SECTION 7

Internal Storage Formats

7.1 INTRODUCTION

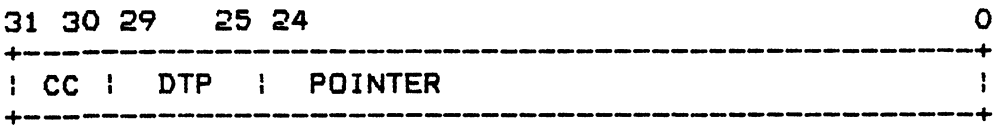
This section details the Explorer virtual machine internal storage formats. Symbolic byte specifiers and constants for most of the formats described can be found in the system parameters file SYS:UCODE;LROY-QCOM. The convention is that byte specifiers begin with a double percent (%%) while symbolic constants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

7.2 THE TAGGED ARCHITECTURE

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

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

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



POINTER <0:24> = Pointer field. Contains immediate data or pointer to actual storage.
DTP <25:29> = Data Type field.
CC <30:31> = Cdr Code field.

Figure 7-1 Q Format

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

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

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

7.3 BOXED VERSUS UNBOXED MEMORY

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

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

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

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

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

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

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

7.4 DATA TYPE SUMMARY

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

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

Table 7-1 Lisp Object Data Type Attributes

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

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

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

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

Table 7-2 Housekeeping Data Types

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

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

7.5 IMMEDIATE DATA TYPES

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

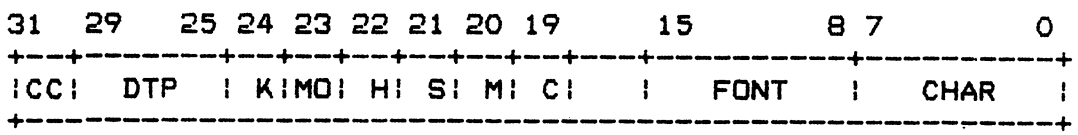
7.5.1 DTP-Fix.

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

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

7.5.2 DTP-Character.

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



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

Figure 7-2 Character Format

7.5.3 DTP-Short-Float.

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

reduced from 23 bits to 16 bits.

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

7.6 POINTER OBJECT TYPES

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

Table 7-3 Pointer Types

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

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

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

7.7 LIST POINTER OBJECTS

7.7.1 DTP-List.

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

Table 7-4 CDR Codes

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

7.7.2 CONS Cells.

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

7.7.3 Cdr-Coded Lists.

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

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

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

7.7.4 Destructive Operations: RPLACA, RPLACD.

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

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

7.7.5 DTP-Stack-List.

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

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

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

7.7.6 Closures.

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

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

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

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

7.8 STRUCTURE POINTER OBJECTS

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

7.8.1 Symbols.

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

Table 7-5 Qs of Symbol

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

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

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

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

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

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

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

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

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

7.8.2 Instances.

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

7.8.3 Arrays.

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

7.8.3.1 Stack Groups.

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

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

7.8.3.2 Regular PDL Arrays.

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

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

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

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

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

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

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

7.8.3.3 Special PDL Arrays.

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

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

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

7.8.4 Compiled Functions (FEFs).

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

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

7.8.5 DTP-Header.

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

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

Table 7-6 Header Types

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

7.8.6 DTP-Extended-Number.

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

7.8.6.1 BIGNUM.

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

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

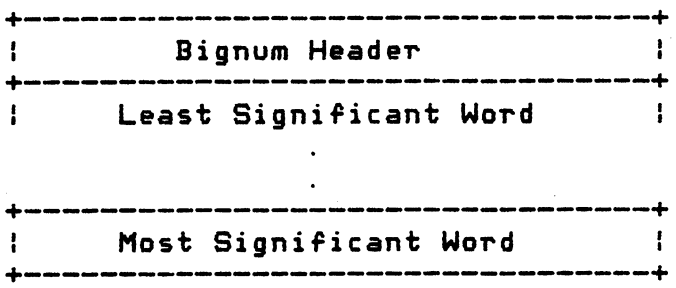


Figure 7-3 BIGNUM Structure

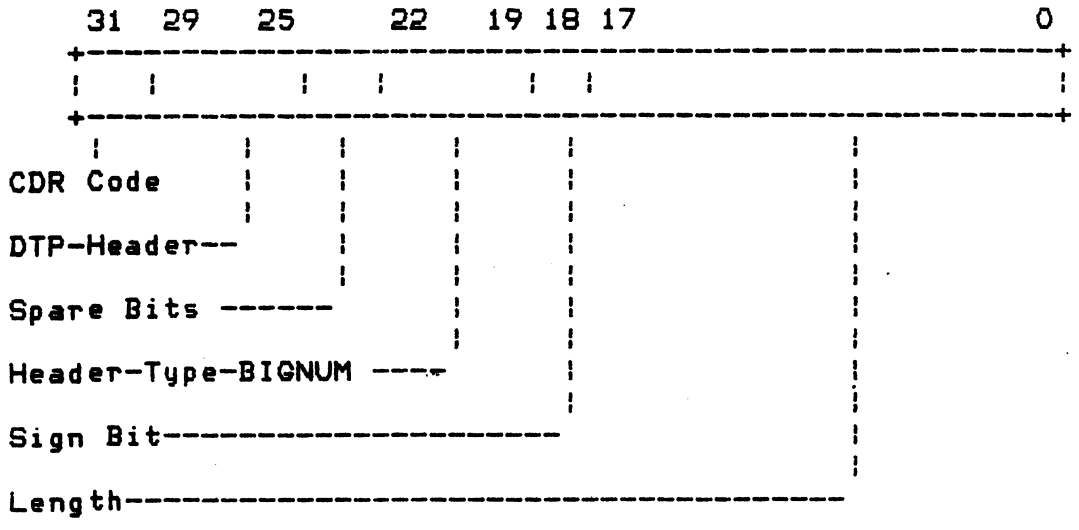


Figure 7-4 BIGNUM Header Format

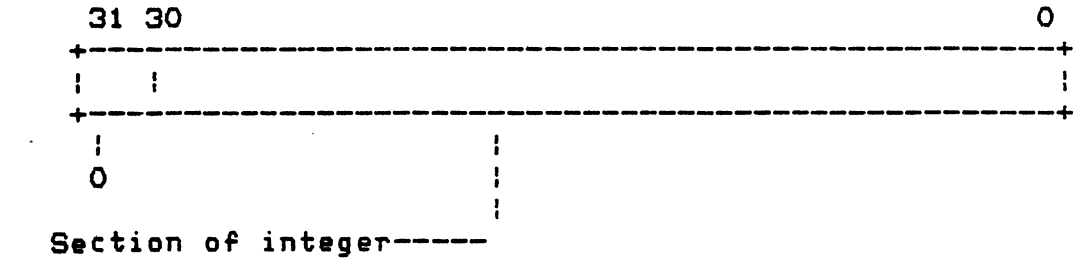


Figure 7-5 BIGNUM Data Format

7.8.6.2 RATIONAL.

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

7.8.6.3 COMPLEX.

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

7.8.7 Floating Point Numbers.

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

7.8.7.1 Short Float.

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

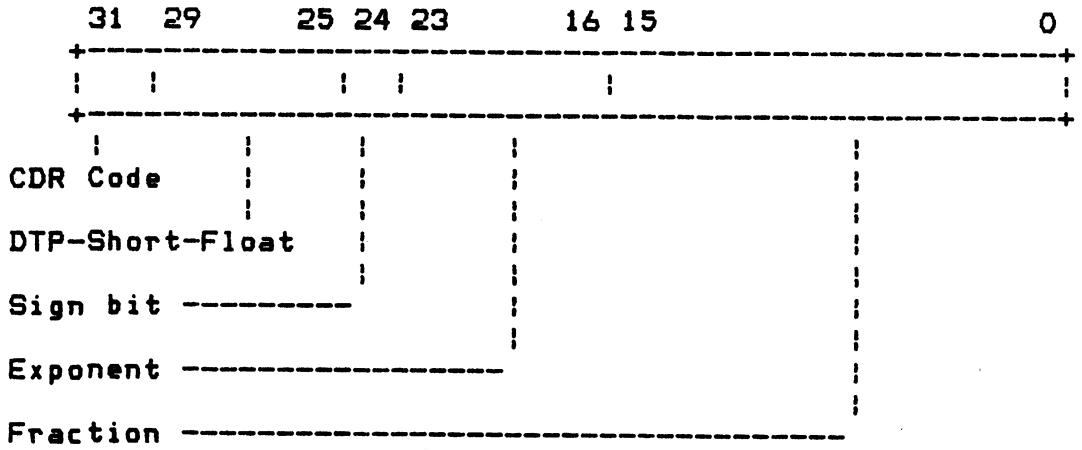


Figure 7-6 Short Float Format

7.8.7.2 Single Float.

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

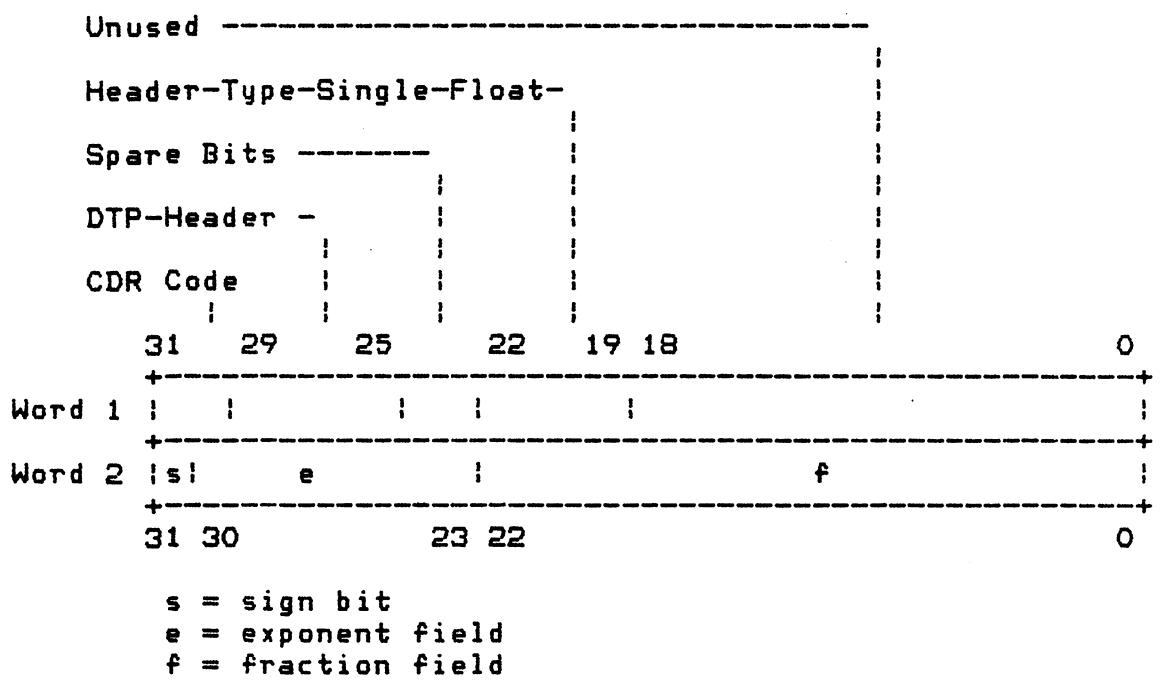


Figure 7-7 Single Precision Float Structure Format

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

7.8.7.3 Double Float.

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

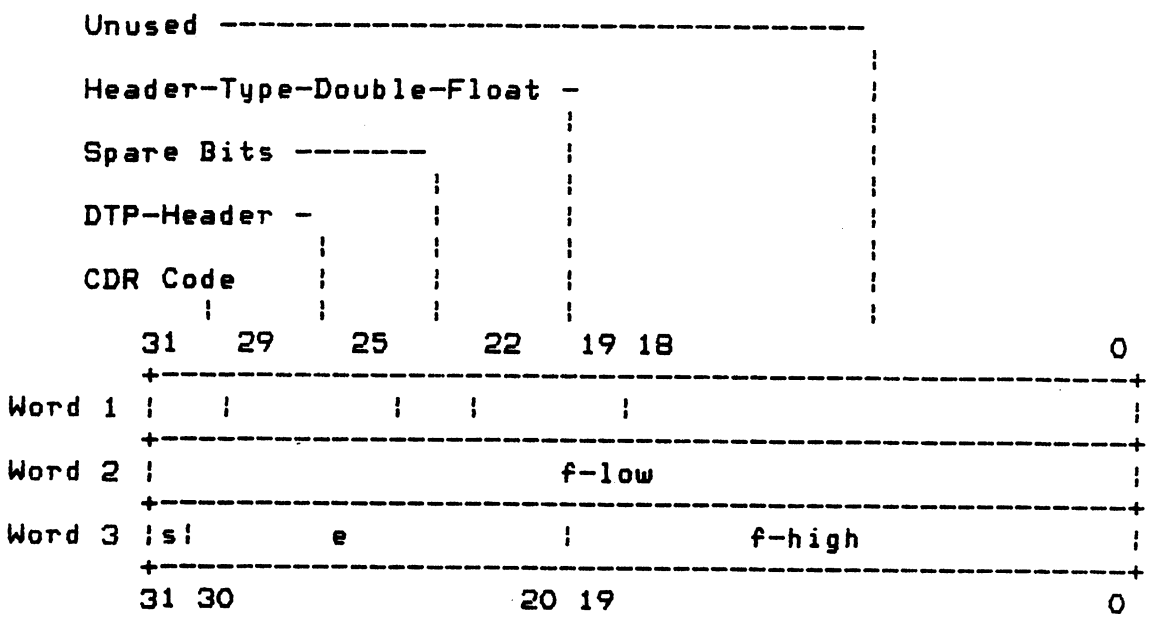


Figure 7-8 Double Precision Float Structure Format

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

7.8.8 The Cdr Code Field in Structures.

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

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

7.9 OTHER POINTER OBJECTS

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

7.9.1 DTP-Locative.

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

7.9.2 DTP-U-Entry.

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

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

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

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

7.10 HOUSEKEEPING TYPES

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

7.10.1 Forwarding Pointer Types.

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

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

7.10.1.1 DTP-One-Q-Forward.

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

7.10.1.2 DTP-External-Value-Cell-Pointer.

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

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

Lisp interpreter also uses EVCPs to implement dynamic binding.

7.10.1.3 GC Forwarding Pointers.

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

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

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

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

7.10.2 Structure Forwarding.

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

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

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

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

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

7.10.3 DTP-Self-Ref-Pointer.

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

Table 7-7 SELF-REF-POINTER Format

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

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

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

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

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

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

7.10.4 Trap Types.

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

7.10.4.1 DTP-Null.

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

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

7.10.4.2 DTP-Trap and DTP-Ones-Trap.

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

7.10.4.3 Unused Type DTP-Free.

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