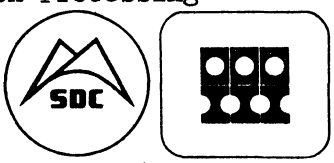
This document was produced by SDC in performance of contract AF 19(628)-5166 with the Electronic Systems Division, Air Force Systems Command, in performance of ARPA Order 773 for the Advanced Research Projects Agency Information Processing Techniques Office, and Subcontract 65-107.



TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 200 Sixth Street / Cambridge, Massachusetts 02142

TM- 2710/570/00

AUTHOR S. L. Kameny, SDC
L. Hawkinson, III

TECHNICAL M. Levin, III
S. L. Kameny, SDC

RELEASE A. L. Fenaughty, III
S. L. Kameny, SDC
for D. L. Drukey

DATE 21 December 1965

PAGE 1 OF 25 PAGES

LISP II INTERNAL STORAGE CONVENTIONS

ABSTRACT

This document presents a description of the internal structures and allocations of the core storage for LISP II and the internal representations of data for the Q-32 implementation of LISP II. In general, the storage area is composed of Fixed Program Space, Character Atoms, Triples, Pushdown Stack, Binary Program Space, Array Space, and List Space. The core map that is in use at any specific time is indicated by a pushdown pointer used in conjunction with a set of OWN variables in LISP II in section SYS.

FOREWORD

LISP II is a joint development of SDC and III. The idea for LISP II as a language combining the properties of an algebraic language like ALGOL and the list-processing language LISP was conceived by M. Levin of MIT. Development of the concepts of LISP II was carried forth in a series of conferences held at MIT and Stanford University. Contributions in concepts and detail were made by Prof. John McCarthy of Stanford University, Prof. Marvin Minsky of MIT, and the LISP II project team consisting of M. Levin, L. Hawkinson, R. Saunders and P. Abrahams of III, and S. Kameny, C. Weissman, E. Book, Donna Firth, J. Barnett and V. Schorre of SDC.

CONTENTS

FIGURES

TABLES

1. GENERAL ORGANIZATION OF CORE STORAGE

The general organization of core storage in LISP II is, as shown in Figure 1, composed of seven different areas called Fixed Program Space, Character Atoms, Triples, Pushdown Stack, Binary Program Space, Array Space, and List Space. The pushdown pointer, PDP, kept in Index Register 8, together with a set of OWN variables, defined in section SYS, serves to define the current core map at any point in time. Each of the fluid variables is of type OCTAL and contains a core address whose meaning is shown in Figure 1. The core address contained in each variable is just higher than the corresponding boundary of the core map. Thus, Fixed Program Space, which is built toward higher addresses, starts at location FPO and extends to the cell just lower than the location FPP. Similarly, the Pushdown Stack, which is built towards lower addresses, starts at the address just lower than location BPO, and extends appropriately to the location PDP. The boundaries FPO, CHO and TRO are fixed in the system and cannot be changed except by reassembling the entire system; all other boundaries are movable. Garbage collection reclaims Triple Space, Array Space, and List Space, and repacks Array Space and List Space. Binary Program Space is also reclaimable, compactable, and usable.

2. CORE AREAS

2.1 FIXED PROGRAM SPACE

Fixed Program Space, which starts at the fixed program origin (FPO) is used to hold fixed, non-relocatable constants and subroutines. Only those essential elements of the system which cannot be deleted should be put here, using LAP with an ORG pseudo-instruction (see LAP II memo for further details).

2.2 CHARACTER IDENTIFIER

Character identifiers, which start at the character origin (CHO) are all identifiers whose print names consist of a single character. The character whose print name has the ASCII code aa is located at aa + CHO. The structure of a character identifier is shown in Figure 2.

2.3 TRIPLE SPACE

Triple cell space, organized in groups of three cells, is used for the storage of identifiers, fluid cells, own cells (including function descriptors) and quote cells. Triple cell space starts at location TRO (immediately after the character identifiers). The triple cell pointer TRP points to the first higher address not occupied by triple cells. The triple cell maximum TRM points to a movable boundary between triple cell space and the pushdown list. Within triple cell space, structures are never moved, but if a triple cell is abandoned it is converted to an empty triple cell and linked onto the (possibly empty) free-triple chain, TRL. TRL is an OWN OCTAL variable in section SYS. A new triple cell is taken from the free-triple chain or if the free-triple chain is empty, the triple cell is placed at the end of triple cell space, and the boundary TRP is moved. If moving the boundary TRP would cause TRP to equal TRM, the garbage collector is called to reclaim triple cells. TRM is moved when necessary.

OWN OCTAL variables in section SYS

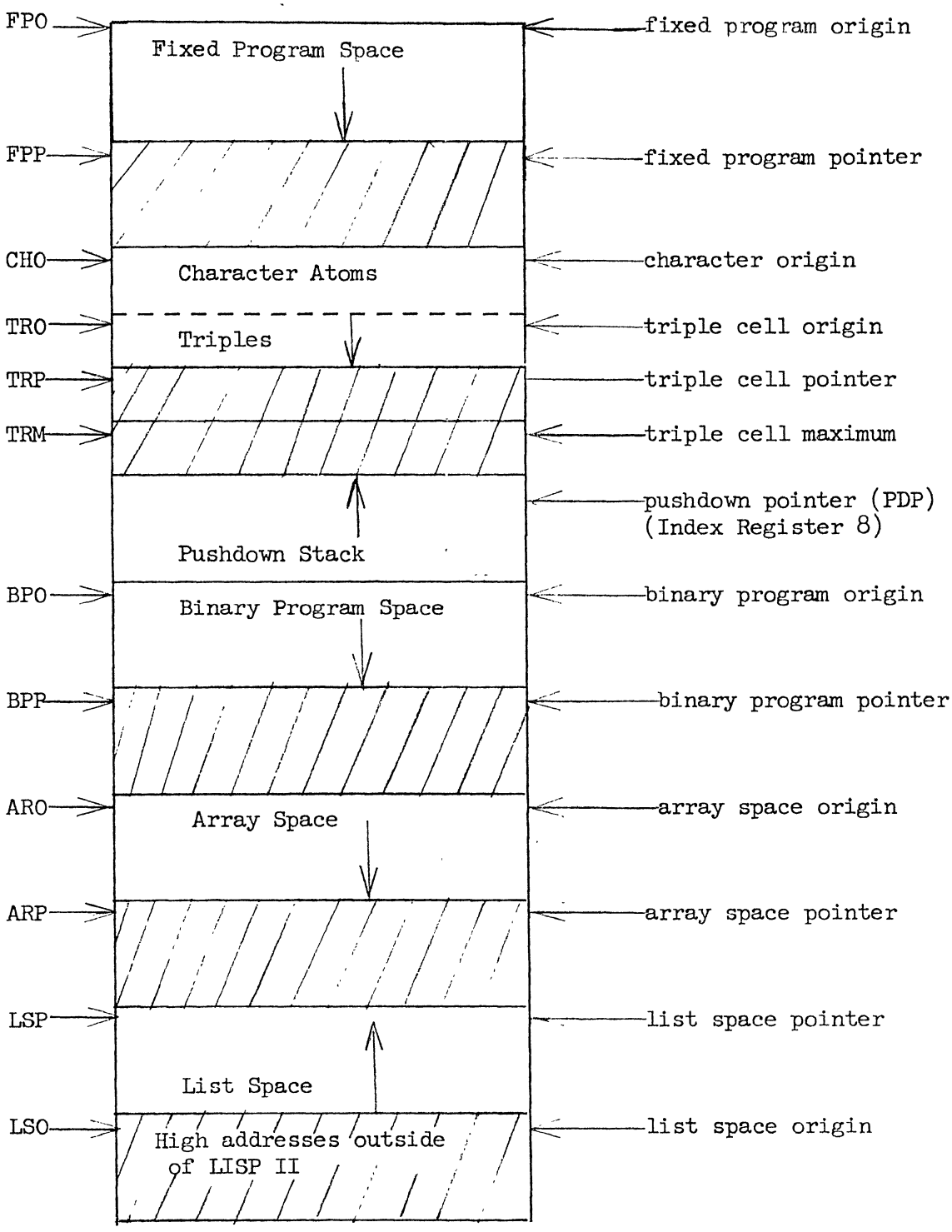


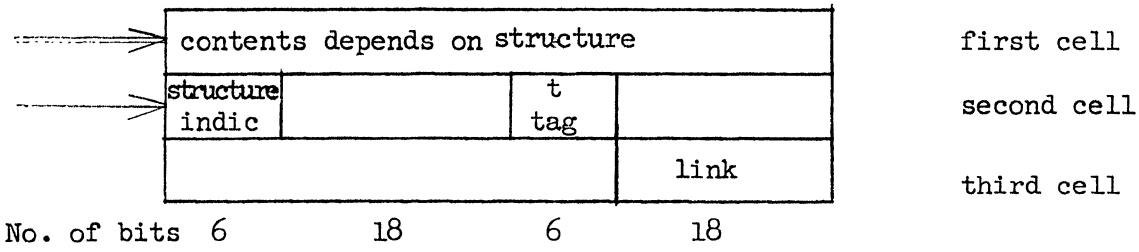
Figure 1. LISP II Core Allocation

∅7	v-f-chain	t tag	property list

t = 6∅ for alphabetic characters A-Z (identifier with standard spelling)

t = 7∅ for other characters (identifier with unusual spelling)

Figure 2. Character Identifier



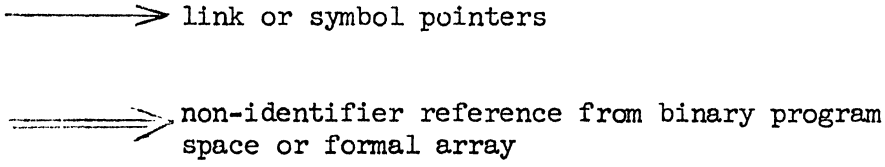


Figure 3. Triple Space Structure-General

A triple space structure consists of three consecutive cells in memory, as shown in Figure 3. Pointers in general go to the second cell of a triple. These pointers include the link pointer and symbol pointers. This is shown by the single arrow ⟶ in the figure.

References to non-identifier triples from binary program space, particularly direct or indirect load and store instructions and BUC indirect instructions use the address of the first cell, as shown by the double arrow ⟹ in the figure.

Triple cells other than quote cells are organized into a free-triple chain plus a series of "buckets." The free-triple chain is a chain of empty triples pointed to from variable TRL in section SYS. The chain is tied together through the link pointer of the third cell, with NIL in the last link. Each non-empty "bucket" is a chain of identifiers of which the first identifier is in (pointed to by) the OBLIST, an OWN SYMBOL array in section SYS. The bucket is tied together through the link pointer. Each identifier is tied through the v-f-chain in the left half of its second word to a variable-and-function chain composed of those fluid cells and own cells having it as a name. The v-f-chain is a circular list strung by means of the link pointer, with the last triple in the chain pointing back to the identifier. An identifier also contains a property-list pointer. The property-list is by convention a list containing any mixture of flags and property pairs. A flag is any atom, while a property pair is a dotted pair whose CAR is the property name (an atom) and whose CDR is the property value (any symbol).

The structure of buckets and v-f-chains is shown in Figure 4.

Quote cells, also contained in triple cell space, have no linked structure.

Genids, or generated identifiers, are generated by the function GENID of no arguments in section NIL. Genids have the same structure as identifiers except that they have a genid indicator bit in the tag. They are not strung on the OBLIST. Genids are produced by GENID with no pname, i.e., the first word is all zero. The first time that a specific genid is printed out, it is supplied a name by the function GENPNAME in section SYS, and this name persists for the life of the genid. Genids are normally printed in the form %Gname. where % is the escape character. The LISP II READ program prints genids to be read in in this form and converted internally to genids of no name.

2.4 PUSHDOWN STACK

The pushdown stack and its organization are given in the LAP II document.

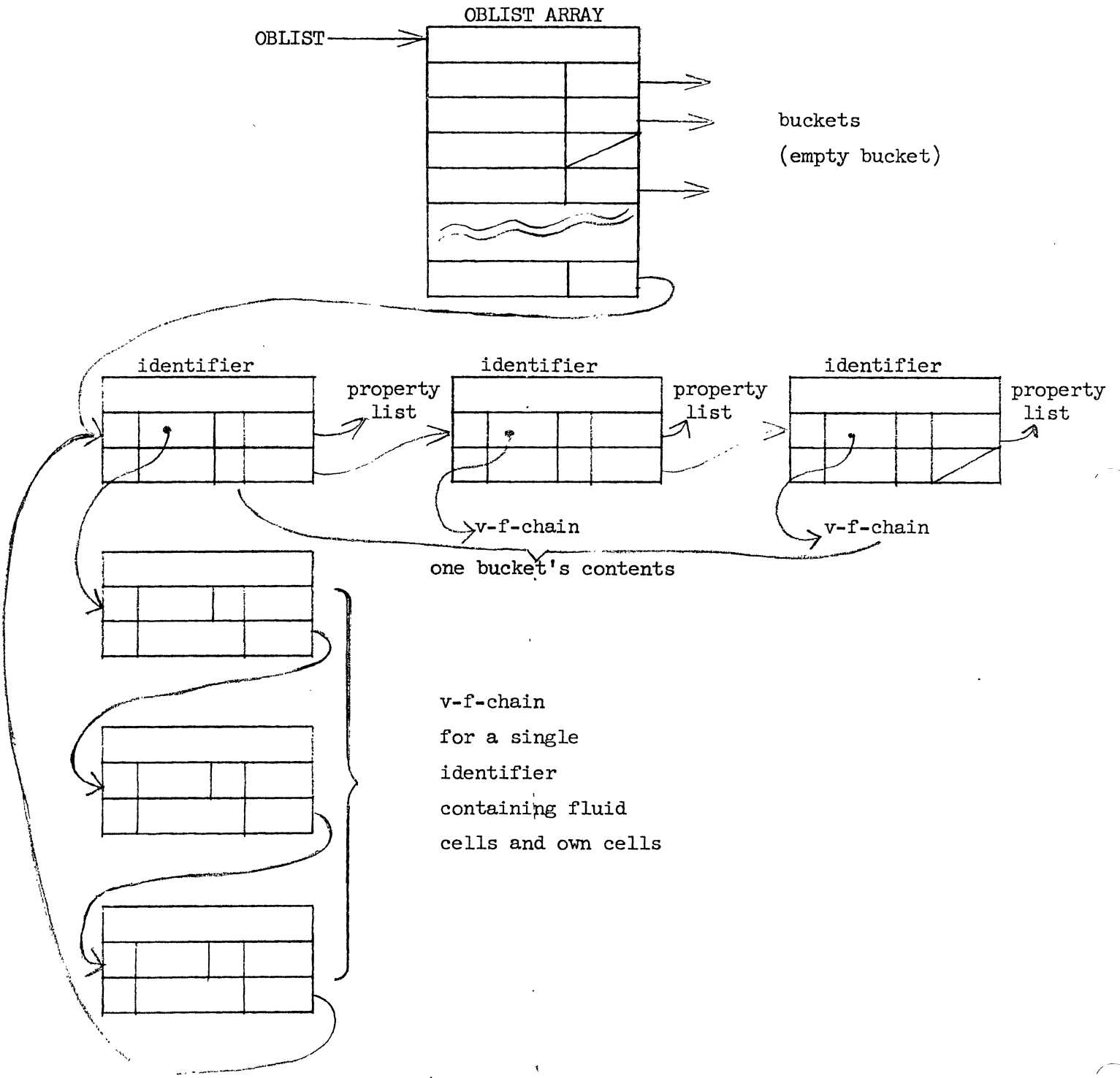
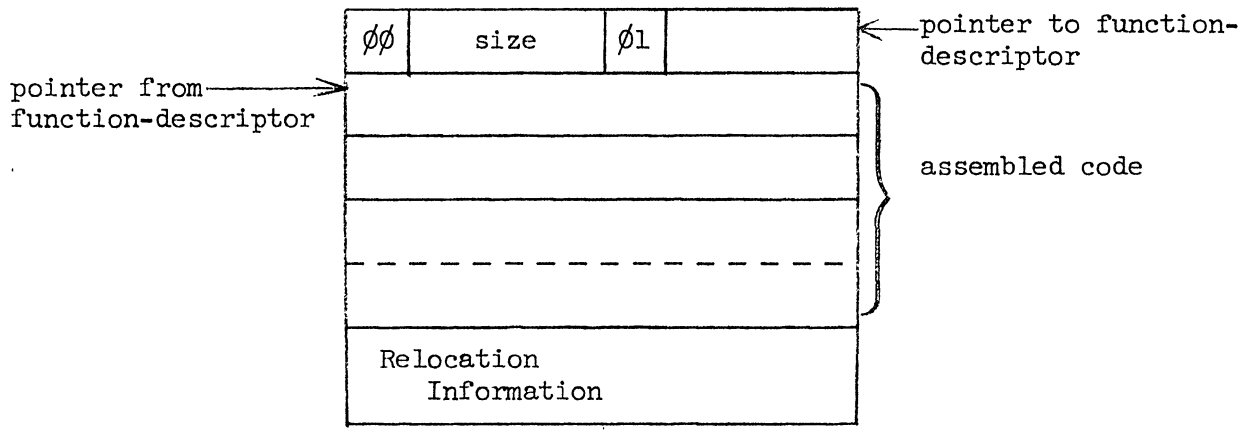


Figure 4. Buckets and v-f-chains

2.5 BINARY PROGRAM SPACE

Binary program space consists of a packed series of assembled functions, each of which consists of a header word, assembled code, and relocation information, as shown in Figure 5.

The size contained in the header word is the total length of the assembled-function, including the relocation information. The relocation information, two bits per word of assembled-function, is packed from left to right into a series of words starting with the last word and working toward the header word of the assembled-function. The two bits refer to the left half and the right half of words of assembled code, beginning with the header word and extending to the cell immediately preceding the relocation information. The coding employed is the following: $\emptyset$ means no relocation or count, 1 means that if the address lies within this assembled-function it is a relocatable address. If the address does not lie within this assembled-function, then it points into triple cell space. The 1 in this latter sense means that the count in the triple referred to is to be incremented when this assembled-function is loaded and decremented when the function is excised.



size = total number of cells

Figure 5. Assembled Function

2.6 ARRAY SPACE--REPRESENTATION OF DATA

Array space is used to hold arrays, numbers, formals, and strings, the last three of which may be regarded as special cases of arrays. The four structures are shown in Figure 6. One-dimensional arrays containing n elements have size = n + 1. (An empty array has size = 1.) The self-pointer, which is always contained in an array header word, is used by the garbage collector. Bit t_{24} of the tag portion of the header is used by the garbage collector for marking the array structure during garbage collection and $t_{24} = \emptyset$ otherwise. The meaning of the structure indicator is given in Table 1.

Arrays

Each element of an array having structure indicator 2$\emptyset$, 21, 22, 23, 24, or 25 is a datum of the appropriate type (SYMBOL, BOOLEAN, OCTAL, INTEGER, REAL, FORMAL, respectively). An array with structure 3$\emptyset$ is used to hold full-locative pointers.

Number

A number structure contains a single numeric datum. The numerical value contained in a number is a real number for structure indicator $\emptyset$4, an octal for structure indicator $\emptyset$2, and an integer for structure indicator $\emptyset$3. The numerical value is right-justified and occupies the entire second word of the number structure.

Formal

A formal structure, denoted by structure indicator $\emptyset$5, holds a single formal datum. As shown in Figure 6, a formal datum contains a code-pointer in its address field, a symbol pointer in the decrement, a zero prefix, and has the indirect bit set in its tag to permit indirect addressing through the formal datum. The symbol contained in the formal datum points in general either to a full-locative array or to NIL. (The full locative-array is used in functionals to hold funarg variables.)

String

A string structure, denoted by structure indicator $\emptyset$6, contains a single string. Strings contain six 8-bit ASCII character bytes per word, filled from the left end, as shown by c_1, c_2, c_3 ... n (Figure 6). Unused bytes are filled with the null-character $\emptyset\emptyset_{16}$. The tag portion of the array header shows the number of characters in the last word of the array. An empty string consists of the single header cell having a prefix of $\emptyset$6, size 1, tag $\emptyset$6, and a self-pointer.

Full-locative

A full-locative, which can occur only in a full-locative array, or as the first word of a fluid cell, is a full word containing a pointer to a cell containing a datum. The cell pointed to may be an array element or it may be the first word of an own cell, or a cell on the pushdown list. As shown in Figure 7, the address (CDR) portion of the full-locative contains the address of the cell pointed to, while if the cell is an array element, the decrement (CAR) portion of the full-locative points to the array head.

One dimensional array AA

structure indicator	Size	t Ø	self-pointer	size = n+1
datum or full-locative				(AA 1)
				(AA 2)
				(AA 3)
				(AA n)

structure indicator = 2Ø, 21, 22, 23, 24, 25, 3Ø

number

structure indicator	2	t Ø	self-pointer
numeric value			

structure indicator = Ø2, Ø3, Ø4

formal

Ø 5	2	t Ø	self-pointer
Ø	symbol	2 Ø	code-pointer

string

Ø 6	Size	t Ø n	self-pointer	n = no. of characters in last word
C_1	C_2 C_3	C_4	C_5 C_6	
C_7 etc.				

Figure 6. Array Space Structures

Datum

A datum as shown above is one of the following:

number - represented by its numeric value

Boolean - represented by 1 for TRUE, ∅ for false

formal - represented by a symbol-pointer in the CAR, a code-pointer in the CDR, and the indirect bit of tag set (a tag of 2∅)

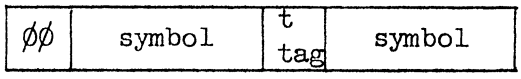
symbol - a symbol datum can represent a string, array, formal, Boolean, number, or a symbolic expression (identifier or list). The representation of a symbol is one of the following:

- NIL, represented by ∅∅∅∅∅∅
- TRUE, represented by ∅∅∅∅∅1
- An octal number q in the range ∅Q≤ q ≤ 177777Q, represented by q + 2Q5
- An integer n in the range -2Q5 < n ≤ 177777Q, represented by n + 6Q5
- A pointer to a list node, an array head, or a character identifier
- A pointer to an identifier, fluid cell, quote cell, or own cell in triple space. Symbol pointers always point to the second word of a triple.

When used as a datum or supplied as the parameter to or value of a function, a symbol is always right-justified into the address (CDR) portion of a word.

2.7 LIST SPACE

List Space consists of a series of list nodes, each of which contains two symbols corresponding to the car and cdr as shown in the following figures:



t = ∅ except during garbage collection, when t_{24} is used for marking nodes.

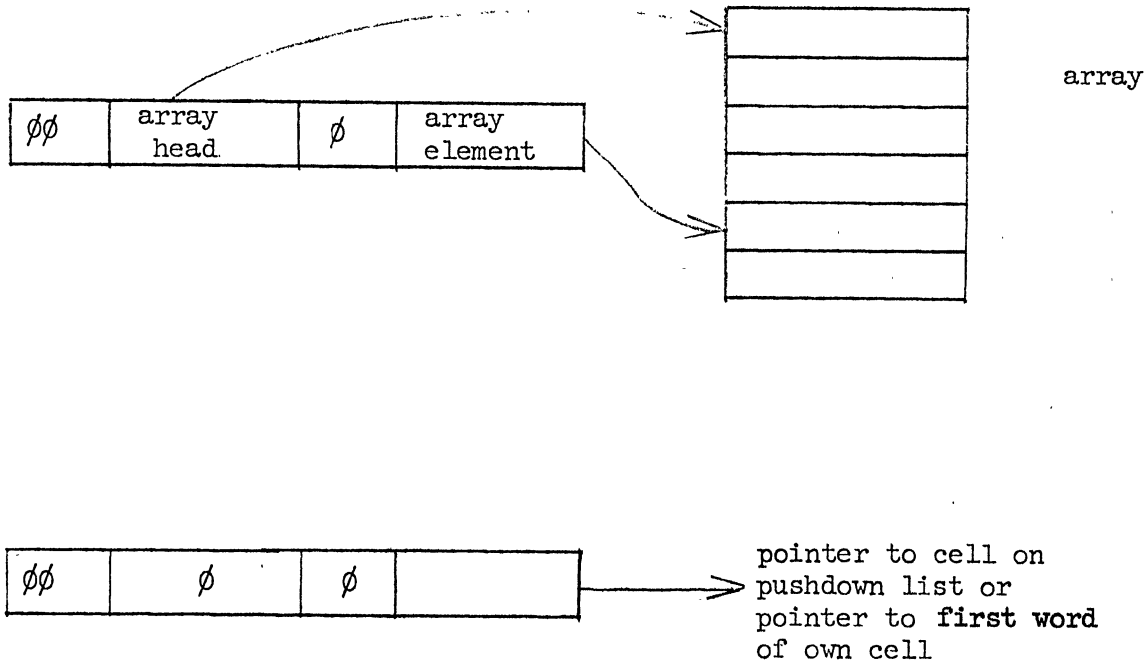


Figure 7. Full-locative

3. TYPE AND STRUCTURE INDICATORS

Type and structure indicators are six-bit codes. Type indicators are used to record type declaration information for variables and functions. Structure indicators are used as identification tags to distinguish types of storage structures. Given a legitimate pointer to a data structure, one can determine its type by looking at the structure indicator.

The current assignment of values for type and structure indicators is given in Table 1. Dashes indicate values which are currently unassigned.

In general, within type declaration information, the coding is as follows: the basic range ØØ - Ør is used for simple types. To each simple type, 1Ø is added to indicate LOC, 2Ø is added to indicate ARRAY, and 4Ø is added to show sub-specification. Since functions and FORMAL variables must be sub-specified, their declarations are described by a sequence of type indicators, as detailed in Section 4.

4. DETAILED USE OF TRIPLE-SPACE STRUCTURE

Triple-space structures include empty triples, quote cells, identifiers, fluid cells, and own cells. Each triple cell structure is identified by the structure indicator and the tag portion of its second word. The structure indicator values of Ø7, 1Ø, 11, 12 and 13 distinguish identifiers, quote cells, fluid cells, own cells, and empty triples, respectively.

The tag occupies bit positions 24 through 29 in the word (counted from Ø at the left end). (Tag-bits will be designated t_{24} through t_{29} left to right.)

Of these bits, t_{24} is used only by the garbage collector and is normally Ø. The remaining bits t_{25} through t_{29} are used in differing fashion depending upon the structure indicator.

4.1 EMPTY TRIPLES

The contents of an empty triple as shown in the figure below is empty (all Ø), except for the link portion of the third word and the structure indicator 13 in the second word.

Ø			
13	Ø	ØØ	Ø
ØØ	Ø	ØØ	link

Table 1. Type and Structure Indicators

Octal Value	Meaning as Structure Indicator	Meaning as Type Indicator
ØØ	list node	SYMBOL
Ø1	-	BOOLEAN
Ø2	octal	OCTAL
Ø3	integer	INTEGER
Ø4	real	REAL
Ø5	formal	FORMAL
Ø6	string	-
Ø7	identifier	-
1Ø	quote cell	(SYMBOL LOC)
11	fluid cell	(BOOLEAN LOC)
12	own cell	(OCTAL LOC)
13	empty triple	(INTEGER LOC)
14	-	(REAL LOC)
15	-	(FORMAL LOC)
16	-	-
17	-	-
2Ø	symbol array	(ARRAY SYMBOL)
21	boolean array	(ARRAY BOOLEAN)
22	octal array	(ARRAY OCTAL)
23	integer array	(ARRAY INTEGER)
24	real array	(ARRAY REAL)
25	formal array	(ARRAY FORMAL)
26	-	-
27	-	-
3Ø	full-locative array	((ARRAY SYMBOL) LOC)
31	-	((ARRAY BOOLEAN) LOC)
32	-	((ARRAY OCTAL) LOC)

Table 1 (Cont'd.)

Octal Value	Meaning as Structure Identifier	Meaning as Type Declaration
33	-	((ARRAY INTEGER) LOC)
34	-	((ARRAY REAL) LOC)
35	-	((ARRAY FORMAL) LOC)
36	-	-
37	-	NOVALUE or INDEF
4ø-44	-	-
45	-	FORMAL sub-specified
46-54	-	-
55	-	FORMAL LOC sub-specified
56-74	-	-
77	-	stop code

4.2 QUOTE CELL

A quote cell contains a single symbol datum in its first word, a structure indicator of 1Ø and a count of Ø1 in it second word, and all zeros in the third word, as shown in the figure below.

	Ø		symbol
1Ø	Ø	ØØ	ØØØØØ1
ØØ	Ø	ØØ	Ø

4.3 IDENTIFIER TRIPLES

The structure of an identifier triple is shown in Figure 8. It is a triple whose second word resembles a character identifier, except that t_{25} is Ø

Other bits of the tag are used to designate genids and to describe the relationship of the first word to the printname (pname) of the identifier. The third word contains a link used, as described in Section 2, to chain the identifier buckets together. It also contains a count of the number of assembled code references to the identifier. The identifier can be reclaimed by the garbage collector if, at any garbage collection, the count is zero, the property list is NIL, the v-f-chain is empty (self-pointer), and the identifier is not pointed to from collectable list structure.

If t_{27} = Ø, the pname of the identifier is contained in the first word, and the tag of the third word contains the number of characters in the pname. If t_{27} = 1, only the first three characters of the pname are in the first word, and the pname is a string pointed to by the first word, as shown in Figure 9.

The bit t_{26} of the tag is used to indicate an identifier of unusual spelling, i.e., FALSE, NIL, TRUE, or any identifier whose print name is not a letter followed by a sequence of letters, digits and dots. The identifiers FALSE, NIL, and TRUE, which must be input as %#FALSE#, %#NIL#, and %#TRUE#, respectively (where % is the escape-character) have a count ≥ 1 so they cannot be collected by the garbage collector, and have bit t_{26} set. The "empty string" identifier %## has bit t_{26} set and has Ø in its first word and a tag of Ø in the third word. An identifier can be protected from garbage collection by means of the count.

4.4 FLUID CELLS AND OWN CELLS

An identifier can have on its v-f-chain at most one fluid cell or own cell for any given section. Fluid cells and own cells are shown in Figure 10. Fluid cells are used to hold fluid bindings, particularly function descriptors. A fluid cell contains a full-locative in its first word, and a structure

Character identifier

∅7	v-f-chain	t tag	property list

t_{24} = 1 (permanently tagged for use by garbage collector)

t_{25} = 1 meaning character identifier

t_{26} = ∅ for A-Z (identifiers with standard spelling)
1 for other characters (identifiers with unusual spelling)

Identifier triple

	print-name-information			first word
all pointers →	∅7	v-f-chain	t tag	property list	second word
	∅∅	count	∅n	link	third word

t_{24} used by garbage collector, normally ∅

t_{25} = ∅ meaning not character identifier

t_{26} = ∅ for identifier with standard spelling
1 for unusual spelling

t_{27} = ∅ pname in triple (no p-name array)
1 p-name array exists (pointer in first word)

t_{29} = 1 for genid (generated identifier)
∅ for normal identifier

∅n = number of characters in first word if t_{27} = ∅

Figure 8. Identifier

identifier pname ≤ 6 characters

C_1 C_2 C_3		C_4 C_5 C_6	
∅7	v-f-chain	t tag	property list
∅∅	count	∅n	link

n = no. of characters in pname (0 for unnamed GENIDs)

$t_{27} = \emptyset$

C_i for i > n are all ∅'s

identifier pname > 6 characters

C_1 C_2 C_3		∅∅	pname array
∅7	v-f-chain	t tag	property list
∅∅	count	∅∅	link

$t_{27} = 1$

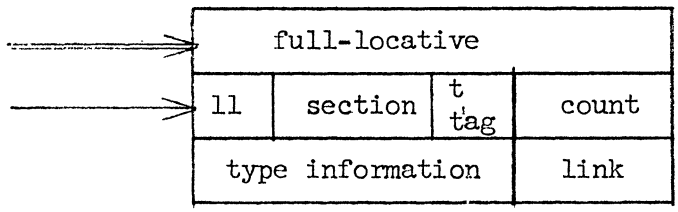
∅6	Size	∅n	self-pointer
C_1 C_2 C_3 C_4 C_5 C_6			
C_7 etc.			

n = no. of characters in last word

in array space

Figure 9. Pname of Identifiers

Fluid Cell

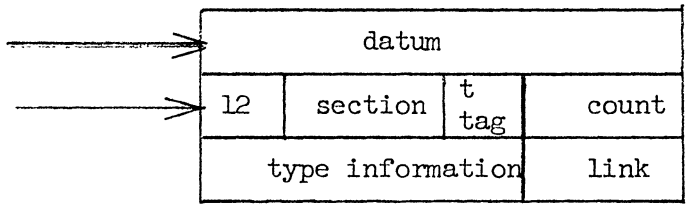


t_{24} used by garbage collector, normally $\emptyset$

t_{25} = 1 if FLUID declarative exists, $\emptyset$ otherwise

t_{26} = 1 means never collectable by garbage collector
$\emptyset$ means garbage collection possible

Own Cell



t_{24} used by garbage collector, normally $\emptyset$

t_{25} = 1

t_{26} = 1 means never collectable by garbage collector
$\emptyset$ means garbage collection possible

⟹ = reference from code

⟶ = symbol or link pointer

Figure 10. Fluid Cell and Own Cell

indicator of 11 in its second word. An own cell contains its datum directly in its first word, and a structure indicator of 12 in its second word. The contents of the second and third words are similar for both kinds of triples.

The second word contains the structure indicator of 11 or 12, the section name (NIL or an identifier), and a count of the number of code references to this fluid or own cell. The tag bits are not used, except for t_{24}, which is used by the garbage collector; t_{25}, used to designate a variable for which a top-level FLUID declarative exists; and t_{26}, used to designate a fluid or own cell which is never collectable by the garbage collector.

The third word contains type information and a link.

The contents of a full-locative are shown in Figure 7. A full-locative may point to a word on the pushdown list or to the first word of an own cell, in which case it consists of a single pointer. Alternatively, a full-locative may point to an element of an array, in which case it contains two pointers. In particular, the full locative contained in a fluid cell of a variable whose transmission mode is not LOC) is initialized at the top level to point to the first element of a unique one-element array of the same type as the variable. This array is used to hold top-level free settings of the fluid variable.

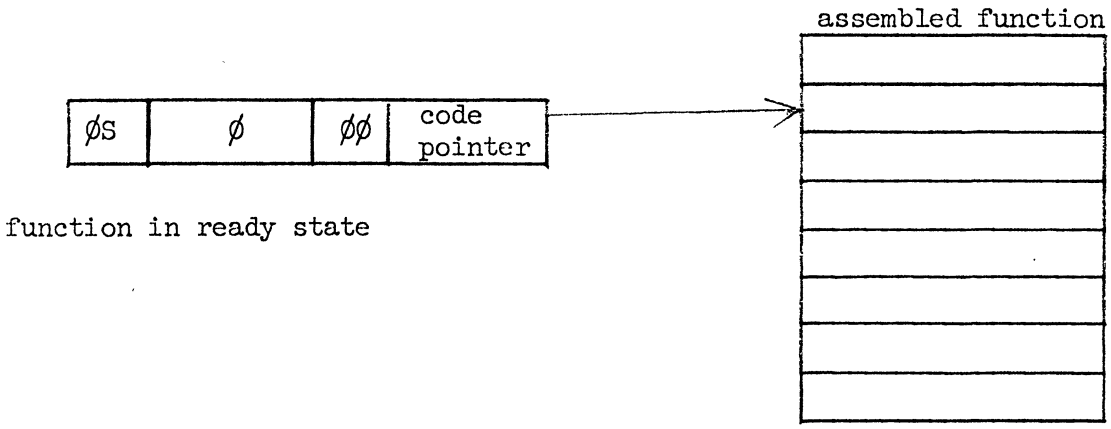
4.5 OWN FORMAL

An OWN FORMAL cell contains a formal pointer as datum in its first cell (see Figure 7), and so has a prefix of ØØ in the datum, which distinguishes an OWN FORMAL from a function descriptor.

4.6 FUNCTION DESCRIPTOR

The function descriptor contained in the first word of an own cell can exist in one of three states, as shown in Figure 11. A normal function in ready state, i.e., one than can be operated directly, contains a code pointer in the address portion of the function descriptor, a tag of Ø, a decrement of Ø, and a prefix of 1 for a FUNCTION, 2 for a MACRO, or 3 for an INSTRUCTION.

A function with a formal trap, e.g., a function that is being traced, has the code pointer in its decrement, a formal trap in its address portion, and the indirect bit in the tag is set, so that branches through the function descriptor will go indirectly to the formal trap. A formal trap is simply a pointer to another function descriptor. The prefix of 1S indicates this condition, with S having the same meaning as for a ready function.



S = 1 for FUNCTION
2 for MACRO
3 for INSTRUCTIONS

Figure 11. Function Descriptor (first word of own cell)

The third case of a function descriptor, for an unready function, has a prefix of 2S and a symbol in its decrement portion. The prefix 2S is used by the garbage collector to indicate that the left half of this word is to be marked during garbage collection. The symbol is used to hold information as to the location of symbolic code for this function, and the formal trap points to another function which is to be used to obtain or compile the unready function. S has the same meaning as for a ready function.

Type Encoding

Type encoding is contained either directly in the third word of a fluid or own cell or indirectly in an array or triple cell pointed to by the type information. The various possibilities are distinguished by the value of the prefix, as shown in Figure 12.

A prefix of $\emptyset\emptyset$ indicates a type other than FORMAL where no sub-specification is required. In this case, a single type indicator, contained in the tag of the word, is used. The rest of the world is $\emptyset$ except for the link.

The prefix $\emptyset 2$ is used to indicate that the fluid cell or own cell is a synonym. In this case, the decrement of the third cell contains a pointer to another fluid cell or own cell, in which the type and value are to be found.

A fluid or own FORMAL, or an own cell used as a function descriptor, and used for a function of fewer than 3 arguments has its type information encoded directly in the third word. The prefix of 45 shows that a sub specified FORMAL is to be represented. The coding $f_1\ f_2\ f_3\ f_4$ is used to specify the type.

The prefix $\emptyset 1$ is used for fluid or own FORMALs and own cells which require more than five type indicators to encode their type information. In this case, the decrement of the third word contains a pointer to an octal array (structure identifier = 22) which then contains the type coding.

The type coding of a formal or function is as follows:

$f_\emptyset$ is 45 for a function or FORMAL
55 for a FORMAL LOC

f_1 specifies value-type

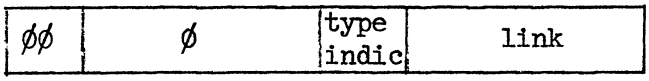
37 means NOVALUE

$\emptyset\emptyset$ - $\emptyset 5$ mean SYMBOL, BOOLEAN, OCTAL, INTEGER, REAL, FORMAL

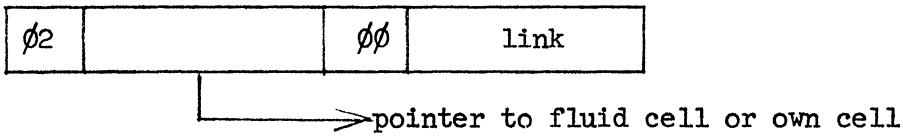
f_2 specifies type of first argument

77 means no arguments
37 means INDEF with type given by f_3
$\emptyset\emptyset$ - $\emptyset 5$, $1\emptyset$ - 15 mean parameter type, according to Table 1.

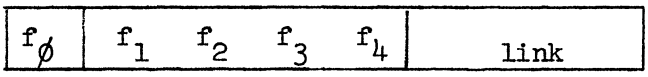
Fluid or own cell of other than FORMAL type



Synonym

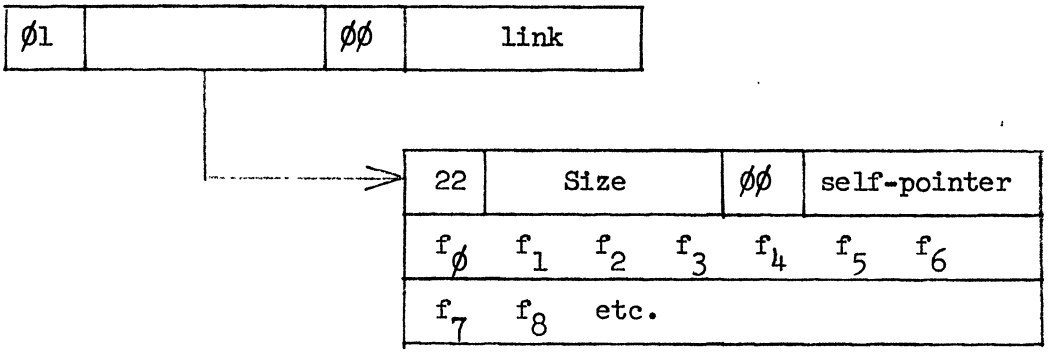


Fluid FORMAL or own function descriptor or own FORMAL (Ø, 1, or 2 args)



$f_Ø$ = 45 for function or FORMAL
55 for FORMAL LOC

Fluid FORMAL or own function descriptor or own FORMAL (3 or more args.)



$f_Ø$ = 45 or 55

Figure 12. Type Information

f_3, f_4 ... may be

ØØ - Ø5, 1Ø - 15, which mean parameter types, according to Table I.

The stop code 77 means that there are no more arguments. Hence a function of n arguments requires n + 3 f's if the first argument is INDEF, or n + 2 f's if the first argument is not INDEF. However, the stop code is not required if the type information completely fills its allotted space. Hence, the third word of a triple can encode a formal or function containing up to 3 arguments (2 if INDEF), and an array of n cells can store type information for a function of 6n - 8 arguments (or 6n - 9 if INDEF).