1979

Author: T. Knight

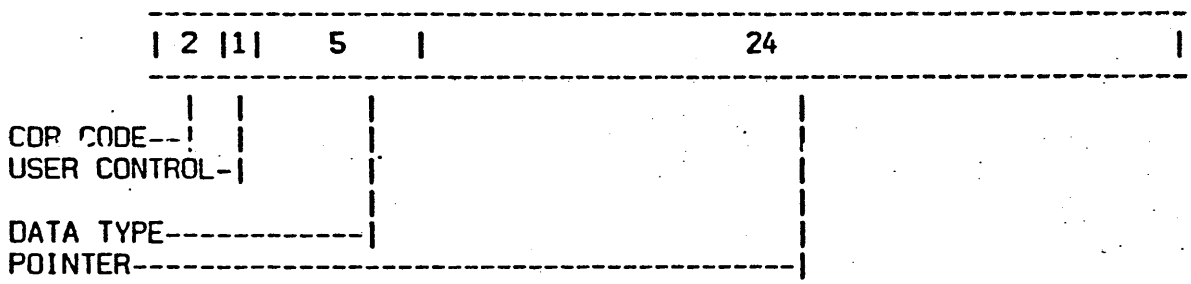
Lisp Machine Internal Storage Formats:

Q formats, SYMBOL formats, ARRAY formats, PDL formats, LINEAR BINDING PDL formats, STACK GROUP formats, FEF formats, AREA formats, CALLING CONVENTIONS, and ADI formats.

> 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 rversal 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

The formats are not in exactly this order. Also, it is hard to understand the macro-code instruction set without first understanding the FEF format, and vice-versa; they are very closely related. It is assumed that the reader of this document has read the MACROCODE document (for the FEF formats) and is at least somewhat familiar with the workings of the CONS machine.

Lisp objects in the LISP machine are stored in the following form:




CDR CODE field (2 bits) - This field shows where the CDR of this object is:

0 - CDR NORMAL: The CDR is contained in the Q following this one. This is the "two pointers" form used by most Lisps.
1 - CDR NIL: The CDR of this node is NIL.
2 - CDR NEXT: The CDR is the next Q.
3 - CDR ERROR: It is an error to take the CDR of this location, since this is the second half of a full (CDR NORMAL) node.

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 (see below).

USER CONTROL BIT field (1 bit) - This bit is not used by the system in normal list structure, and is thus available for use by the user. In cells which are not part of normal list structure, though, the system may use the bit. (For example, it is used in indexed-offset arrays.)

DATA TYPE (5 bits) - This field determines the data type of the Q. Since each Q has a separate data type field, there is no need for "fixnum space," "list space," etc. The datatypes are:

| NUMBER | NAME | USE |
|---|---|---|
| 0 | DTP-TRAP | Any attempt to reference this cell will cause a trap. This is mostly for error checking (maybe also for debugge |
| 1 | DTP-NULL | This datatype is used for various things to mean "nothing For example, an unbound atom has one of these as its valu The pointer field points back at the atom, for ease in debugging. |
| 2 | DTP-FREE | This cell is free unallocated storage. The user should n see this too often. |
| 3 | DTP-SYMBOL | This is a non-numeric atom. The pointer points to a four Q "atom header" (see SYMBOL formats). |
| 4 | 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, unl PDP-10 MACLISP. |
| 5 | DTP-EXTENDED-NUMBER | Any type of number other than FIXnums. Not in yet. |
| 6 | DTP-INVOKE | This has the peculiar property that if anything tries to perform a fundamental operation on it (such as CAR, CDR, RPLACA, RPLACD, or CHECK-DATA-TYPE) the invoke pointer gets called as a function. This feature is not yet fully developed, so stay tuned for further developments. |
| 7 | DTP-GC-FORWARD | The forwarding address left behind by the garbage collect |
| 10 | DTP-SYMBOL-COMPONENT-FORWARD | This causes indirection when used by operat which operate on symbol components. |
| 11 | DTP-Q-FORWARD | Forwards only the Q that it is in, not the whole structur |
| 12 | DTP-FORWARD | Forwarding address left behind by anything which copies something other than the garbage collector. |
| 13 | DTP-MEM-POINTER | (going away. only used on simulator) |
| 14 | DTP-LOCATIVE-TO-LIST | |
| 15 | DTP-LOCATIVE-INTO-STRUCTURE | |
| 16 | DTP-LOCATIVE-INTO-SYMBOL | |
| 17 | DTP-LIST | The pointer points to a list (actually, to a node). |
| 20 | DTP-LIST-INTO-STRUCTURE | |
| 21 | DTP-LIST-INTO-SYMBOL | |
| 22 | DTP-U-ENTRY | The pointer points to a micro-coded function. The pointer field is actually an index into the MICRO-CODE-ENTRY-AREA, which contains a pointer to the actual code. |
| 23 | DTP-MESA-ENTRY | The pointer points to a mesa-compiled routine. |
| 24 | DTP-FEF-POINTER | Points to a FEF-HEADER. |
| 25 | DTP-FEF-HEADER | Header of a Function Entry Frame (see below) |
| 26 | DTP-ARRAY-POINTER | The pointer points to the ARRAY HEADER word of an array This is the equivalent of an "array object." |
| 27 | DTP-ARRAY-HEADER | There is an array header for each array. The pointer fi holds various encoded information about the array (see th section on ARRAY formats). |
| 30 | DTP-ARRAY-LEADER | This datatype is used for the Q at the head of an array leader (see ARRAY formats). |

| | | |
|---|---|---|
| 31 | DTP-STACK-GROUP | See STACK GROUP formats. |
| 32 | DTP-CLOSURE | Super win!!! |
| 33-37 | | Not used at present. |

POINTER (24 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.

Note that some of the datatypes are useful mostly for their meaning in "function context" (see SYMBOL formats).

; The invisible pointer datatype is a one of the LISP machine's
;unique new features. They are like indirect addressing where instead of the
;instruction specifying the indirectness, the data referenced does! Thus if
;you take the CAR of a Q which is an invisible pointer, you will really be given
;the CAR of what the pointer POINTS TO. The possibility of implementing
;invisible pointers from the system design point of view depends on the fact that this
;is an INTEGRATED system. For example, in a conventional machine, it would clearly
;be unacceptable to have a range of numbers such that when they are added
;together with the ADD instruction, what gets added is not these instructions
;themselves, but the contents of the memory location they point to. Here, that
;is exactly what happens.

Some of the proposed uses for invisible pointers arte described in the paper "The LISP Machine" [Grenblatt 74, A.I. Woking Paper 79] and in the LISP machine progress report [whatever].

SYMBOL FORMATS:

A symbol is stored as a Q of datatype DTP-SYM whose pointer points to a four Q "atom header." The four words are:

| NAME | USE |
|---|---|
| PRINT-NAME-CELL | This cell holds a word of datatype ARRAY-POINTER pointing to a STRING array which is the PNAME for the symbol. (See ARRAY formats). |
| VALUE-CELL | This cell holds the value of the symbol, and so can be of any datatype. |
| FUNCTION-CELL | This 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. Note that this replaces the purpose of the "EXPR," "SUBR" etc. properties in Maclisp. |
| PROPERTY-CELL | This cell contains the property list. Properties are not used by the basic system at all, so this is likely to be NIL. |

When a symbol is initially created, the value and function cells contain null data type.

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

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 "function context." When a symbol is used as the destination of a CALL instruction, or the first argument to APPLY, its FUNCTION CELL is likewise examined and the contents considered in function context.
Here is what some of the datatypes mean in function context:

| DATATYPE | MEANING IN FUNCTION CONTEXT |
|---|---|
| LIST | This should be handled by the interpreter. Usually the list is a LAMBDA expression. It can also be a MACRO expression. |
| SYMBOL | This means that the contents of the function cell of the specified symbol should be used as the function. |
| FRAME POINTER | This function is macro compiled, so use the FEF pointed to (see FEF formats). |
| MICRO-CODE-ENTRY | This function is micro-compiled. |
| MESA-CODE-ENTRY | This function is MESA compiled, do that stuff. |
| ARRAY-POINTER | This function is an array. Array referencing is handled by the microcode, so there is no "code" associated with an array. |
| STACK-GROUP | Transfer control to the designated stack group. |

ARRAY FORMATS:

Every array has an ARRAY HEADER word. The pointer field is divided into fields which hold various information about the array. The array may optionally have an ARRAY LEADER which is formed of a number of words BEFORE the array header. 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 datatype (since any object can be stored there), and before that is a word of datatype ARRAY LEADER which is a self-relative pointer to the ARRAY HEADER. The presence of the ARRAY-LEADER Q is necessary for such routines as the garbage collecter which scan through memory in the usual direction. The presence or absence of the leader is determined by a bit in the array header.

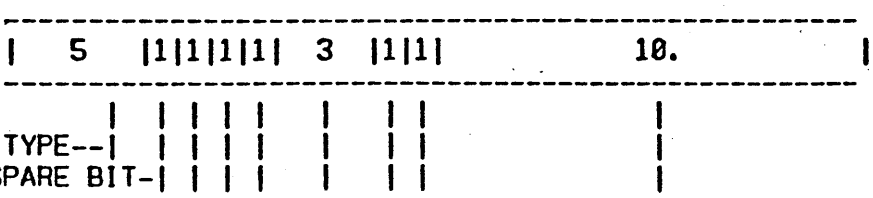
If the array has more than one dimension, then there is a block of <number of dims>-1 Q'S immediately after the array header holding the size of each dimension. Note that only <number of dims>-1 are needed because one can compute the total index length from the array header itself.

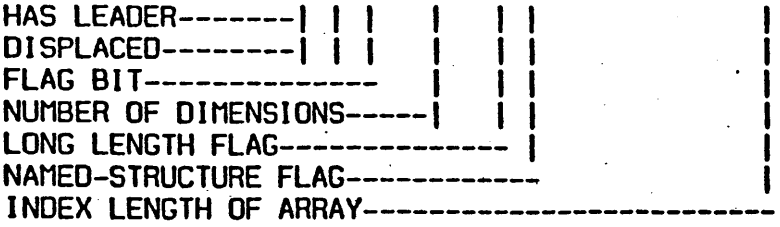
If the index length of the array (number of data elements) is too big to fit in the field allocated for it in the array header Q, an extra Q is inserted between the header and the dimensions, which has data type FIXNUM and contains the index length. A bit in the header Q is on to indicate the presence of this extra Q.

Now all that is left are the actual storage cells of the array. An array may optionally be "displaced," according to a bit in the header. If the array is not displaced, then the data Q's 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 Q is actually a pointer to the data cells. Thus, a displaced array can be used to point at the beginning of an area (this is done often, in fact). Following the displacement word, in what would have been the SECOND data cell, is the length of the data in Q's for the array. This is used instead of the normal index length, since that will be 2 (or 3) to indicate the length of the pointer. This SECOND data cell is used as the length even in the case of indirect arrays, unless that would cause a reference off the end of the array indirected to.

Further hair is provided as follows: if the array is displaced and the word which would be the pointer has datatype ARRAY POINTER, then it points to another array header! This is called an INDIRECT array. If that isn't hairy enough, get this: If the USER CONTROL bit of the indirect array pointer is set, then this array has an INDEX-OFFSET from the array pointed to. This means that whenever this array is referenced, it is as if that array were referenced, but with an index <n> higher. The <n> is the offset, and is stored as a FIXNUM in what would be the THIRD data cell if this array were non-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). (Note that the length of the array being pointed at is also stored, in that arrays' header, etc. When a reference is made to an INDIRECT array, an error check is performed to make sure the reference is not out of bounds.)

The format of the pointer field of the header word is as follows:






The FLAG BIT, in the case of a string array, is 1 to indicate that this string may be relied upon to contain only ordinary printing characters. Its use with other array types is not yet defined. (THIS IS AN EFFICIENCY HACK, WHICH IS CURRENTLY IGNORED).

The %%ARRAY-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-BIT is set, otherwise array element 0.

Named structures may be viewed as implementing a sort of user defined data typing facility. Certain system primitives, if handed a NAMED-STRUCTURE, will obtain the name and obtain from that a function to apply, ACTOR like, to perform the primitive. One can see that there is some potential

The only one of these fields which has not yet been mentioned is the ARRAY TYPE field. The options are:

| NUMBER | TYPE | USE |
|---|---|---|
| 0 | ART-ERROR | This is always an error, to prevent randomness. |
| 1 | ART-1B | Each element is one bit, and 32 are stored per word. |
| 2 | ART-2B | Analogous. |
| 3 | ART-4B | Analogous. |
| 4 | ART-8B | Analogous. |
| 5 | ART-16B | Analogous. |
| 6 | ART-32B | Analogous. Since FIXNUM datatype is supplied 24 bits of data are retrievable. |
| 7 | ART-Q | Each element is a Q, that is, it has a datatype and a pointer field. |
| 8 | ART-Q-LIST | Same as Q, but the elements also form a list. By using GET-LIST-POINTER-INTO-ARRAY and G-L-P, you can get pointers into the beginning or even the middle of such an array. |
| 9 | ART-STRING | This is stored the same way as an 8 BIT array. |
| 10. | ART-STACK-GROUP-HEAD | (see STACK GRUOP FORMATS) |
| 11. | ART-PDL-SEGMENT | (see STACK GROUP FORMATS) |
| 12. | ART-TVB | TV Buffer |
| 13. | ART-TVB-PIXELS | TV Buffer in pixel mode. |

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

However, TV buffer arrays (ART-TVB) are DIFFERENT, for hardware reasons. Only the bottom 16 bits of each word are used, and the bits are stored left to right.

TV-BUFFER-PIXEL arrays have a plane mask in array leader element 0. 1 bits in the plane mask correspond to active tv-buffer planes, 0 bits to inactive

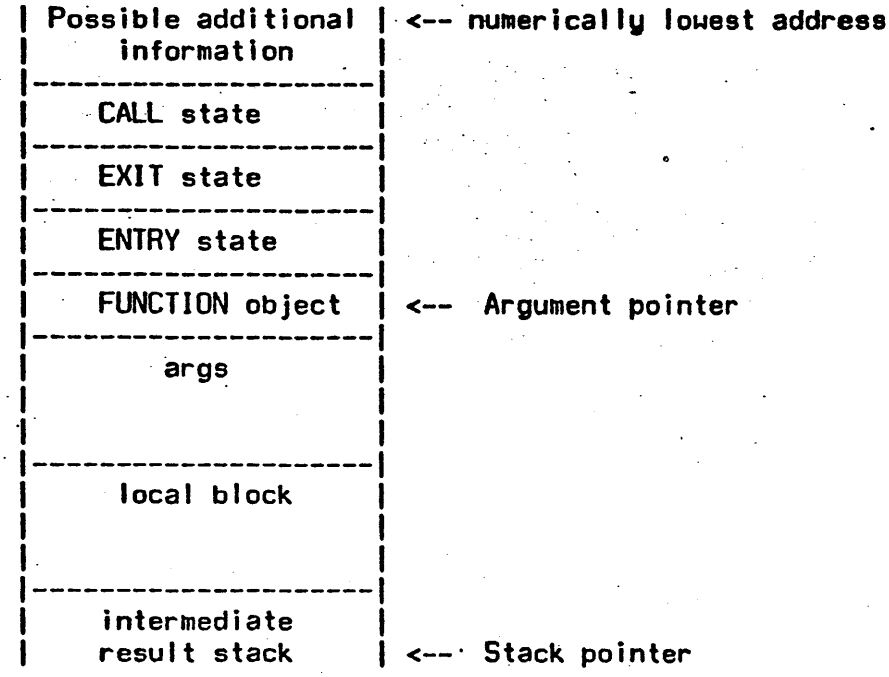
planes. Each time a active plane is encountered on a store, the low order bit is stored in that plane (a la ART-TVB), and the remaining bits shifted right one.

STRING arrays are stored the same way as Q-ARRAYs, and STACK-GROUP-HEAD and STACK-SEGMENT arrays are stored the same as Q-ARRAYs are. The reason for supporting both array types is so that programs can easily tell apart those 8-bit arrays used for strings, etc. Strings, although like 8-BIT arrays at low levels, are treated differently at higher levels, such as by READ, EVAL, and PRINT.

PDL FORMATS:

The stack in the LISP Machine is stored in Main memory, with the top kept in the PDL BUFFER memory of the CONS Machine. (The PDL Buffer acts as a sort of 1K cache which greatly speeds up almost all references to the stack. The "swapping" is done in micro-code, invisibly to the macro-code and all higher levels.)

For each function call, a CALL BLOCK is stored on the PDL. The format of a call block is:




The "possible additional information" (ADI) is used by certain hairy types of calls which need to convey more information.

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 "TO NEXT" and "TO LAST" are executed. When the block is activated (see below) space is reserved for that block's local variables (i.e. PROG and DO variables).

Each CALL instruction creates a new open block, and stores in its CALL state word the delta (offset) to the ACTIVE block at the time of the CALL (i.e., the function which called it) in the low 8 bits. This is used to restore M-AP when leaving the function. It also stores a delta to the previous OPEN block (just the previous block on the stack) in the next 8 bits. 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-FRAME, that is) (when a macro-compiled function is being called). Further, CALL stores its DESTINATION field in a three bit field in the CALL state word, so that when the called function returns, its result can be stored in the correct place.

When something is stored in destination "TO LAST," the current open call block (the last block pushed) is ACTIVATED. The currently active block's PC is stored in the EXIT PC (the return address) in that block's EXIT state word, and the PC is set to the starting address of the new function

(see FEF formats). Also stored in the EXIT state word is the BINDS-PUT-ON-BINDING-PDL bit (see LINEAR-BINDING-PDL formats). 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 "TO RETURN," the current block is finished with. The micro code 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.

Note that the way the stack and the macro-instruction set are set up, to refer to its args the function need never reference indexed "negatively off the stack pointer." That is, a function with five args doesn't refer to its second arg by -3(SP). Thus any function does refer to its second arg by 2(AP) regardless of the total number of args 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:

In the CALL state word: (%%LP-CLS-)

CLOSURE-BINDING-BLOCK-PUSHED: An extra binding block was pushed because this was a closure invocation.
ADI-PRESENT: There is ADI present (see ADI formats).
DOWNWARD-CLOSURE-PUSHED:
MACRO-SAVED-DESTINATION (3): Saved DESTINATION field of the CALL instruction.
DELTA-TO-OPEN-BLOCK (6): Delta on stack to previous open block.
DELTA-TO-ACTIVE-BLOCK: ditto

In the EXIT state word (%%LP-EXS-)

BINDING-BLOCK-PUSHED (1): The QBBFL bit in M-FLAGS
and either
EXIT-PC (17): The saved PC, if we are macro code, or
RETURN-MICRO-PC (14): if we are micro code

In the ENTRY state word (%%LP-ENS-)

NUM-ARGS-SUPPLIED (6): Number of args passed to us.
And either
MACRO-LOCAL-BLOCK-ORIGIN (8): Offset from call state to local block? M-QLOCO.
or U-MICRO-STACK-XFER-COUNT (8): Number of words transferred form ustack to specpdl before this call.

Historical note: In the "LISP Machine" paper (W.P. 79) there is much talk of PDL Frames and Frame Pushdown List mode. This feature has not been implemented, and probably will not be. The original datatype FRAME POINTER would have been used for these PDL frames as well as Function Entry Frames (FEFs) and others, each of which would have been a page long. However, this does not reflect the current state of implementation.

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, and the macrocode itself. One of the things which must be kept handy is the manner in which the function interprets its arguments. There must be provision for storing very complex, hairy specifications, such as whether each arg is to be EVALed or not, whether it 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, hairy format were always used. The solution to the problem is that simple functions use only a "Numeric Argument Description" word and a "Special Variable Bit map" (the third and fourth words in the FEF) to store this information, while more complicated functions use the more general "Argument Descriptor List" (ADL). Note that the ADL is, confusingly, sometimes called the "Binding Descriptor List" or BDL; this should eventually be fixed.

The exact (ha ha) way this works is as follows:
There is one bit in the FEF which tells whether the ADL is present. If it is not present, then (of course) it is not used, and presumably the format is simple enough to be conveyed through the information in the Numeric arg description and the S.V. Bit map word. Even if the ADL is present, it may not be used (and only be there for debugging).

There is a bit specifying that there are special variables being bound by this function. If this bit is set, then the information about which args and/or locals are special will be found either in the S.V. Bit map word, or in the ADL, as follows:

The S.V. 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 S. V. bit map word IS active, it is interpreted by considering it as (of course) a bit map, in which the least significant bit corresponds to the first variable, etc. (??? fraid its the most significant. 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.

In all FEFs, the Numeric Argument Description List will be present and accurate [he he]. It has the following fields:

QUOTED-REST: (1 bit) There is a &REST arg, and it is quoted.
EVALED-REST: (1 bit) There is a &REST arg, and it is EVALed.
Note: These two may not be on together, of course.
FEF-QUOTE-HAIR: (1 bit) There is hairy quoting, the FEF must be checked by EVAL.
INTERPRETED: (1 bit) This is an interpreted function.
Note: This will never be on in the FEF, but other kinds of functions use this format also.
FEF-BIND-HAIR: (1 bit) There is hairy binding, the ADL must be checked by the linear enter routine (don't worry about what that is).
MIN-ARGS: (6 bits) The minimum number of required args.
MAX-ARGS: (6 bits) The maximum number of required + optional args.
Note that neither of these two six-bit fields include the rest arg, if any; it was covered by the first two bits.

When the ADL is used:
If the FEF-QUOTE-HAIR bit is set, or the FEF-BIND-HAIR bit is set, or if the "S.V. bit map active" bit is clear and the Special Variables Present 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 which is semi-historical. If it is set, it is a guarantee <ho ho> 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.

The fields of the first Q are:

NAME-PRESENT: (1 bit) There is a second word containing the name of this variable.

SPECIAL-BIT: (1 bit) This variable is special; get a pointer to its 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.

DES-DT: (4 bits) Desired datatype for this variable, which may be (in numeric order starting with zero)

| | |
|---|---|
| DT-DONTCARE | We don't care what we get. |
| DT-NUMBER | Any number. |
| DT-FIXNUM | Only FIXNUM. |
| DT-SYM | Only SYMBOL. |
| DT-ATOM | Any number or symbol. |
| DT-LIST | Only LIST. |
| DT-FRAME | Only FRAME (i.e. FEF. This is pretty random...) |

QUOTE-STATUS: (2 bits) The desired quotage/evalage of the argument, which may be: (not implemented in any case)

| | |
|---|---|
| QT-DONTCARE | We don't care what we get. |
| QT-EVAL | Should be EVALed. |
| QT-QT | Should be QUOTEd (not EVALed). |
| QT-BREAKOFF | Should be the name of a function compiled from a quoted lambda expression. |

ARG-SYNTAX: (3 bits) The desired arg syntax, which may be:

| | |
|---|---|
| ARG-REQ | Required. |
| ARG-OPT | Optional. May be initialized if arg not present. |
| ARG-REST | Rest arg. (There may only be one.) |
| -- below here, they're not really arguments | |
| ARG-AUX | Prog-variable. May be initialized. |
| -- below here, they're ignored by the function entry operation | |
| ARG-FREE | Variable is referenced free. Included merely because this might be a nice thing to be able to determine sometime. Totally unnecessary to actual execution of function. |
| ARG-INTERNAL | cell used to pass an argument to an internal LAMBDA. |

ARG-INTERNAL-AUX cell used by an internal PROG.

INIT-OPTION: (4 bits) The desired initialization of this variable, which may be:

| | |
|---|---|
| INI-NONE | Do not initialize (all required args have this.) |
| INI-NIL | Initialize to NIL. (The default for locals.) |
| INI-PNTR | Initialize variable to 3rd Q. |
| INI-C-PNTR | Initialize variable to what 3rd Q points at. |
| 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 and thus must be initialized.) |
| INI-COMP-C | Variable initialized by compiled code. Initialization too hairy to be done by above mechanisms. |
| INI-EFF-ADR | Interpret 3rd Q as macro-code effective address (i.e. 3 bit register, 6 bit delta). Reference tha adr and initialize variable to what you get. (This is used to compile (LAMBDA (A &OPTIONAL (B A)) ..) with A and B local, for example.) |
| INI-SELF | Initialize to self. used for (LAMBDA (&OPTIONAL (FOO FOO)) ..) which isn't reasonable unless FOO is special. |

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).

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

And now, here is the FEF format:

There are first seven words of various information about the function. The first word contains the initial PC, relative to the top of the FEF (i.e. itself), which points to the macrocode for the function, which is stored at the end of the FEF. It also contains three one-bit fields which have already been discussed:
The NO-ADL-PRESENT bit, the FAST-ARGUMENT-OPTION-ACTIVE bit, and the SPECIAL-VARIABLES-PRESENT bit.

The second word is the function name. This is only here for debugging.

The third word is the Numeric Argument Description word, and the fourth

is the S. V. bit map word.

The fifth word has three fields:
The low 7 bits: The size of the Local block. (When the function is activated, this many words will be reserved on the PDL.)
The next 8 bits: The location of the ADL relative to the start of the FEF.
The next 8 bits: The number of entries on the ADL (the number of variables described; there may be one or two words per variable).

The sixth word has one 8 bit field which holds the maximum length of the local block plus any pushing the function might do. This is here for use by the microcode which swaps the PDL in and out of the main memory, so that it can assure that there will be room for execution of the function.

The seventh word contains the total size of the FEF.

Then, after these seven words, are the S. V. Value Cell Pointer List (if any), the ADL (if any), the space for random constants used by the program (if any), and finally, the macrocode itself, packed two instructions per word.

LINEAR BINDING PDL formats:

The LINEAR BINDING PDL (LBP) corresponds fairly closely with the SPECPDL in PDP-10 MACLISP. The LISP machine uses shallow-binding, so the current value of any symbol is always found in the symbol's value cell, and when a symbol is bound, its previous value is saved on the Linear binding PDL, and the new value is placed in the value cell. (Note, however, that the use of the linear binding pdl is bypassed in the simple cases through the mechanisms described on the preceding page.)

The LBP also serves some other functions. When a MICRO-TO-MACRO call is made, the "MICRO-PDL" of the Cons machine is stored there (this is needed because the hardware micro-PDL is only 32 words long).

Note: When discussing the LBP, "first" means the location with the numerically highest address, and thus the LAST word pushed. The "last" word is actually the FIRST pushed. Oh, well...

The LBP is block oriented. The blocks are delimited by setting the USER CONTROL bit in the last Q in each block (i.e. the first one pushed). The datatype of the first word of each block determines what kind of block the block is, as follows:

| DATATYPE | USE |
|---|---|
| LOCATIVE | The block is a normal binding block. |
| FIXNUM | This is a block transferred from the CONS machine 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. |
| MESA ENTRY | This is a MESA code leave block. The block should be 2 Q's long; the first one (the MESA ENTRY type Q) is a pointer to the MESA-FEF left, and the second is the saved MESA-PC (the return address). (See MESA-CODE formats.) |

A normal binding block is stored as a pair of Q's for each binding; the first Q is a LOCATIVE pointer to the bound location, and the second is the saved contents of the location. Note that any location can be bound; usually these locations will be the value cells of symbols, but they can also be array elements, etc. (only of arrays of type Q-LIST).

The SPC blocks and the MESA code leave blocks are always pushed onto the LBP 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 LBP. This bit is saved during MACRO-TO-MACRO calls (see CALLING conventions) on the regular PDL in the EXIT state word (see PDL formats) so that when a MACRO-compiled function is done, a binding block will get popped off the LBP. If the bit is 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.

The LBP is pointed to by the location QLBNDP in LMI, and by

A-QLBNDP in the real machine. In the current setup there is an area devoted to storing the LBP called LINEAR-PDL-AREA.

AREA formats:

Areas don't have much of a format, mostly, but there are still some interesting things to say about them.

There are several areas which are important to the basic keeping track of the other areas; in the nuclear system, the atoms with their names have as their properties arrays which point at the areas, so that they can be easily referred to. These are: AREA-NAME, AREA-ORIGIN, AREA-LENGTH, AREA-FREE-POINTER, AREA-PARTIALLY-FREE-PAGE, AREA-FREE-STORAGE-MODE, and AREA-FREE-PAGE-LIST. The uses of these are documented in LMNUC, sections 3.6.X . Each area has a number (simply numbered 0 and on up) which is used to index into theses areas.

There are several ways free space may be allocated within an area; for each area the storage alocation mode is given by the area's entry in the AREA-FREE-STORAGE-MODE area. The ones currently implemented are LINEARLY-ALLOCATED, FREE-LIST, and PAGE-ALLOCATED. All depend on the area's item in the AREA-FREE-POINTER area. In a LINEARLY-ALLOCATED area, the Qs are allocated linearly; that is, one at a time sequentially. The FREE-POINTER points to the next free Q, and so every tim one is allocated, 1 is added to the FREE-POINTER. In order to reclaim storage, the garbage collector would have to compactify. In a FREE-LIST allocated area, the free Qs are kept in a linked list (they have datatype DTP-FREE) and the 23 bit pointer field points to the next free element. Here a garbage collector would have no need to compactify. In a PAGE-ALLOCATED area, allocation is done one whole page at a time. (A page is 200 (octal) words long.) The FREE-POINTER points to the first word of the first page, and the first words of the pages form a linked list.

stack groups, calling convs, adi.
remember: show how Numeric Arg Description Word is used by other than MACRO.
STACK GROUP formats:

A stack-group is the data structure behind the implementation of a "process" in the LISP machine. 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 ther is no time-sharing betwen users on the LISP machine, it is still useful for system-hacking purposes to be able to support multiple processes; for example, when a message is received from the CHAOSnet, 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.

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 micro-code variables, etc.

There is provision (although not initially implemented) for allowing the two PDLs to be stored as a chain of linked arrays rather than just one (so that getting a PDL overflow would not require reallocating a bigger array and copying such an array would be of type ART-PDL-SEGMENT. Both of these two array types are treated by the low-level routines the same as ART-Q arrays.

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.

The elements of the array leader are:

| NAME | USE |
|---|---|
| SG-PDL-PDL-POINTER | Saved PDL pointer, stored as a fixnum offset from SG-PDL-STORAGE-ARRAY. |
| SG-PDL-STORAGE-ARRAY | This points to the the array in the chain which we are now using. |
| SG-LINEAR-BINDING-ARRAY | Points to array for LBP. (??? which is ART-what?) |
| SG-LB-PDL-POINTER | PDL pointer to LBP, stored as a fixnum offset from SG-LINEAR-BINDING-ARRAY. |
| SG-PDL-OVFL-SECTION | PDL overflow section chain (???) |
| SG-LB-OVFL-SECTION | Bindin PDL overflow section chain (???) |
| SG-U-STACK-QS | Number of Qs transferred to micro-stack on switchout (???) |
| SG-INITIAL-FCTN-INDEX | Position in SGBA (which is what???) of the topmost function pointer cell. This is normally 3, but may difer if ADI is present. |
| SG-UCODE | Used somehow (not in yet) to indicate what microcode packages this stack group requires to be loaded. |

The following Qs "hold the state" across macro-instruction boundaries:

| | |
|---|---|
| SG-AP | Points to currently running block on the stack in this st group, stored as a fixnum offset to SG-PDL-STORAGE-ARRAY. |
| SG-IPMARK | Points to currenlt open block on the stack. Stored the s |

way.

SG-SAVED-QLARYH (i.e. the last array referenced.)
SG-SAVED-QLARYL (i.e. the last element of an array referenced.)
SG-SAVED-NARGS (i.e. number of args.) This enables one to compute how much of the PDL beyond SG-IPMARK is eveluated args, and how much is temp storage.
SG-SAVED-INDICATORS (i.e. PC flags, condition codes, etc.)

The following Qs have to do with CALL-STACK-GROUP (see below)

SGSTAT The STACK-GROUP state. This has a field of the low six bits which may hold: (contents are given symbolically; SGNERR=0, SGNACT=1 etc.)

| CONTENTS | MEANING |
|---|---|
| SGNERR | Error (for the usual reason). |
| SGNACT | Active. |
| SGNINT | Interrupted. |
| SGNIND | Interrupted dirty. |
| SGNINC | Interrupted cleansed. |
| SGNAER | Awaiting error recovery. |
| SGNAED | Awaiting error recovery dirty. |
| SGNAEC | Awaiting error recovery cleansed. |
| SGNART | Awaiting return. |
| SGNACL | Awaiting call. |
| SGNAIC | Awaiting initial call. |
| SGNAGC | Awaiting garbage collection. |
| SGNEXH | Exhausted. |

It also has the following 1-bit fields (numbering from 2.3 down to 1.7)

| NAME | MEANING |
|---|---|
| SGSHLT | Halt if there is an attempt to "error out" of this stack group (' |
| SGSSVD | The special variables in this SG are swapped out (this is a non-runnable state) |
| SGSSWI | A special variable swap is in progress. This should not be on ul in the middle of the swapper. If some error occurs and the swap is not completed, this bit will be left on; the stack group is tl screwwed fatally. |
| SGSNSP | This SG binds no special variables at all, and it will therefore an error if it tries. |
| SGSSTO | Swap special variables on trap-out. |
| SGSSCO | Swap special variables on call out. |
| SGSSCI | Swap special variables of the SG which is previous to me when about to enter me. |

SG-PREVIOUS-STACK-GROUP Pointer to SG which called be or was interrupted "for" me. (so that I could be run)
SG-CALLING-ARGS-POINTER Pointer to argument-block which last called me.
SG-CALLING-ARG-NUMBER Number of args in above block.
SG-FOLLOWING-STACK-GROUP Pointer to SG I called or was interrupted to run.

SGAAS
SGAAJ
SGAAI
SGAAQ

These locations are used to save the states of some of tl locations in the CONS M-memory. (It is OK for these cells to have DTP-TRAP).

SGAAR
SGAAT
SGAAE
SGAAD
SGAAC
SGAAB
SGAAA
SGAAZR

SGSVMA Saved VMA register. Note that the MRD and MWD are NOT sa

This ends the elements of the array-leader of the ART-STACK-GROUP array. The leader of an ART-PDL-SEGMENT array are:

| NAME | MEANING |
| --- | --- |
| PSGPRV | Pointer to previous segment in chain. |
| PSGFOL | Pointer to following segment in chain. |
| FSGHDP | Pointer to the ART-STACK-GROUP array. |