Stack and Function Header Format

original: August 15, 1981 by Bill van Melle, Larry Masinter
reformatted for implementor’s manual: July 2, 1985 by Ron Fischer
revised: August 1, 1985 by Bill van Melle
revised: July 16, 1988 by Frank Shih (minor note re: Maiko)

Stack structure

The stack segment in the Interlisp-D virtual memory consists of a series of stack *blocks*, which represent frames for active invocations of Lisp functions and free space left over. There are actually four specific kinds of blocks that can appear in the stack space: basic frames (abbreviated BF), frame extensions (abbreviated FX), free blocks (FSB), and guard blocks (GUARD). Each BF holds the arguments of a particular function call, while the FX holds other locally-bound variables and temporary storage for the function. A BF and FX are created at each call or entry, and released upon return or exit. A FX and its associated BF will simply be referred to as a *frame*.

Basic frame

A Basic Frame consists of the n arguments to the function being called, possibly followed by a cell of padding, followed by the BF word containing flags and a pointer to the first argument. The BF word is quadword-aligned. Every frame extension is immediately preceeded by a BF, either the actual BF, or by a "dummy" BF pointer in which the R bit is on and the IVAR pointer is valid. Only "slow" FX’s can be preceded by a dummyBF.

```
IVAR:   *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |       0       |                  V a l u e 1                  |
        |      ...      |                      ...                      |
        |       0       |                  V a l u e n                  |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
BF:     |1 0 0 0 0 0|R|P|   Ext. cnt.   |           IVAR                |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The fields are interpreted as follows:

IVAR The 2nd word of the BF cell points back at the IVAR.

R "Residual". This bit is on in "dummy" basic frames which are actually only 2-word quantities. Normal function call leaves the bit off. Dummy BF’s are created when an FX has to move or be copied (on stack overflow or returning to a frame with non-zero use count). It’s not clear these are really necessary—one could move/copy the entire frame (BF+FX).

P "Padded". If 1, the actual number of arguments in the frame is 1 less than the size would indicate.

USECNT Number of frame extensions (less 1) pointing at this BF. Initially 0.

Frame extension

A Frame extension consists of 10 words of control information, followed by storage for locally bound variables (PVARs), then binding slots for any variable referenced freely by the function, followed by dynamic storage ("the stack").

```
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

```
FX:     |1 1 0|F|-|C|V|N|   Use count   |         Alink=oldPVar       |X|
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |       fn header lo            |  undefined    | fn header hi  |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |      next (= TOP+2)           |         PC                    |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |       (name table lo )        |  undefined    |(nametable hi) |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
        |            (Blink)            |            (Clink)            |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
PVAR:   |1|          undefined        |1|    undefined                  |
        |1|          ...              |1|        ...                    |
        |1|          undefined        |1|    undefined                  |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
TEMP:   |       0       |               Temporary values                |
        |      ...      |                      ...                      |
Top:    |       0       |               Last temp. value                |
        *-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The fields are interpreted as follows:

F "Fast". This frame binds no variables. This bit is (optionally) copied from the definition cell (see below).

C "InCall". Used by microcode to denote a frame interrupted in the middle of function call. Returning to a function with C on should invoke APPLYFN immediately instead of doing the return (the N bit is still valid).

V "NameTableValid". The "name table pointer" field is valid (see below).

N "NoPush". When returning to this frame, throw out the return value (used by punts, interrupts.)

The bits C and N need only be checked in a frame when performing a "slow return" to it. It is guaranteed that they will not be on in a frame which can be fast-returned to. The V bit is tested by free variable lookup.

ALink This field (with the low bit denoting "slow return") contains the ALink field of this frame, i.e., the pointer followed when chasing free variable references. The ALink points to the PVAR of the relevant frame, i.e., FX+10. The ALink field is also the CLink when the X (slow) bit is off. The CLink indicates the control chain, i.e., the frame to be returned to by this one.

X "SlowReturn". Must be set if the returnee is not contiguous with this this frame's BF, this frame's BF is not contiguous with its FX, or if there is a need to set the CLink independently of the ALink. Thus, if set, (a) the BLink and CLink fields of this FX are valid, and (b) when returning from this frame, the microcode for RETURN must do the "slow" case (usually punting, unless it's really a fast case after all).

Fn Header This is the pointer to the code base of this function (see Function Header format). Note that the high and low words of the function header are swapped.

next Points at the next stack word following this FX. Not valid while control is in this frame, of course—it is set when the frame is "closed out" by a function call or context switch of some sort.

PC Byte offset from FnHeader of next executable byte of this FX.

NameTable Where to find the table of names of arguments, prog variables, and free variable names. If the V bit in the flag word is off (normal case), this field is undefined, and the FnHeader field is used (the table of names is inside the function header); otherwise this field points at something that has the same format as a function header (see below).

BLink When X is set, points at the BF for this frame. When X is not set, BF=FX-2.

CLink When X is set, points at the returnee for this frame. When X is not set, CLink=ALink. CLink, like ALink, points at the PVAR area of the FX rather than at the beginning of the FX.

PVAR This area contains PROG/local variable values and free variable binding pointers. Initialized with all ones in the left half, which denotes {variable not bound} for PROG variables, and {variable not looked up} for free variables (see below). PVAR is quad-aligned (hence so are FX+1 and BF), and contains an even number of cells (so that TEMP is also quad-word aligned).

TEMP contains the temporaries (dynamic stack space) of the function. The first two cells of this region are garbage, for the benefit of the Dolphin hardware stack reloading. TEMP is quadword aligned, if necessary by padding out the PVAR area.

When a function is called, its arguments appear at the end of the caller's FX. The implementation is designed so that the bookkeeping for a BF appears at its high end, and for a FX at its low end, so that the arguments can be made into a new BF without having to copy them. After thus fabricating a new BF and shortening the old FX, the new FX is created. Upon return, the function value ("top of stack", which appears at the end of the returner's FX) is preserved while the returner's frame is deleted; then the value is pushed onto the caller's FX and execution resumes in the caller.

PVAR Slots and FVAR (Binding) Slots

When a variable is bound, via the BIND opcode, the corresponding slot in the PVAR area is filled in with the value:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|0|             |          value                                |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

When a free variable is found during free variable lookup, the variable's binding slot is filled in with:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|  binding pointer (even)     |0|  bind-addr-hi | bind-addr-hi  |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

This two word quantity is the (swapped) address of the location where the binding of the variable actually occurs. If the variable is not bound on the stack, this is the location of the variable's top level value, or possibly some other piece of storage. Note that all legitimate pointers are double-word aligned, and so have the low order bit turned off.

Note: on Maiko, the bind-addr-hi is *not* duplicated, i.e.:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|  binding pointer (even)     |0|       0       | bind-addr-hi  |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The only way to tell between the PVAR and the FVAR region is to look in the name table (or function header if V is off): (PV+1)*2 is the number of cells (doublewords) in the PVAR region; NLOCALS is the number of those which are PVAR slots; the rest are FVAR slots or padding.

FVAR_ (the free variable assignment opcode), tests whether the binding address of the variable is in stack space. If so, it can do the assignment directly. If not, it must do the assignment using RPLPTR, i.e., decrement the reference count of the old value and increment the reference count of the new. This can be done by punting to the ufn if desired: the arguments to the ufn are the new value and the address (in normal order) of the binding location.

Note that this architecture does not specify anything about where non-stack bindings can be. Ordinarily, if the binding pointer is not in stack space, it is the top-level value cell of the variable, but it can also be used for closures, etc. to allow data structure manipulation to look like variable reference. The only current instance of this use is that the binding of RESETVARSLST at the top level of each process is actually a pointer into the process handle, for the benefit of cleaning up after a HARDRESET.

Free blocks

The other two stack block types are used to manage free space on the stack and as markers for book-keeping purposes. They are:

Ordinary Free block

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|1 0 1 0 0 0 0 0|       0       |              Size             |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

Guard block

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|1 1 1 0 0 0 0 0|       0       |              Size             |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

The only difference between Free blocks and Guard blocks is that microcode is guaranteed not to "use" Guard blocks. They are used at the end of stack regions. Size is in words.

Stack blocks may be discriminated by selecting the leftmost three bits of the flags into:

000b	Ordinary pointer (not a flag word)
100b	Basic frame
101b	Free block
110b	Frame extension
111b	Guard block

Function format

1. Function definition cells

Function definition cells are either:

Interpreted:
```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|     0         |           ptr to definition (list/NIL)        |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

Compiled:
```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|1|F|Aty|   0   |           ptr to definition (array block)     |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

Compiled definitions have 1 in the high order bit. The `Aty` field contains the 'argtype' of the function, and is used only by the `CHECKAPPLY*` opcode. The `F` bit is set if the corresponding `NTSIZE` of the definition is zero, i.e., the function has an empty name table. Undefined functions appear as an 'Interpreted' function with definition = `NIL`.

2. Function header

A compiled function begins with a function header and a name table, which contains variable names; function names and other constants are stored in-line in the code. The first 8 words of the function are called the function header. The function header is quadword aligned. Format:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|               STK             |              NA               |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|               PV              |             START             |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|-|-|Aty|   0   |           function name                       |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|               NTSIZE          |    NLOCALS    |  FVAROFFSET   |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

`NA` The number of arguments this function expects. For `LAMBDA*` functions (arbitrary number of args, no adjustment) this field is –1.

`PV` Number of quadwords required for the `PVAR` region, less 1, i.e., (#PVars + #Fvars + 1)/2 – 1.

`STK` Amount of stack space required to call this function:

$$2*(\max(NA, 0) + (PV+1) * 2) + FrameOverhead + MinExtraStackWords.$$

max(NA, 0) is maximum number of extra (doubleword) pushes to fill in unsupplied arguments; FrameOverhead=12 (includes BF overhead); MinExtraStackWords is the number of words microcode requires before punting, currently 32 (for D0 HStack microcode).

`START` Initial PC of the function.

`Aty` Argtype of the function: 0 = `LAMBDA`, 2 = `LAMBDA*`, 1 = `NLAMBDA`, 3 = `NLAMBDA*`). ignored by microcode, but transcribed to the definition cell, where `CHECKAPPLY*` tests it.

NTSIZE Name table size (in words). This is a multiple of 4.

NLOCALS Number of PROG variables.

FVAROFFSET Offset (from start of header) of first FVAR in name table.

This is then followed by the specvar Name Table (not to be confused with the NAMETABLE field of the FX, which always points at something looking like a function header), which is actually two parallel tables.

The first table contains atom numbers ("value index"), terminated with enough zeros to fill out a quadword, but always at least one word of zero. Thus the size of the table in words is NTSIZE+1 rounded up to a multiple of 4. Note that in the special case where NTSIZE is 0, there is still a quad-word of zeros. The name table is arranged with PVAR names in reverse order of binding, IVAR names, and then FVAR names. Thus, when free variable lookup scans the name table in order, it will find the most recent bindings first.

The second table (which starts NTSIZE words beyond) contains entries of the form:

```
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
|vty|     0     |     offset    |
*-+-+-+-+-+-+-+-*-+-+-+-+-+-+-+-*
```

vty codes are as follows:

00	This name corresponds to an IVAR.
10	This name corresponds to a PVAR.
11	This name corresponds to a FVAR.

Offset is a zero-based doubleword offset from the start of the corresponding section of the BF or FX. Both PVAR and FVAR offsets are relative to PVAR, i.e., PVAR *n* is followed by FVAR *n*+1.

Free variable lookup scans the first table for a match of the "looked up variable". If a match is found, the word at the same offset in the second table is fetched to determine whether the value is in the IVAR section (vty=0), in the PVAR section (vty=2) or pointed to by the FVAR section (vty=3).

The regular name table is followed by the "localvar argument" name table, which is not visible to the microcode. It is in the same format as the regular name table, but lists names of arguments to the function that are declared LOCALVARS and hence would otherwise be invisible. This table is for the benefit of ARGLIST and the debugger.