *NUM2* must both be numbers or an **ARGTYP** error is signalled. The result will be a number.

M-// *NUM1, NUM2* (Class 4 - miscop 424)

MISC version of // (**QUOTIENT**). Divides *NUM1* by *NUM2* to produce the result. *NUM1* and *NUM2* must both be numbers or an **ARGTYP** error is signalled. The result will be a number. Also signals **DIVIDE-BY-ZERO** if *NUM2* is zero.

M-LOGAND *NUM1, NUM2* (Class 4 - miscop 425)

MISC version of **LOGAND**. Result is the logical AND of *NUM1* and *NUM2*. *NUM1* and *NUM2* must both be integer numbers (Fixnum or Bignum) or an **ARGTYP** error is signalled. The result will be an integer number.

M-LOGXOR *NUM1, NUM2* (Class 4 - miscop 426)

MISC version of **LOGXOR**. Result is the logical exclusive-OR of *NUM1* and *NUM2*. *NUM1* and *NUM2* must both be integer numbers (either Fixnum or Bignum) or an **ARGTYP** error is signalled. The result will be an integer number.

M-LOGIOR *NUM1, NUM2* (Class 4 - miscop 427)

MISC version of **LOGIOR**. Result is the logical inclusive-OR of *NUM1* and *NUM2*. *NUM1* and *NUM2* must both be integer numbers (either FIXNUM or BIGNUM) or an **ARGTYP** error is signalled. The result will be an integer number.

ARRAY-LEADER *ARRAY, INDEX* (Class 4 - miscop 430)

Gets the *INDEX*'th element of the leader of *ARRAY*. *ARRAY* must be an array or an **ARGTYP** error is signalled. If *ARRAY* does not have a leader, ARRAY-HAS-NO-LEADER is signalled. *INDEX* must be a fixnum or an **ARGTYP** error is signalled. If *INDEX* is greater than or equal to the length of the leader, **SUBSCRIPT-OOB** is signalled.

STORE-ARRAY-LEADER *VALUE, ARRAY, INDEX* (Class 4 - miscop 431)

Store *VALUE* into the *INDEX*'th element of the leader of *ARRAY*. *ARRAY* must be an array or an **ARGTYP** error is signalled. If *ARRAY* does not have a leader, ARRAY-HAS-NO-LEADER is signalled. *INDEX* must be a fixnum or an **ARGTYP** error is signalled. If *INDEX* is greater than or equal to the length of the leader, **SUBSCRIPT-OOB** is signalled.

GET-LIST-POINTER-INTO-ARRAY *ignore* (Class 4 - miscop 432)

Ignore the argument and return a list pointer to the last array element referenced. The last array referenced element is remember in a register and kept in the stack group. The last array referenced must be an ART-Q-LIST array or **ARGTYP** is signalled.

ARRAY-PUSH *ARRAY, VALUE* (Class 4 - miscop 433)

Add *VALUE* as an element at the end of *ARRAY*. The fill pointer (leader element 0) is the index of the next element to be added. Returns **NIL** and doesn't update the fill pointer if the array is full, otherwise returns the index of the element written. Does not automatically increase the size of the array like **ARRAY-PUSH-EXTEND**.

APPLY *FN, ARGS* (Class 4 - miscop 434)

Call *FN* on *ARGS*. *ARGS* are passed to *FN* spread or combination of spread and a rest as *FN* expects them. *FN* can be any functional object.

%MAKE-LIST *INITIAL-VALUE, AREA, LENGTH* (Class 4 - miscop 435)

Construct a CDR-coded list of *INITIAL-VALUE, LENGTH* elements long in *AREA*.

LIST *&REST ELEMENTS* (Class 4 - miscop 436)

Return a list in **DEFAULT-CONSING-AREA** of the arguments. Actually treated like 63 optional args rather than a rest arg.

LIST* *FIRST, &REST ELEMENTS* (Class 4 - miscop 437)

Like **LIST** except that the last cons of the constructed list is "dotted". The last argument to **LIST*** becomes the last CDR of the constructed list.

Example:

```
(LIST* 'A 'B 'C) => (A B . C)
```

which is the same as produced by

```
(CONS 'A (CONS 'B 'C))
```

LIST* of a single argument is just that argument, *FIRST*; no list is consed. This is actually treated like 63 optional args rather than a rest arg.

LIST-IN-AREA *AREA &REST ELEMENTS* (Class 4 - miscop 440)

Returns a list of the *ELEMENTS* in *AREA*. This is actually treated like 63 optional args rather than a rest arg.

LIST*-IN-AREA *AREA, FIRST, &REST ELEMENTS* (Class 4 - miscop 441)

Like **LIST*-IN-AREA** is to **LIST-IN-AREA** as **LIST*** is to **LIST**. This is actually treated like 63 optional args rather than a rest arg.

LOCATE-IN-INSTANCE *instance, symbol* (Class 4 - miscop 442)

Returns a locative to te slot in *instance* for the instance variable *symbol*. *** errors? ***

%P-CDR-CODE *POINTER* (Class 4 - miscop 443)

Returns the CDR code value of the word addressed by *POINTER*. This is a number from 0 to 3. The values have standard names which are **CDR-NEXT**, **CDR-NIL**, **CDR-NORMAL**, and **CDR-ERROR**. *POINTER*'s data type is ignored, it can even be fixnum.

%P-DATA *444 POINTER* (Class 4 - miscop TYPE)

Returns the data type field of the word addressed by *POINTER*. This does not follow forwarding pointers nor complain about illegal data types. *POINTER*'s data type is ignored, it can even be fixnum.

%P-POINTER *POINTER* (Class 4 - miscop 445)

Returns the pointer field of the word addressed by *POINTER*. This does not follow forwarding pointers nor complain about illegal data types. *POINTER*'s data type is ignored, it can even be fixnum.

%PAGE-TRACE *table* (Class 4 - miscop 446)

Enable or disable swap in and swap out metering. *Table* is either a wired down array or **NIL**. If **NIL** the page metering is disabled. Otherwise, it is an array which has been wired down. On each swap in or swap out event, a 4-word entry is added in the array. The words are shown in *Table 12-11*.

THROW-N *TAG, &REST VALUES-AND-COUNT* (Class 4 - miscop 447)

Throw passing *COUNT* values. *COUNT* is the last argument. See ***THROW**.

Word	Meaning
0	Microsecond clock value
1	Virtual Address
2	Miscellany:
	bit 31: swap out flag
	bit 30: stack-group-switch flag
	bit 29: transport flag
	bit 28: scavenge flag
	bit 15-0: micro-PC
3	Current function (**PDLON-AP**)

Table 12-11 Page Trace Entry

%P-STORE-CDR-CODE *POINTER, CDR-CODE* (Class 4 - miscop 450)

Store *CDR-CODE* into the CDR-code field of the word addressed by *POINTER*. *CDR-CODE* is a number from 0 to 3 *** is it checked? truncated? *** *POINTER*'s data type is ignored; it can even be fixnum, so this can be dangerous unless used with extreme care.

%P-STORE-DATA *451 POINTER, DATA TYPE* (Class 4 - miscop TYPE)

Store *DATA TYPE* into the data type field of the word addressed by *POINTER*. *DATA TYPE* is a value in the range 0 to 31. *POINTER*'s data type is ignored; it can even be a fixnum, so this can be dangerous unless used with extreme care.

%P-STORE-POINTER *POINTER, POINTER-TO-STORE* (Class 4 - miscop 452)

Store *POINTER-TO-STORE* into the pointer field of the word addressed by *POINTER*. *POINTER*'s data type is ignored: it can even be fixnum, so this can be dangerous unless used with extreme care.

FLOAT-EXPONENT *FLONUM* (Class 4 - miscop 453)

Return as a fixnum the exponent of the floating point number, *FLONUM*. **I think**

FLOAT-FRACTION *flonum* (Class 4 - miscop 454)

Return *flonum* modified to contain 0 as its exponent. The result is either zero or has absolute value at least 1/2 and less than one.

SCALE-FLOAT *flonum, integer* (Class 4 - miscop 455)

Return a FLONUM like *flonum* but with *integer* added to its exponent.

%CATCH-OPEN *restart-pc* (Class 4 - miscop 456)

Open a call block for the function *CATCH. This is a catch block that will catch throws to the catch tag which is pushed as the first argument to this block. There are two ADI on the block produced by this instruction, one recording the binding stack level and the other recording *restart-pc* as the restart PC in this function if this catch is thrown to.

%CATCH-OPEN-MV *restart-pc, num-vals* (Class 4 - miscop 457)

Like **%CATCH-OPEN** but expects multiple values. Allocates space for a multiple value block for *num-vals* values and adds an ADI for multiple value block.

TI Internal Data

Dest	Operation
0	FLOOR
1	CEIL
2	TRUNC
3	ROUND

Table 12-11 Internal Floor 1 Decode

INTERNAL-FLOOR-1 *DIVIDEND, DIVISOR* (Class 4 - miscop 460)

The destination field of this instruction is used to select the operation which is shown in *Table 12-11*. All divide, and all round to a fixnum result in some way.

%DIV *X, Y* (Class 4 - miscop 461)

Divide *X* by *Y* returning a rational number if both *X* and *Y* are integers. ;; otherwise ?? ;;

%FEXPR-CALL *function* (Class 4 - miscop 462)

Open a call block to *function*, indicating that it is a FEXPR call. Function entry will handle the spreading of the part of the final list argument to get enough spread args for *function*.

%FEXPR-CALL-MV *function, num-vals* (Class 4 - miscop 463)

%FEXPR-CALL that expects multiple values. Sets up a multiple value return block to receive *num-vals* values.

%FEXPR-CALL-MV-LIST *function* (Class 4 - miscop 464)

%FEXPR-CALL that expects multiple values. Sets up a multiple value list return to receive all returned values.

%CATCH-OPEN-MV-LIST *restart-pc* (Class 4 - miscop 465)

Open a ***CATCH** block that will receive values into a list. If it is thrown to, execution will resume at *restart-pc*. See ***CATCH** below.

***CATCH** *TAG &REST FORMS* (Class 4 - miscop 466)

Set up a tag, *TAG*, that a ***THROW** can throw to. If a ***THROW** with argument **EQ** to *TAG* is executed dynamically within *FORMS*, it returns immediately from the ***CATCH**, skipping the rest of the execution of *FORMS*. The second argument of ***THROW** is returned as the value of ***CATCH**.

%BLT *FROM-ADDRESS, TO-ADDRESS, COUNT, INCREMENT* (Class 4 - miscop 467)

Copy a block of memory, a word at a time, with no decoding, for untyped data. Use **%BLT-TYPED** for words which contain Lisp data types. The first word is copied from *FROM-ADDRESS* to *TO-ADDRESS*. *INCREMENT* is added to each address and then another word is copied, and so on. *COUNT* is number of words to copy.

***THROW** *TAG, VALUE* (Class 4 - miscop 470)

Return immediately from the innermost ***CATCH** that handles this *TAG*. The ***CATCH** returns *VALUE* as its value.

%XBUS-WRITE-SYNC *IO-ADDR, WORD, DELAY, SYNC-LOC, SYNC-MASK, SYNC-VAL* (Class 4 - miscop 471)

This instruction is not implemented. Any attempt to use it will signal an **UNIMPLEMENTED-HARDWARE** trap with XBUS as the missing hardware.

%P-LDB *ppss, pointer* (Class 4 - miscop 472)

Does **%LOGLDB** from the word pointed to by *pointer*. Does not interpret the data type of that word or follow forwarding pointers there. *Ppss* is a field specifier as in **LDB**. Returns the specified field of the word as a fixnum. The field may be up to 25bits wide. May return a negative value if the field is 25bits wide. Signals **ARGTYP** if *ppss* is not a fixnum or specifies a field wider than 25bits.

%P-DPB *VALUE, PPSS, POINTER* (Class 4 - miscop 473)

Store *VALUE* into byte *PPSS* in the word addressed by *POINTER*. *Ppss* is a field specifier as in **LDB**. This byte can include any of the bits in the word, and can overlap between the various fields normally used by Lisp. But it may not be more than 24 bits long. *POINTER*'s data type is ignored; it can even be fixnum, so this can be dangerous unless used with extreme care.

MASK-FIELD *ppss, fixnum* (Class 4 - miscop 474)

Return a number which is *fixnum* with all but the byte *ppss* replaced by zero. *ppss* is a field specifier as in **LDB**. *** returns a fixnum? is arg checked for fixnum? ***

%P-MASK-FIELD *ppss, pointer* (Class 4 - miscop 475)

Returns (**MASK-FIELD** *ppss* (**%P-POINTER** *pointer*)).

DEPOSIT-FIELD *value, ppss, fixnum* (Class 4 - miscop 476)

Return a number which in the byte *ppss* matches *value* and the rest matches *fixnum*. *ppss* is a field specifier as in **LDB**. *** type of result, errors? ***

%P-DEPOSIT-FIELD *value, ppss, pointer* (Class 4 - miscop 477)

Stores into the byte *ppss* of the word addressed by *pointer* from the same byte of *value* *** is this right?? ***. This byte can include any of the bits in the word, and can overlap between the various fields normally used by Lisp. For example, part of *value*'s data type field may be included. *pointer*'s data type is ignored; it can even be fixnum, so this can be dangerous unless used with extremem care.

COPY-ARRAY-CONTENTS *from, to* (Class 4 - miscop 500)

Copy all the elements of the array *from* into *to*. If *to* is longer than *from*, it is filled out with zeros (if a numeric array) or **NIL**s. If either array is multidimensional, its elements are used in the order they are stored in memory.

COPY-ARRAY-CONTENTS-AND-LEADER *from, to* (Class 4 - miscop 501)

Copy all the elements and leader slots of the array *from* into *to*. If *to* is longer than *from*, it is filled out with zeros (if a numeric array) or **NIL**s. If either array is multidimensional, its elements are used in the order they are stored in memory.

TI Internal Data

Dest	Operation
0	FLOOR
1	CEIL
2	TRUNC
3	ROUND

Table 12-11 Internal Floor 2 Decode

%FUNCTION-INSIDE-SELF (Class 4 - miscop 502)

Returns the functional part of **SELF**. If **SELF** is an instance, return the contents of the cell referenced by the **%INSTANCE-DESCRIPTOR-FUNCTION** slot of the instance descriptor. This is usually an funcallable hash array. If **SELF** is an entity or a closure, return the function from the closure. Otherwise, return **SELF**.

ARRAY-HAS-LEADER-P *array* (Class 4 - miscop 503)

Returns the symbol **T** if *array* has a leader. If *array* does not have a leader, the symbol **NIL** is returned. *array* must be an array or an **ARGTYP** error is signalled.

COPY-ARRAY-PORTION (Class 4 - miscop 504)
from-array, from-start, from-end, to-array, to-start, to-end

Copies specified elements of *from-array* into *to-array*. *From-start* and *from-end* are indices in *from-array* indicating the portion to copy. *To-start* and *to-end* are indicies in *to-array*. If the specified portion of *to-array* is longer, it is filled out with zeros (if *to-array* is a numeric array) or **NIL**s. If either array is multidimensional, its elements are used in the order they are stored in memory.

FIND-POSITION-IN-LIST *item, list* (Class 4 - miscop 505)

Looks down *list* for an element which is EQ to *item*, like **MEMQ**. However, it returns the numeric index in the list at which it found the first occurence of item, or the symbol **NIL** if it did not find it at all. The index returned is zero-based. *list* must be a list. *Item* may be any object.

%GET-SELF-MAPPING-TABLE *method-flavor-name* (Class 4 - miscop 506)

Method-flavor-name is a symbol for the flavor of the method for which to get a self mapping table. If **SELF** is not an instance, **NIL** is returned. If the mapping table is already the value of **SELF-MAPPING-TABLE**, it is returned. Otherwise, the table is located by searching the mapping table alist of the instance descriptor for **SELF** for the mapping table for **method-flavor-name** and returning the mapping table which is in the **CDDR** of it.

G-L-P *array* (Class 4 - miscop 507)

Return a list overlayed with the contents of *array*. *array* must be an array of type ART-Q-LIST.

INTERNAL-FLOOR-2 *dividend, divisor* (Class 4 - miscop 510)

The destination field of this instruction is used to select the operation which is shown in *Table 12-11*. All divide, and all round to a fixnum result in some way. Returns two values which are ***?***.

EQL *X, Y* (Class 4 - miscop 511)

When both arguments are numbers, true only if they are of the same type and have the same value; otherwise same as **EQ**. This function is for Common Lisp.

AR-1 *array, index* (Class 4 - miscop 512)

One dimensional array reference. Return element *index* of *array*. *Array* must be an array and *index* must be a fixnum, otherwise an **ARGTYP** error is signalled. If *index* is less than zero or greater than the largest index premissible for *array* then a **SUBSCRIPT-OOB** error is signalled. If *array* does not have exactly one dimension, **ARRAY-NUMBER-DIMENSIONS** is signalled. The type of the result depends on the type of *array* and the element stored at *index*.

AR-2 *array, sub1, sub2* (Class 4 - miscop 513)

Two dimensional array reference. Return element selected by subscripts *sub1* and *sub2* of *array*. *Array* must be an array and the subscripts fixnums, otherwise an **ARGTYP** error is signalled. If either *sub1* or *sub2* are less than zero or greater than the largest index premissible for that subscript of *array* then a **SUBSCRIPT-OOB** error is signalled. If *array* does not have exactly two dimensions, **ARRAY-NUMBER-DIMENSIONS** is signalled. The type of the result depends on the type of *array* and the element read from the array.

AR-3 *array, sub1, sub2, sub3* (Class 4 - miscop 514)

Three dimensional array reference. Return element selected by subscripts *sub1*, *sub2* and *sub3* of *array*. *Array* must be an array and the other arguments fixnums, otherwise an **ARGTYP** error is signalled. If any of *sub1*, *sub2* or *sub3* are less than zero or greater than the largest index premissible for that subscript of *array* then a **SUBSCRIPT-OOB** error is signalled. If *array* does not have exactly three dimensions, **ARRAY-NUMBER-DIMENSIONS** is signalled. The type of the result depends on the type of *array* and the element read from the array.

AS-1 *value, array, index* (Class 4 - miscop 515)

One dimensional array store. Store *value* into *array* at *index*. *Array* must be an array and the subscripts fixnums, otherwise an **ARGTYP** error is signalled. If either *sub1* or *sub2* are less than zero or greater than the largest index premissible for that subscript of *array* then a **SUBSCRIPT-OOB** error is signalled. If *array* does not have exactly two dimensions, **ARRAY-NUMBER-DIMENSIONS** is signalled. Returns *value*.

AS-2 *value, array, sub1, sub2* (Class 4 - miscop 516)

Two dimensional array store. Store *value* into *array* at *sub1*, *sub2*. *Array* must be an array and the subscripts fixnums, otherwise an **ARGTYP** error is signalled. If either *sub1* or *sub2* are less than zero or greater than the largest index premissible for that subscript of *array* then a **SUBSCRIPT-OOB** error is signalled. If *array* does not have exactly two dimensions, **ARRAY-NUMBER-DIMENSIONS** is signalled. Returns *value*.

AS-3 *value, array, sub1, sub2, sub3* (Class 4 - miscop 517)

Three dimensional array store. Store *value* into the *array* element selected by *sub1*, *sub2* and *sub3*. *Array* must be an array and the other arguments fixnums, otherwise an **ARGTYP** error is signalled. If any of *sub1*, *sub2* or *sub3* are less than zero or greater than the largest index premissible for that subscript of *array* then a **SUBSCRIPT-OOB** error is signalled. If *array* does not have exactly three dimensions. **ARRAY-NUMBER-DIMENSIONS** is signalled. Returns *value*.

TI Internal Data

%INSTANCE-REF *instance, index* (Class 4 - miscop 520)

Return the contents of slot *index* in *instance*. The lowest valid *index* is 1.

%INSTANCE-LOC *instance, index* (Class 4 - miscop 521)

Return a locative pointer to slot *index* in *instance*. The lowest valid *index* is 1.

%INSTANCE-SET *val, instance, index* (Class 4 - miscop 522)

Set contents of slot *index* in *instance* to *val*. The lowest valid *index* is 1.

%BINDING-INSTANCES *list-of-symbols* (Class 4 - miscop 523)

Returns a list of locatives which are alternately internal and external value cell pointers. One pair of pointers is placed on the list for each symbol in *list-of-symbols*. This is similar to closure except that it does not have a function parameter.

%EXTERNAL-VALUE-CELL *symbol* (Class 4 - miscop 524)

Returns a locative to whatever the value cell of *symbol* points to. If symbol is closure bound, this will be a locative to the external value cell. Does not check that the internal value cell contains an external value cell pointer.

%USING-BINDING-INSTANCES *binding-instances* (Class 4 - miscop 525)

Install the bindings in *binding-instances*. *Binding-instances* is a list of alternating internal and external value cell pointers as returned by **%BINDING-INSTANCES**. Binds the first of each pair to the second of each pair.

%GC-CONS-WORK *NQS* (Class 4 - miscop 526)

Use this to indicate to the microcoded GC support that you have done some consing. *NQS* is the number of Q's allocated. There is no need to do this if storage is allocated by the microcoded storage allocation routines.

%P-CONTENTS-OFFSET *pointer, offset* (Class 4 - miscop 527)

Returns the contents of the word *offset* beyond that addressed by *pointer*. This is not the same as what could be done with **%P-CONTENTS** and **%MAKE-POINTER-OFFSET** because it checks for a forwarding pointer in the word addressed by *pointer*. The idea is that *pointer* points at the beginning of a structure and *offset* is an offset within it.

%DISK-RESTORE *PARTITION-HIGH-16-BITS LOW-16-BITS* (Class 4 - miscop 530)

Restore a load partition. The partition to load is selected by concatenating the two arguments to form a 32-bit number that is interpreted as the 4 character partition name of a partition in the disk label. If the number is zero, the current band is used.

%DISK-SAVE (Class 4 - miscop 531)
MAIN-MEMORY-SIZE PARTITION-HIGH-16-BITS LOW-16-BITS

Save current memory contents in a load partition. The partition to write into is determined by concatenating the last two arguments to form a 32-bit number. This is matched against the partition names in the disk label to find the designated partition. If the number is zero, the current partition is used.

%ARGS-INFO *function* (Class 4 - miscop 532)

Returns a numeric argument descriptor (as described in section on FEF Layout) for *function*. Function may be any object meaningful as a function.

%OPEN-CALL-BLOCK *function, ADI-pairs, destination* (Class 4 - miscop 533)

Push a call block on the stack. for function *function*. *ADI-pairs* is the number of two-word ADI units you have already pushed. *Destination* is a numeric destination code, 0 through 3, which stands for D-IGNORE, D-PDL, D-LAST, or D-RETURN ;; order right? ;;. This works only in compiled code.

%PUSH *X* (Class 4 - miscop 534)

Push *X* onto the stack. Useful with **%OPEN-CALL-BLOCK**. You must make sure you have room on the stack with **%ASSURE-PDL-ROOM**, before you push words with **%PUSH**. This works only in compiled code.

%ACTIVATE-OPEN-CALL-BLOCK (Class 4 - miscop 535)

Actually call the function in a call block you made with **%OPEN-CALL-BLOCK**. This is done after pushing the arguments with **%PUSH**.

%ASSURE-PDL-ROOM *room* (Class 4 - miscop 536)

Make sure there is room in the PDL buffer to do *room* more pushes in this active frame. Will signal **STACK-FRAME-TOO-LARGE** if the current size of the frame plus *room* is larger than about 248. Note that the maximum stack frame size is limited to 255.

STACK-GROUP-RETURN *X* (Class 4 - miscop 537)

Resume the stack group which invoked this one. with *X* as the argument. Does not change which stack group is recorded as that one's resumer.

AS-2-REVERSE *value, array, index2, index1* (Class 4 - miscop 540)

Store *value* into *array*, optionally reversing the indices. While arrays are stored with the first index varying fastest, this is the same as **ASET**. When arrays are stored with the last index varying fastest, this uses *index1* as the first index even though it is the last argument.

%MAKE-STACK-LIST *N* (Class 4 - miscop 541)

Pushes *N* − 1 **NIL**'s onto the stack with CDR-Next and one more **NIL** with CDR-NIL. Returns a list pointer to the list. Since this pushes a bunch of words onto the stack, it may interfere with the use of the stack for expression evaluation. This does not check for PDL room. Should probably issue **%ASSURE-PDL-ROOM** before this.

STACK-GROUP-RESUME *SG, X* (Class 4 - miscop 542)

Resume stack group *SG* with *X* as the argument. See section on multiprocessing.

%CALL-MULT-VALUE-LIST *function* (Class 4 - miscop 543)

Builds an open call block to *function* with a multi-value list return to receive returned values. Pushes an ADI for a multiple value list return, then completes construction of the open call block for *function*.

TI Internal Data

%CALLO-MULT-VALUE-LIST (Class 4 - miscop 544)

%CALLO-MULT-VALUE-LIST is to **%CALL-MULT-VALUE-LIST** as **CALLO** is to **CALL**.

%GC-SCAV-RESET *region* (Class 4 - miscop 545)

Makes the scavenger forget about a particular region. This also removes the region from the cons cache. Returns **T** if the scavenger was looking at this region, or **NIL** otherwise.

%P-STORE-CONTENTS-OFFSET *value, pointer, offset* (Class 4 - miscop 546)

Store *value* in contents of word *offset* beyond that addressed by *pointer*. This is not the same as could be done with **%P-STORE-CONTENTS** and **%MAKE-POINTER-OFFSET** because this instruction checks for a forwarding pointer in the word addressed by *pointer*. The idea is that *pointer* points at the beginning of a structure and *offset* is an offset within it.

%GC-FREE-REGION *region* (Class 4 - miscop 547)

The makes *region* free. *Region* is a region number. Use this on an oldspace region after scavenging is complete and the region contains nothing but garbage.

%GC-FLIP *region* (Class 4 - miscop 550)

Flips *region* converting new space to old space. Then makes sure nothing in the machine points to old space. If *region* is **T** all new space and copy space.

ARRAY-LENGTH *array* (Class 4 - miscop 551)

Returns the number of elements in *array*. Does not take account of the fill pointer.

ARRAY-ACTIVE-LENGTH *array* (Class 4 - miscop 552)

Returns the number of elements in *array*, or the fill pointer if there is one.

%COMPUTE-PAGE-HASH *addr* (Class 4 - miscop 553)

Computes the page hash table index that corresponds to *addr* and returns it as a fixnum.

THROW-SPREAD *tag, value-list* (Class 4 - miscop 554)

This does not really throw. It returns values in preparation for a throw. Values are extracted from *value-list* and all but the last one is returned. That last one is left on the stack. On return, the stack contains the tag and a single value, which you can pass to ***THROW** to complete the throw.

%UNIBUS-READ *Unibus-addr* (Class 4 - miscop 555)

Signals **UNIMPLEMENTED-HARDWARE** because Unibus is not implemented on Explorer.

%UNIBUS-WRITE *Unibus-addr, word* (Class 4 - miscop 556)

Signals **UNIMPLEMENTED-HARDWARE** because Unibus is not implemented on Explorer.

%GC-SCAVENGE *work-units* (Class 4 - miscop 557)

Scavenge for *work-units* of work or until a page fault. Returns **NIL** if completed *work-units* of work or ran out of work to do. Returns non-**NIL** if took a page fault before done. A "work unit" is the scavenging of one Q.

%CHAOS-WAKEUP (Class 4 - miscop 560)

This is an illegal instruction.

%AREA-NUMBER *X* (Class 4 - miscop 561)

Returns the area number of the area the pointer *X* points into.

***MAX** *num1, num2* (Class 4 - miscop 562)

Return the greater of *num1* and *num2*. Both *num1* and *num2* must be numbers or an **ARGTYP** trap is signalled.

***MIN** *num1, num2* (Class 4 - miscop 563)

Return the lesser of *num1* and *num2*. Both *num1* and *num2* must be numbers or an **ARGTYP** trap is signalled.

CLOSURE *symbol-list, function* (Class 4 - miscop 565)

Returns a closure, closing *function* over the variables in *symbol-list*. The closure is a function which when called will perform *function* in an environment in which those variables have the same bindings they have now (when the closure is created). Only special variables may be closed over.

AR-2-REVERSE *array, index2, index1* (Class 4 - miscop 566)

Return an element of *array*, optionally reversing the indices. While arrays are stored with first index varying fastest, this is the same as **AREF**. When arrays are stored with last index varying fastest, this uses *index1* as the first index even though it is the last argument.

LISTP *X* (Class 4 - miscop 567)

If the ojbect *X* is a list, the symbol **T** is returned. Otherwise, the symbol **NIL** is returned.

NLISTP *X* (Class 4 - miscop 570)

If the object *X* is a lisp atom, the symbol **T** is returned. Otherwise, the symbol **NIL** is returned.

SYMBOLP *X* (Class 4 - miscop 571)

If the object *X* is a symbol, the symbol **T** is returned. Otherwise, the symbol **NIL** is returned.

NSYMBOLP *X* (Class 4 - miscop 572)

If the object *X* is a symbol, the symbol **NIL** is returned. Otherwise, the symbol **T** is returned.

ARRAYP *X* (Class 4 - miscop 573)

If the object *X* is an array (DTP-ARRAY-POINTER), the symbol **T** is returned. Otherwise, the symbol **NIL** is returned.

FBOUNDP *symbol* (Class 4 - miscop 574)

If the function cell of *symbol* does not contain an unbound marker, the symbol **T** is returned. Otherwise, the symbol **NIL** is returned. *symbol* must be a symbol or an **ARGTYP** trap is signalled.

TI Internal Data

STRINGP *X* (Class 4 - miscop 575)

If the object *X* is a one dimensional array of ART-STRING or ART-FAT-STRING, the symbol **T** is returned. Otherwise, the symbol **NIL** is returned.

BOUNDP *symbol* (Class 4 - miscop 576)

If the value cell of *symbol* contains an unbound marker, the symbol **NIL** is returned. Otherwise, the symbol T is returned. *symbol* must be a symbol or an **ARGTYP** trap is signalled.

INTERNAL- *num1, num2* (Class 4 - miscop 577)

Calculate GCD on *num1* and *num2*. Both *num1* and *num2* must be numbers or an **ARGTYP** trap is signalled.

FSYMEVAL *symbol* (Class 4 - miscop 600)

Returns *symbol*'s definition, the contents of its function cell. If the function cell is unbound, a **TRANS-TRAP** is signalled. *Symbol* must be a symbol, otherwise an **ARGTYP** trap is signalled.

AP-1 *array, index* (Class 4 - miscop 601)

Returns a locative to array slot *index* of *array*. *** what about number arrays - what about?? ***

AP-2 *array, sub1, sub2* (Class 4 - miscop 602)

Returns a locative to array slot specified by *sub1* and *sub2* of *array*. *** see AP-1 for questions ***

AP-3 *array, sub1, sub2, sub3* (Class 4 - miscop 603)

Returns a locative to the slot of *array* specified by *sub1*, *sub2*, and *sub3*. *** see AP-1 for questions ***

AP-LEADER *array, index* (Class 4 - miscop 604)

Returns a locative to leader slot *index* of *array*.

%P-LDB-OFFSET *ppss, pointer, offset* (Class 4 - miscop 605)

Returns the contexts of byte *ppss* in the word *offset* beyond *pointer*. *Ppss* is a field specifier as in **LDB**. This is not the same as what could be done with **%P-LDB** and **%MAKE-POINTER**, because this instruction checks for a forwarding pointer in the word addressed by *pointer*. The idea is that *pointer* points at the beginning of a structure and *offset* is an offset within the structure.

%P-DPB-OFFSET *value, ppss, pointer, offset* (Class 4 - miscop 606)

Stores *value* into the byte *ppss* in the word *offset* beyond *pointer*. *Ppss* is a field specifier as in **LDB**. This is not the same as what could be done with **%P-DPB** and **%MAKE-POINTER-OFFSET** because this checks for a forwarding pointer in the word addressed by *pointer*. The idea is that *pointer* points at the beginning of a structure and *offset* is a offset within the structure.

%P-MASK-FIELD-OFFSET *ppss, pointer, offset* (Class 4 - miscop 607)

MASK-FIELD of *ppss* from the contents of the word *offset* beyond *pointer*. *Ppss* is a field specifier as in **LDB**. This is not the same as

```
(%P-MASK-FIELD ppss
(%MAKE-POINTER-OFFSET . . pointer offset))
```

because it checks for a forwarding pointer in the word addressed by *pointer*. The idea is that *pointer* points to the beginning of a structure and *offset* is an offset within it.

%P-DEPOSIT-FIELD-OFFSET *value, ppss, pointer, offset* (Class 4 - miscop 610)

Copy byte *ppss* from *value* into the word *offset* beyond *pointer*. *Ppss* is a field specifier as in **LDB**. This is not the same as what could be simulated using **%P-DPB** because this instruction checks for a forwarding pointer in the word addressed by *pointer*. The idea is that *pointer* points to the beginning of the structure and *offset* is an offset within it.

%MULTIPLY-FRACTIONS *num1, num2* (Class 4 - miscop 611)

Multiply *num1* and *num2* returns the high part of the resulting product. Both *num1* and *num2* must be fixnums.

%DIVIDE-DOUBLE *high-dividend, low-dividend, divisor* (Class 4 - miscop 612)

Divide the double precision number made from *high-dividend* and *low-dividend* by *divisor*. Return the integer quotient. Do the thing you expect.

%REMAINDER-DOUBLE *high-dividend, low-dividend, divisor* (Class 4 - miscop 613)

Do the thing you expect.

HAULONG *integer* (Class 4 - miscop 614)

Returns the "size" of *integer* in bits. (eg. the size of #o777 is nine bits.) *Integer* may be either a fixnum or a bignum.

%ALLOCATE-AND-INITIALIZE (Class 4 - miscop 615)
return-dtp, header-dtp, header, word2, area, nqs

Do the thing you expect.

%ALLOCATE-AND-INITIALIZE-ARRAY (Class 4 - miscop 616)
header, index-length, leader-length, area, nqs

Do the thing you expect.

%MAKE-POINTER-OFFSET *new-dtp, pointer, offset* (Class 4 - miscop 617)

Make a Lisp object (possibly an invalid one) which has a datatype of *new-dtp* and a pointer of *pointer* plus *offset*. The data types of *pointer* and *offset* are not checked and can even be fixnum. This instruction is dangerous unless used with extreme caution.

^ *num, expt* (Class 4 - miscop 620)

Exponentiate *num* to the *expt* power. Generic operation that works with all numeric types.

%CHANGE-PAGE-STATUS *virt-addr, swap-status, access-and-meta* (Class 4 - miscop 621)

Sets the swap status, access and meta-bits for the page corresponding to *virt-addr* if it is paged in. This does no error checking and can easily be very dangerous. Updates the page hash table entry and the memory map. If either *swap-status* or *access-and-meta* are **NIL** that parameter is not set. Returns **T** if the page was found in the page has table or **NIL** if it was not (meaning it was not swapped in).

%CREATE-PHYSICAL-PAGE *phys-addr* (Class 4 - miscop 622)

Add a new physical page to the pool of page frames. *Phys-addr* is the *** logical *** physical address of the page being added. Returns **T** if it succeeds. **ILLOP**'s if it fails. *** don't think this checks for adding a page already in the pools which would be very bad. ***

%DELETE-PHYSICAL-PAGE *phys-addr* (Class 4 - miscop 623)

Deletes the physical page from the page frame pool. *Phys-addr* is the *** logical *** physical address of the page. Returns **T** if it deletes the page and **NIL** if the page was already deleted.

%24-BIT-PLUS *num1, num2* (Class 4 - miscop 624)

Do the thing you expect. ;; should be 25-bit? ¿¿

%24-BIT-DIFFERENCE *num1, num2* (Class 4 - miscop 625)

Do the thing you expect. ;; should be 25-bit? ¿¿

%24-BIT-TIMES *num1, num2* (Class 4 - miscop 626)

Do the thing you expect. ;; should be 25-bit? ¿¿

ABS *num* (Class 4 - miscop 627)

Returns the absolute value of *num* which can be any type of number.

%POINTER-DIFFERENCE *ptr1, ptr2* (Class 4 - miscop 630)

Return the number of words difference between *ptr1* and *ptr2*. They had better be locatives into the same object for this operation to be meaningful; otherwise, their relative position will be changed by garbage collection.

%P-CONTENTS-AS-LOCATIVE *pointer* (Class 4 - miscop 631)

Returns a locative whose pointer field is copied from the word that *pointer* points to. If you have determined that the contents of that word is a pointer type, this is a good way to find the object it points to without getting things confused by forwarding or by DTP-NULL or by header data types.

%P-CONTENTS-AS-LOCATIVE-OFFSET *pointer, offset* (Class 4 - miscop 632)

Like **%P-CONTENTS-AS-LOCATIVE** but fetches the word *offset* locations beyond where *pointer* points. This is not the same as

```
(%P-CONTENTS-AS-LOCATIVE
(%MAKE-POINTER-OFFSET ... pointer offset))
```

because it checks for a forwarding pointer in the word addressed by *pointer*. The idea is that *pointer* points at the beginning of a structure and *offset* is an offset within it.

M-EQ *X, Y* (Class 4 - miscop 633)

Result is the symbol **T** if *X* and *Y* are the same Lisp object. Otherwise returns the symbol **NIL**.

%STORE-CONDITIONAL *pointer, old, new* (Class 4 - miscop 634)

Store *new* into *pointer* if the old contents of *pointer* was *old*. This is a basic interlocking primitive, which can be used to simulate any sort of atomic test-any-modify operation.

%STACK-FRAME-POINTER (Class 4 - miscop 635)

Returns a locative pointing at the first slot in the current stack frame. This is the slot that contains a pointer to the function that is executing. While this will execute in interpreted code, it is not likely to give anything useful therein.

***UNWIND-STACK** *tag, value, frame-count, action* (Class 4 - miscop 636)

Do the thing you expect.

%XBUS-READ *io-addr* (Class 4 - miscop 637)

Signals **UNIMPLEMENTED-HARDWARE** since Explorer does not have an XBUS.

%XBUS-WRITE *IO-ADDR, WORD* (Class 4 - miscop 640)

Signals **UNIMPLEMENTED-HARDWARE** since Explorer does not have an XBUS.

ELT *sequence, index* (Class 4 - miscop 641)

Common Lisp sequence access. *Sequence* may be a one-dimensional array or a list. *Index* must be a positive fixnum or **ARGTYP** is signaled. Returns the *index-th* element of *sequence*.

MOVE-PDL-TOP (Class 4 - miscop 642)

Copies the top object on the PDL to the destination without popping it.

SHRINK-PDL-SAVE-TOP *value-to-move, n-slots* (Class 4 - miscop 643)

Pops *n-slots* from the PDL and moves *value-to-move* to the destination. Both arguments are popped before counting the *n-slots* Q's to remove.

SPECIAL-PDL-INDEX (Class 4 - miscop 644)

The result is a locative to the last slot on the special PDL that was bound.

UNBIND-TO-INDEX *special-pdl-index* (Class 4 - miscop 645)

Undo bindingings on the special PDL until the special PDL pointer is less than or equal to *special-pdl-index*. Does not affect the indicators.

UNBIND-TO-INDEX-MOVE *special-pdl-index, value-to-move* (Class 4 - miscop 646)

Like **UNBIND-TO-INDEX**, undo bindingings on the special PDL until the special PDL pointer is less than or equal to *special-pdl-index*. Return *value-to-move*.

FIX *number* (Class 4 - miscop 647)

Convert *number* to the largest integer, which is less than or equal to *number*.

TI Internal Data

Code	Memory
1	Microinstruction
2	Dispatch
4	A and M
5	Tag Classifier

Table 12-11 Internal Memory Selector Codes

FLOAT *number* (Class 4 - miscop 650)

Convert *number* to a full-size floating point number.

SMALL-FLOAT *number* (Class 4 - miscop 651)

Convert *number* to a small flonum.

%FLOAT-DOUBLE *number, number2* (Class 4 - miscop 652)

Do the thing you expect.

BIGNUM-TO-ARRAY *bignum, base* (Class 4 - miscop 653)

Converts a bignum into an array. *Bignum* is a bignum, *base* is a fixnum. *Bignum* is expressed in *base* and stuffed into a appropriate art-q array. The sign of *bignum* is ignored. *** needs work ***

ARRAY-TO-BIGNUM *array, base, sign* (Class 4 - miscop 654)

Converts an array into a bignum. *Array* is an ART-Q array, *base* is a fixnum, and *sign* is the sign bit (0 or 1). *Array* is interpreted as a bignum expressed in *base* and with *sign*. Inverse of **BIGNUM-TO-ARRAY**. *** needs work ***

%UNWIND-PROTECT-CONTINUE *value, tag, count, action* (Class 4 - miscop 655)

This is similar to ***UNWIND-STACK** but takes its arguments in a different order. If *tag* is **NIL** this is a normal exit from an unwind-protect: simply move *value* to the destination. If *tag* is 1, means return to a **POP-OPEN-CALL** instruction tat popped an unwind protect's frame.

%WRITE-INTERNAL-PROCESSOR-MEMORIES *code, adr, d-hi, d-low* (Class 4 - miscop 656)

Code selects which memory gets written as shown in *Table 12-11*. If *code* is not one of the listed values, **BAD-INTERNAL-MEMORY-SELECTOR-ARG** is signaled. *Adr* is the address in that memory; *adr* is bounds checked against the size of the hardware memory and **INTERNAL-MEMORY-LOCATION-OOB** is signaled if out of bounds.

For microinstruction (I) memory, *d-hi* supplies the portion of the I-Mem data that can be modified with an **IMOD-HIGH** destination (see processor documentation). This will not need to be a bignum for Explorer. *D-low* supplies the portion of the I-Mem data that can be modified with an **IMOD-LOW** destination. This will sometimes need to be a bignum to supply 32 bits.

For A and M memories, the low order bits of *d-hi* supply the bits above Q-POINTER and *d-low* supplies the bits of Q-POINTER for the 32-bit data to be written. Neither should be a bignum. If *adr* is below the size of M memory, both A and M memories are written with the 32-bit data. otherwise only A memory is written.

For tag-classifier (T) memory, *d-hi* and *d-low* are combined as for A and M memories. The 32-bit data is loaded into the class selected by *adr*. Bit 0 corresponds to Q-Data-Type = 0 and bit 31 to Q-Data-Type = 31.

For dispatch (D) memory, *d-hi* is ignored. The low order bits of *d-low* supply the 17-bit data to be written.

%PAGE-STATUS *ptr* (Class 4 - miscop 657)

Returns the page hash table PHT1 word of the page that *ptr* addresses or **NIL** if the page is not swapped in. The modified bit is always up to date, even though that in PHT1 may not be.

%REGION-NUMBER *ptr* (Class 4 - miscop 660)

Return the number of the region *ptr* points into. Only the POINTER field of *ptr* is significant.

%FIND-STRUCTURE-HEADER *ptr* (Class 4 - miscop 661)

Given a locative, return the object containing the cell addressed by the locative. Finds the overall structure containing the cell addressed by the locative, *ptr*. Does not follow structure forwarding.

%STRUCTURE-BOXED-SIZE *ptr* (Class 4 - miscop 662)

Returns the number of normal Lisp pointers in an object. This many words at the beginning of the object contain normal Lisp data objects. The remaining words contain just numbers (just bits) that do not have the normal Lisp data typing. Does not follow structure forwarding.

%STRUCTURE-TOTAL-SIZE *ptr* (Class 4 - miscop 663)

Returns the number of words in the object, *ptr*.

%MAKE-REGION *bits, size* (Class 4 - miscop 664)

Create a region. Its size will be at least *size* and its region bits will be set to *bits*. You probably don't want to use this.

BITBLT (Class 4 - miscop 665)
ALU, width, height, from-array, from-x, from-y, to-array, to-x, to-y

Move with logical operation for bit and byte arrays. *From-array* and *to-array* must both be two-dimensional numeric arrays. *From-x* and *from-y* specify the coordinates of the upper, left-hand corner of a rectangle region on the source array. The rectangle has height and width as specified by *height* and *width*, but with wrap around if an array dimension is exceeded.

Elements are copied from *from-array* to *to-array* with logical operaton. The result stored into the *to-array* is (**BOOLE** *ALU from-elt to-elt*). *ALU* is as specified in **BOOLE**. Normally *ALU* is specified as the value of one of these symbols: **TV:ALU-SETA** for just copying, **TV:ALU-IOR** (inclusive-OR) for merging images. **TV:ALU-XOR** (exclusive-OR) for complementary drawing, or **TV:ALU-ANDCA** (AND with complement of source) for masking.

Normally, copying is performed to to bottom then left to right, but making *width* or *height* negative will alter the order of copying. Coordinates should still be for the top left corner.

Requires that the first dimension of the array be a multiple of 32 bits. Also index-offset arrays do not work with wrap-around.

TI Internal Data

%DISK-OP *RQB* (Class 4 - miscop 666)

Invalid instruction.

%PHYSICAL-ADDRESS *ptr* (Class 4 - miscop 667)

Return the address in core of the memory pointed to by the virtual address *ptr*. The returned value is a FIXNUM which may be negative. Only the pointer field of PTR is significant.

POP-OPEN-CALL *restart-PC* (Class 4 - miscop 670)

Remove one open call block from the stack without changing the PDL pointer. If the block being popped is an unwind-protect, *restart-PC* is the PC of the restart for the unwind-protect, otherwise it is zero. This should always have destination D-IGNORE.

%BEEP *half-wavelength, duration* (Class 4 - miscop 671)

Sound a beep on the console's speaker. The *half-wavelength* is the number of microseconds between zero crossings and *duration* is the number of microseconds for which the sound should continue. Returns after *duration* has passed.

This is being migrated to macrocode. No need for it to be an instruction with modern beep hardware.

%FIND-STRUCTURE-LEADER *ptr* (Class 4 - miscop 672)

Given a locative, return the object containing it and its leader. This is like **%FIND-STRUCTURE-HEADER** except that italways returns the base of the structure: thus for an array with a leader it gives a locative to the base instead of giving the array. Does not follow structure forwarding.

BPT (Class 4 - miscop 673)

Breakpoint. Signals **BREAKPOINT**.

%FINDCORE (Class 4 - miscop 674)

Returns the page frame number of an available page. Makes one available if none are available.

%PAGE-IN *PFN, VPN* (Class 4 - miscop 675)

This does not work and will trap with *** what error ***.

ASH *N, nbits* (Class 4 - miscop 676)

Shift *N* arithmetically by *nbits*. *N* may be any integer (DTP-FIXNUM or DTP-BIGNUM).

%MAKE-EXPLICIT-STACK-LIST *length* (Class 4 - miscop 677)

Immediately before the argument on the stack are *length* values that are to be made into a list. Changes the CDR-Code of the last one to CDR-NIL and returns a list pointer to the first of them.

%DRAW-CHAR (Class 4 - miscop 700)
font-array, char-code, x-bitpos, y-bitpos, alu-function, sheet

Draw character *char-code* of font *font-array* on *sheet* using *alu-function*. *Alu-function* is typically **TV:ALU-IOR**, **TV:ALU-ANDCA**, or **TV:ALU-XOR**. *X-bitpos* and *y-bitpos* are the position in *sheet* for the upper left corner of the character to be drawn.

%DRAW-RECTANGLE (Class 4 - miscop 701)
width, height, x-bitpos, y-bitpos, alu-function, sheet

Draw a solid rectangle on *sheet* using *alu-function*. *Alu-function* is typically **TV:ALU-IOR**, **TV:ALU-ANDCA**, or **TV:ALU-XOR**. *Height* and *width* are the size of the rectangle, and *x-bitpos* and *y-bitpos* are the location of the upper left corner.

%DRAW-LINE *X0, Y0, X, Y, ALU, draw-end-point, sheet* (Class 4 - miscop 702)

Draw a straight line from (*X0, Y0*) to (*X, Y*) on *sheet* using *ALU* as the ALU function. *ALU* is typically **TV:ALU-IOR**, **TV:ALU-ANDCA**, or **TV:ALU-XOR**.

%DRAW-TRIANGLE *X1, Y1, X2, Y2, X3, Y3, ALU, sheet* (Class 4 - miscop 703)

Draw a triangle with corners at (*X1, Y1*), (*X2, Y2*), and (*X3, Y3*) on *sheet*. *ALU* is the drawing function to use and is typically **TV:ALU-IOR**, **TV:ALU-ANDCA**, or **TV:ALU-XOR**.

%COLOR-TRANSFORM (Class 4 - miscop 704)
N17, N16, N15, N14, N13, N12, N11, N10, N7, N6, N5, N4, N3, N2, N1, N0, width, height, array, start-x, st

Modify all pixels in a rectangle with *array*, an **ART-4B** array. If the pixel contains 0, it is replaced by *N0*; if it contains 1, it is replaced by *N1*, etc. *Width* and *height* are the size of rectangle; *start-x* and *start-y* are the top left corner of the rectangle.

%RECORD-EVENT (Class 4 - miscop 705)
data-4, data-3, data-2, data-1, stack-level, event, must-be-4

Records a meter event on the meter band. Records *event* as the event number. The last argument indicates the number of data words supplied. Only 4 data words are supported and *MUST-BE-4* is only supported with a value of 4. Results are unpredicatable if this argument is not supplied as the FIXNUM 4.

The four data words are included in the event record *in reverse order*. That is, *data-1* is the first data word in the meter event record.

The function *stack-level* frames up the stack is recorded as the metered function in the meter event record.

%AOS-TRIANGLE *X1, Y1, X2, Y2, X3, Y3, increment, sheet* (Class 4 - miscop 706)

Add *increment* to each pixel in a triangle on *sheet*. The triangle's corners are at (*X1, Y1*), (*X2, Y2*), and (*X3, Y3*). Pixels are regarded as being between coordinate points, and the top left pixel is considered between X values 0 and 1, and between Y values 0 and 1.

%SET-MOUSE-SCREEN *sheet* (Class 4 - miscop 707)

Set the mouse screen to *sheet* which may be either an array or an instance of a sheet.

%OPEN-MOUSE-CURSOR (Class 4 - miscop 710)

Sets the mouse cursor state to open and undraws the mouse if it is currently being displayed.

SETELT *sequence, index, value* (Class 4 - miscop 711)

This is the Common Lisp compatible function to set the element of any sequence type object. The the element of *sequence* at *index* to be *value*. Sequence types include LIST, and ARRAY.

TI Internal Data

%BLT-TYPED *from-address, to-address, count, increment* (Class 4 - miscop 712)

Block copy typed data. Copy *count* words from *from-address* to *to-address*. After each transfer both addresses are incremented by *increment*. Each word read is transported and each word written is **GC-WRITE-TEST**'ed. Returns **NIL**.

UNUSED-713 (Class 4 - miscop 713)

Do nothing but error.

AR-1-FORCE *array, index* (Class 4 - miscop 714)

Return contents of element of *array* at *index*. *array* is treated as a one-dimensional in that it is indexed with a single subscript regardless of its rank.

AS-1-FORCE *value, array, index* (Class 4 - miscop 715)

Store *value* into element of *array* at *index*. *Array* is treated as one-dimensional in that it is indexed with a single subscript regardless of its rank.

AP-1-FORCE *array, index* (Class 4 - miscop 716)

Return a locative to element of *array* at *index*. *Array* is treated as one-dimensional in that it is indexed with a single subscript regardless of its rank.

AREF *array, &REST subscripts* (Class 4 - miscop 717)

Return the contents of the element of *array* specified by *subscripts*.

ASET *value, array, &REST subscripts* (Class 4 - miscop 720)

Store *value* into the element of *array* specified by *subscripts*.

ALOC *array, &REST subscripts* (Class 4 - miscop 721)

Return a locative to the element of *array* specified by *subscripts*.

EQUALP *X, Y* (Class 4 - miscop 722)

This is the Common Lisp EQUAL function. Almost the same as EQUAL. ;;; How is it different?? ;;;

%MAKE-EXPLICIT-STACK-LIST* *length* (Class 4 - miscop 723)

Same as **%MAKE-EXPLICIT-STACK-LIST*** except that the last of the words on the stack becomes the (*length* − 1)-th CDR of the returned list.

SETCAR *cons, newcar* (Class 4 - miscop 724)

Replaces the CAR of *cons* with *newcar*. Returns *newcar*. Just like **RPLACA** except that the returned value is different.

SETCDR *cons, newcdr* (Class 4 - miscop 725)

Replaces the CDR of *cons* with *newcdr*. Returns *newcdr*. Just like **RPLACD** except that the returned value is different.

GET-LOCATION-OR-NIL *plist, property* (Class 4 - miscop 726)

Returns a locative to the plist location containing the value of *property*. *Plist* can be a symbol, instance or disembodied property list. If the property is not found, returns **NIL**.

%STRING-WIDTH *table, offset, string, start, end, stop-width* (Class 4 - miscop 727)

Take each charracter in *string* from *start* to *end*, subtract the *offset* and use the difference as an index into *table*. *Table* is a character width table. Accumulate these values into a total width. Stop when the total width would go over the *stop-width*. If *stop-width* is **NIL**, same as having an infinite *stop-width*. Also stops if any element of *string* is not a number or character, if the *table* entry is not a number or if index into *table* is out of bounds for *table*.

Returns the cumulative sum and the index of the last location of *string* examined.

AR-1-CACHED-1 *array, subscript* (Class 4 - miscop 730)

Do the thing you expect.

AR-1-CACHED-2 *array, subscript* (Class 4 - miscop 731)

Do the thing you expect.

%MULTIBUS-READ-16 *multibus-byte-adr* (Class 4 - miscop 732)

Multibus not currently implemented on the Explorer System. Signals **UNIMPLEMENTED-HARDWARE**.

%MULTIBUS-WRITE-16 *multibus-byte-adr, word* (Class 4 - miscop 733)

Multibus not currently implemented on the Explorer System. Signals **UNIMPLEMENTED-HARDWARE**.

%MULTIBUS-READ-8 *multibus-byte-adr* (Class 4 - miscop 734)

Multibus not currently implemented on the Explorer System. Signals **UNIMPLEMENTED-HARDWARE**.

%MULTIBUS-WRITE-8 *multibus-byte-adr, word* (Class 4 - miscop 735)

Multibus not currently implemented on the Explorer System. Signals **UNIMPLEMENTED-HARDWARE**.

%MULTIBUS-READ-32 *multibus-byte-adr* (Class 4 - miscop 736)

Multibus not currently implemented on the Explorer System. Signals **UNIMPLEMENTED-HARDWARE**.

%MULTIBUS-WRITE-32 *multibus-byte-adr, word* (Class 4 - miscop 737)

Multibus not currently implemented on the Explorer System. Signals **UNIMPLEMENTED-HARDWARE**.

SET-AR-1 *array, subscript, value* (Class 4 - miscop 740)

Do the thing you expect.

SET-AR-2 *array, subscript1, subscript2, value* (Class 4 - miscop 741)

Do the thing you expect.

TI Internal Data

SET-AR-3 *array, subscript1, subscript2, subscript3, value* (Class 4 - miscop 742)

Do the thing you expect.

SET-AR-1-FORCE *array, subscript, value* (Class 4 - miscop 743)

Do the thing you expect.

SET-AREF *array, &REST subscripts-and-value* (Class 4 - miscop 744)

Do the thing you expect.

SET-ARRAY-LEADER *array, index, value* (Class 4 - miscop 745)

Do the thing you expect.

SET-%INSTANCE-REF *instance, index, value* (Class 4 - miscop 746)

Do the thing you expect.

VECTOR-PUSH *new-element, vector* (Class 4 - miscop 747)

For Common Lisp. *Vector* must have a leader or **ARRAY-HAS-NO-LEADER** is signalled. The fill pointer in leader element 0 must be a fixnum or **FILL-POINTER-NOT-FIXNUM** is signalled. The fill pointer is incremented and then *new-element* is stored in that location. If the array is full return **NIL** and leave the fill pointer unchanged, otherwise return the new fill pointer. *** this does not check that VECTOR is a vector ***

ARRAY-HAS-FILL-POINTER-P *array* (Class 4 - miscop 750).

Returns the symbol T if *array* is an array with a leader and leader element 0 contains a fixnum. Signals **ARGTYP** if *array* is not an array. Otherwise, returns the symbol **NIL**.

ARRAY-LEADER-LENGTH *array* (Class 4 - miscop 751)

Returns the length of the leader of *array*. Signals **ARGTYP** if *array* is not an array.

ARRAY-RANK *array* (Class 4 - miscop 752)

Returns the rank (number of dimensions) of *array*.

ARRAY-DIMENSION *array, dimension* (Class 4 - miscop 753)

Do the thing you expect.

ARRAY-IN-BOUNDS-P *array &REST subscripts* (Class 4 - miscop 754)

Do the thing you expect.

ARRAY-ROW-MAJOR-INDEX *array &REST subscripts* (Class 4 - miscop 755)

Do the thing you expect.

RETURN-N-KEEP-CONTROL *&REST values N* (Class 4 - miscop 756)

Do the thing you expect.

RETURN-SPREAD-KEEP-CONTROL *value-list* (Class 4 - miscop 757)

Do the thing you expect.

COMMON-LISP-LISTP *object* (Class 4 - miscop 760)

Returns T if *object* is the symbol NIL or has data type LIST. otherwise returns the symbol NIL.

%NuBus-READ *NuBus-slot, slot-byte-adr* (Class 4 - miscop 761)

Read the word (32 bits) from the location addressed by taking the low order 24 bits from *slot-byte-adr* and concatenating the low 8 bits from *NuBus-slot*. For NuBus slot space accesses, the 4 bits at bit 4 of *NuBus-slot* should be ones. Returns an integer, either a FIXNUM or a BIGNUM. The result is signed.

%NuBus-WRITE *NuBus-slot, slot-byte-adr, word* (Class 4 - miscop 762)

Write *word* (32 bits) at the location addressed by taking the low order 24 bits from *slot-byte-adr* and concatenating the low 8 bits from *NuBus-slot*. For NuBus slot space accesses, the 4 bits at bit 4 of *NuBus-slot* should be ones. *Word* should be an integer. either a FIXNUM or a BIGNUM. *Word* is signed.

%MICROSECOND-TIME (Class 4 - miscop 763)

Returns the 32-bit microsecond time as an integer. either a FIXNUM or a BIGNUM.

%FIXNUM-MICROSECOND-TIME (Class 4 - miscop 764)

Returns the 32-bit microsecond time truncated to 25-bits and typed as a FIXNUM.

%IO-SPACE-READ *IO-ADDR* (Class 4 - miscop 765)

Read 32 bits from HARDWARE-VIRTUAL-ADDRESS space. *IO-ADDR* is added to HARDWARE-VIRTUAL-ADDRESS to get the virtual address to use. Various hardware IO registers are mapped into HARDWARE-VIRTUAL-ADDRESS space. Returns an integer, either a BIGNUM or a FIXNUM. The returned value is signed.

%IO-SPACE-WRITE *IO-ADDR, WORD* (Class 4 - miscop 766)

Write 32 bits of *WORD* into HARDWARE-VIRTUAL-ADDRESS space. Address is developed as in **%IO-SPACE-READ**. *WORD* must be an integer. either a BIGNUM or a FIXNUM. *WORD* is interpreted as a signed quantity.

%NuBus-PHYSICAL-ADDRESS *APPARENT-PHYSICAL-PAGE* (Class 4 - miscop 767)

*** this was for Lambda *** *APPARENT-PHYSICAL-PAGE* gotten from by shifting value from **%PHYSICAL-ADDRESS**. Returned value is the 22-bit NuBus page number. *** don't really understand this. Do we need it? ***

VECTORP *object* (Class 4 - miscop 770)

For Common Lisp. Returns the symbol NIL if *object* is not a vector. A vector is defined by Common Lisp as a one dimensional array. Otherwise, returns symbol T.

SIMPLE-VECTOR-P *object* (Class 4 - miscop 771)

For Common Lisp. Returns the symbol T if *object* is a simple vector and the symbol NIL otherwise. A simple vector is a one dimensional numeric array. with no fill pointer, and which is not displaced or indirect.

TI Internal Data

SIMPLE-ARRAY-P *object* (Class 4 - miscop 772)

For Common Lisp. Returns the symbol T if *object* is a simple array, otherwise returns the symbol **NIL**. A simple array is an array that does not have a fill pointer and is not displaced or indirect. Do the thing you expect.

SIMPLE-STRING-P *object* (Class 4 - miscop 773)

For Common Lisp. Returns the symbol T if *object* is a simple string, otherwise returns the symbol **NIL**. A simple string is a single dimensional array of array type ART-STRING or ART-FAT-STRING with no fill pointer and not displaced or indirect.

BIT-VECTOR-P *object* (Class 4 - miscop 774)

For Common Lisp. Returns the symbol T if *object* is a bit vector, otherwise returns **NIL**. A bit vector is defined as a one dimensional array of array type ART-1b.

SIMPLE-BIT-VECTOR-P *object* (Class 4 - miscop 775)

For Common Lisp. Returns the symbol T if *object* is a simple bit vector, otherwise returns **NIL**. A simple bit vector is defined as a one dimensional array of array type ART-1b with no fill pointer that is not displaced or indirect.

NAMED-STRUCTURE-P *object* (Class 4 - miscop 776)

If *object* is a named-structure array, returns its name. Otherwise returns the symbol **NIL**. If the array has a leader, the name is stored in leader slot 1. If it has no leader, the name is stored in array element 0. The name returned is always a symbol. If the structure name is a closure, the symbol for the closed function is returned. If the name is not a symbol, the symbol **NIL** is returned.

NAMED-STRUCTURE-SYMBOL *object* (Class 4 - miscop 776)

Same as **NAMED-STRUCTURE-P**.

TYPEP-STRUCTURE-OR-FLAVOR *object, type* (Class 4 - miscop 777)

The result of this operation is the symbol T if *object* is a structure or instance whose name is or contains *TYPE*. If *object* is a structure whose name does not match *type* exactly, returns:

```
<<<< this isn't right >>>
(and (setq d (get xname 'defstruct-description))
(defstruct-description-named-p d)
(setq xname (car (defstruct-description-include d))))
(unless xname (return NIL))
```

If *object* is an instance, only returns the symbol T if the flavor of this instance depends on *TYPE*. Uses the %INSTANCE-DESCRIPTOR-DEPENDS-ON-ALL slot of the flavor to determine depended-on flavors.

12.7.2 Miscops Group 1

******** blah blah blah ********

FIXNUMP *OBJECT* (Class 4 - miscop 0)

Do the thing you expect

SMALL-FLOATP *OBJECT* (Class 4 - miscop 1)

Do the thing you expect.

CHARACTERP *OBJECT* (Class 4 - miscop 2)

Do the thing you expect.

CAR-SAFE *OBJECT* (Class 4 - miscop 3)

Do the thing you expect. ;;; what is this? ;;;

CDR-SAFE *OBJECT* (Class 4 - miscop 4)

Do the thing you expect.

CADR-SAFE *OBJECT* (Class 4 - miscop 5)

Do the thing you expect.

CDDR-SAFE *OBJECT* (Class 4 - miscop 6)

Do the thing you expect.

CDDDDR-SAFE *OBJECT* (Class 4 - miscop 7)

Do the thing you expect.

NTHCDR-SAFE *N, OBJECT* (Class 4 - miscop 10)

Do the thing you expect.

NTH-SAFE *N, OBJECT* (Class 4 - miscop 11)

Do the thing you expect.

CARCDR *LIST* (Class 4 - miscop 12)

Do the thing you expect.

ENDP *LIST* (Class 4 - miscop 13)

Returns the symbol **T** if *LIST* is the symbol **NIL**. Returns the symbol **NIL** if *LIST* is of data type list. Otherwise, signals **ARGTYP**. Similar to **NOT** but signals **ARGTYP** with inputs that **NOT** accepts.

CONSP-OR-POP *OBJECT* (Class 4 - miscop 14)

Do the thing you expect.

INDICATORS-VALUE *OBJECT* (Class 4 - miscop 15)

Do the thing you expect.

TI Internal Data

%POINTER-TIMES *pointer1, pointer2* (Class 4 - miscop 16)

Do the thing you expect.

COMMON-LISP-AREF *array, &REST indices* (Class 4 - miscop 17)

For Common Lisp. Do the thing you expect

COMMON-LISP-AR-1 *array, index* (Class 4 - miscop 20)

For Common Lisp. Do the thing you expect.

COMMON-LISP-AR-1-FORCE *array, index* (Class 4 - miscop 21)

For Common Lisp. Do the thing you expect.

INTERNAL-GET-3 *symbol, property, default* (Class 4 - miscop 22)

¡¡¡ What is this?? ¿¿¿ Do the thing you expect.

%IO *RQB, device-desc* (Class 4 - miscop 23)

Initiate IO request described by *RQB* (Request-Block) on the device described by *device-desc*. The interpretation of *RQB* is defined by the device. *RQB* is often an array. *device-desc* is an IO-DEVICE-DESCRIPTOR as defined in section on (device descriptor).

%ADD-INTERRUPT-ENTRY *device-desc, level* (Class 4 - miscop 24)

Installs an interrupt for the device described by *device-desc* at interrupt level *level*. Device must have a microcode interrupt handler for this device type. Does not initialize the device interrupts.

%DRAW-FILLED-TRIANGLE (Class 4 - miscop 25)
x1, y1, x2, y2, x3, y3, ledge, tedge, redge, bedge, ALU, draw3rd, draw2nd, draw1st, fill-color, dest

Do the thing you expect.

%DRAW-FILLED-RASTER-LINE (Class 4 - miscop 26)
x1, y1, y, l-edge, t-edge, r-edge, b-edge, ALU, draw-last-pt, fill-color, dest

Do the thing you expect.

%ADD-PAGE-DEVICE *unit-number, starting-block, size* (Class 4 - miscop 27)

Do the thing you expect.

%NuBus-READ-8B *hi-address, low-address* (Class 4 - miscop 30)

Do the thing you expect.

%NuBus-WRITE-8B *hi-address, low-address, data* (Class 4 - miscop 31)

Do the thing you expect.

%NuBus-READ-16B *hi-address, low-address* (Class 4 - miscop 32)

Do the thing you expect.

%NuBus-WRITE-16B *hi-address, lou-address, data* (Class 4 - miscop 33)

Do the thing you expect.

Miscop codes 34 and 35 reserved for future implementations of physical read and write.

%NuBus-READ-8B-CAREFUL *hi-address, low-address* (Class 4 - miscop 36)

Do the thing you expect.

%RATIO-CONS *numerator, denominator* (Class 4 - miscop 37)

Do the thing you expect.

RATIONALP *x* (Class 4 - miscop 40)

Do the thing you expect.

RATIOP *x* (Class 4 - miscop 41)

Do the thing you expect.

COMPLEXP *x* (Class 4 - miscop 42)

Do the thing you expect.

INT-CHAR *fixnum* (Class 4 - miscop 43)

Do the thing you expect.

CHAR-INT *character* (Class 4 - miscop 44)

Do the thing you expect.

%BLT-TO-PHYSICAL (Class 4 - miscop 45)
source-addr, dest-addr, number-of-words, increment

Do the thing you expect.

%BLT-FROM-PHYSICAL (Class 4 - miscop 46)
source-address, dest-addr, number-of-words, increment

Do the thing you expect.

From 47 to 776 are still free.

%CRASH *code, object, paws-up-p* (Class 4 - miscop 777)

Software crash of the machine. Halt the machine, but first write a crash record with a crash kind of software and a crash code of *code*. *Code* should be a fixnum and will be truncated to 16 bits. *Object* is remembered in the crash record to report. Most pointer objects will not report meaningfully. *Paws-up-p* if **NIL** signifies that the user has been notified of the crash and it is not necessary to display the crash indication (inverse video or whatever).

A normal shutdown should use this after performing all shutdown tasks (like cleanly closing the file system) with *code* of 0.

TI Internal Data

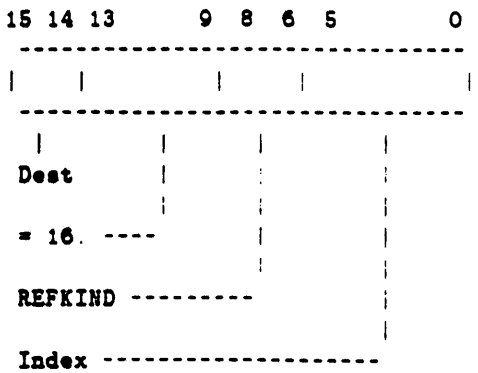


Fig. 12-10 AREFI Instruction Format

REFKIND	Instruction
0	**AREFI-ARRAY-NEW**
1	**AREFI-ARRAY-LEADER-NEW**
2	**AREFI-INSTANCE-NEW**
3	**AREFI-ARRAY-NEW-COMMON-LISP**
4	**AREFI-SET-ARRAY-NEW**
5	**AREFI-SET-ARRAY-LEADER-NEW**
6	**AREFI-SET-INSTANCE-NEW**
7	illegal

Table 12-12 Reference Kind Decoding

12.8 Array Reference Immediate (Class V) Instructions

Class V instructions support abbreviated sequences for accessing one dimensional array's (also array leaders and instances) with a small constant index. The format of an **AREFI** instruction is shown in *Fig. 12-10*. The kind of object to be referenced is indicated in the **REFKIND** field. The decoding of the **REFKIND** field is shown in *Table 12-12*.

Index is a 6-bit index into the structure.

The structure to be accessed is an argument on the stack. If an **AREFI-SET-** instruction, a second argument is the new value.

13. Flavors

31 Flavors

Baskin Robbins Ice Cream

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 provide 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.

13.1 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.

13.2 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 same for all instances of a flavor and is not really dynamic (although some dynamic changes can be accomodated). The elements of the instance descriptor are as indicated in *Table 13-1*. 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).

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

Table 13-1 Instance Descriptor Offsets

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 EVCP's 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. Used to be a select-method.

Next is the %INSTANCE-DESCRIPTOR-TYPENAME which 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 marked %INSTANCE-DESCRIPTOR-IGNORE are used only at higher levels.

%INSTANCE-DESCRIPTOR-ALL-INSTANCE-VARIABLES is a list of all instance variables. *** in same order as slots in 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**.

13.3 Self Mapping Table

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

It will take some effort to explain why this is required. 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 combined. 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 on self-mapping table is needed. If the flavor has **:ORDERED-INSTANCE-VARIABLES**, the methods know which position will contain each instance variable. They can them 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.

TI Internal Data

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

13.4 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.

13.5 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. The $n - 1$ 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 *** about ?90

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.

13.6 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 section on self-reference pointers for more details.

There is an addressing mode provided in macroinstructions that allows access to instance variables. See section on SELF addressing mode for more details.

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