SECTION 11

Function-Calling

11.1 INTRODUCTION

This section explains the functional objects in the Explorer system and how they are called. It is not necessary to understand the earlier virtual machine (VM1) to understand the mechanisms of this virtual machine (VM2), though it may prove helpful in understanding our motivations. The following sections describe the various kinds of functional objects, VM1 function-calling and the motivations behind the VM2 design, the layout of stack-frames, the structure of the compiled-function object, and the actual function-calling mechanisms.

11.2 FUNCTIONAL OBJECTS

Objects of almost every user-visible data-type in the Explorer system can be called as functions (the exceptions being numbers and characters); they may be divided into four categories:

1. Compiled functions: the objects which are produced by the Lisp COMPILE function or by loading the file generated by COMPILE-FILE.

2. Interpreted functions: objects which are represented in list structure.

3. Indirect functions: objects that may be applied to arguments, but which when applied find an internal functional object and apply it instead.

4. Non-functions: objects that may be applied to arguments, but which do some action specific to the data type.

11.2.1 Compiled Functions.

Compiled functions are of data-type DTP-FUNCTION, and are directly interpreted by the microcode. Compiled functions are the most important part of the function-calling code. Most of this chapter deals with calling compiled functions. (The section on FEF LAYOUT describes the layout of a compiled function object.)

11.2.2 Interpreted Functions.

Interpreted functions are produced by DEFUN or LAMBDA, and are passed on to the Lisp interpreter. Objects of data-type DTP-LIST, DTP-STACK-LIST, or DTP-LOCATIVE are treated as interpreted functions. When the microcode encounters an interpreted function, it collects a list of the arguments and passes the argument list and the function to SYS:APPLY-LAMBDA, the microcode's interface to the Lisp interpreter, which it finds in the support vector. (See the section on Support-Vector for more details).

11.2.3 Indirect Functions.

There are several different kinds of indirect functions: symbols, closures, instances, named-structures, and funcallable hash-arrays. They are discussed below in order of complexity.

11.2.3.1 Symbols.

The simplest of the indirect functions is the symbol (DTP-SYMBOL). When a symbol is called, the microcode calls the contents of its function cell instead. Most compiled code does not actually call symbols; rather than a pointer to the symbol, the function will contain an invisible pointer (DTP-EXTERNAL-VALUE-CELL-POINTER) to the symbol's function cell. It is this indirection mechanism that allows the developer to recompile a function without having to recompile its callers.

11.2.3.2 Closures.

Closures are also quite simple: a closure is a pair, one part of which is a functional object to be called. The other part of the pair is a list of pointers to value cells and values to bind them to, if the closure is a dynamic closure (DTP-CLOSURE), or an environment structure, if the closure is a lexical closure (DTP-LEXICAL-CLOSURE). When a dynamic closure is called, the microcode performs all the special-bindings, then calls the closure's functional object. When a lexical closure is called, the microcode calls the closure's functional object and places

the environment structure in a special local variable in the call-frame. (See the section on Closures for more detail). In both cases, a previous environment is restored for the function to execute within, though the type of the environment is quite different.

11.2.3.3 Instances, Named-Structures, and Funcallable Hash-Arrays.

Instances, named-structures, and funcallable hash-arrays are closely related. A named-structure is an array (DTP-ARRAY) with its named-structure-flag (see the sub-section on Arrays) turned on. A funcallable hash-array is a named-structure with its funcall-as-hash-array flag turned on. An instance (DTP-INSTANCE) contains a pointer to its flavor's method table which is a funcallable hash-array. (See the sub-section on Calling an Instance).

A call to a simple named-structure is turned into a call to SYS:NAMED-STRUCTURE-INVOKE found through the support vector, (See the sub-section on Support Vector), with the structure as the first argument and all the other arguments following. SYS:NAMED-STRUCTURE-INVOKE looks for the SYS:NAMED-STRUCTURE-INVOKE or :NAMED-STRUCTURE-INVOKE property on the type-symbol of the structure, and if found calls it with the arguments and (if the named-structure-invoke property is a closure) the structure itself.

Calling a funcallable hash-array causes a hash-table lookup of the first argument in the specified hash-array. If a functional object is found, it is called, otherwise SYS:INSTANCE-HASH-FAILURE (another function found through the support vector) is called. Calling an instance binds SELF, then binds any special instance variables, then picks up the funcallable hash-array that is its flavor's method table and calls it as described above. (More detail on the structure of flavors and the calling of instances may be found in the sub-section on Calling an Instance).

11.2.4 Non-Functions.

There are three kinds of non-functions that can be called as functional objects: arrays, stack-groups, and microcode-entry functions.

11.2.4.1 Arrays.

Calling an array (DTP-ARRAY), if that array is not a named-structure, is equivalent to doing AREF with the array as the first argument and the other arguments as the indices. In fact, that is how it is implemented. Calling a named-structure is described in the section on Indirect Functions.

11.2.4.2 Stack-Groups.

Calling a stack-group (DTP-STACK-GROUP) uses the function SYS:CALL-STACK-GROUP from the support vector to resume that stack-group. (See the sub-section on Support Vector.)

NOTE

> Stack-groups are described from the user's point of view in the Explorer Lisp Reference manual and internally in the section on Stack Groups.

Multiple arguments and values are not yet supported, but all the hooks are in place. Currently, SYS:CALL-STACK-GROUP is passed all the arguments and the stack-group itself, with the stack-group last, and returns one result.

11.2.4.3 Microcode-Entry Functions.

A microcode-entry function (DTP-U-ENTRY) is exactly what its name states, a function that is a direct entry into the microcode, which is interpreted by the hardware. There are very few microcode-entry functions (about ten); most of the time, entry to the microcode is accomplished by macroinstructions. (See the section on Macroinstructions). However, the standard macroinstruction mechanisms do not allow for operations that accept an indefinite number of arguments (a &REST arg). Microcode-entry functions overcome this limitation by using the standard function-calling mechanism to determine the number of arguments and acting accordingly; microcode-entry calling will be discussed in more detail in the section on U-Entries.

NOTE

> Interpreted versions of misc-ops and aux-ops (See the sub-section on misc-ops) are generated at build time. They are simply compiled functions that invoke the appropriate instruction. This is a difference from VM1, in which all interpreted versions of misc-ops were microcode-entry functions.

11.2.5 Obsolete Functional Objects.

There are several functional-object data-types in VM1 that have been dropped in VM2:

* DTP-STACK-CLOSURE has been replaced by DTP-LEXICAL-CLOSURE. Both implement lexical closures (Section on Lexical-Closure), but the newer implementation is a complete redesign that corrects many flaws and offers some new features and improved performance.

* DTP-ENTITY and DTP-SELECT-METHOD have been eliminated. Entities were a variant of dynamic closures that bound SELF as well as their own bindings. Select-methods were essentially a list of operations and functions. When called, the first argument was taken as the operation to look up the function. An entity with a select-method as its functional part was the forerunner of the instance in the class system that predated the flavor system. The only holdover in VM2 is the Lisp macro DEFSELECT, which will now expand into a different structure with the same functionality.

11.3 HISTORY AND MOTIVATIONS

VM2 function-calling differs considerably from its VM1 predecessor. VM1 function-calling used what we have always called an "upside-down" stack layout: a call-block would be "opened" by pushing some state words and the function, then the arguments would be pushed; the last argument would "activate" the call-block, actually starting the function call. Multiple-value calls and other special calls would push additional information words ("ADI" words) before opening the call-block. VM2 function-calling, in contrast, uses a "right-side-up" stack layout, pushing all arguments before doing any sort of call. There is no such thing as an open call-block. Also, there are no ADI words; all information supplied at call time is kept in one word, and other information is kept in a fixed number of registers.

The compiled-function object (called a FEF for historical reasons) is changed as well. The VM1 FEF carried a list, the argument-descriptor list (ADL), that described the type, position, and initialization of every argument and local. Later work added the "fast-arg-option" word, a condensed form of the ADL for simpler cases. Release 1 of the Explorer system modified the header word of the FEF to contain even more condensed information for even simpler cases. The VM1 FEF also contained information about any special arguments, which would be bound by the function-calling microcode. In VM2, the ADL and special-binding information have been eliminated. The microcode initializes all locals and optional arguments to NIL, and the

compiler generates code for non-NIL initializations and for special-bindings. All information about the number of arguments and locals is present either in the header word or, if there are too many arguments or locals for the counts to fit there, in an optional word called the long-args word. Unlike VM1, the long-args word is not always present. Also, the name of the function, which in VM1 was kept in the FEF itself, has been moved to the FEF's debugging-info (see the section on FEF LAYOUT).

The guiding principles of VM2 function-calling have been simplicity and speed. VM1 function-calling is horrendously complex because it developed from a design intended to be perfectly general, but which had all sorts of speed hacks added on later. All this complexity makes it extremely slow in the slow cases and not very fast in the fast cases. The open/active call-block idea has also lost its appeal: most of the current function's state is kept on the stack where it requires an extra cycle to access, probably for historical debugging reasons, and the necessity of checking for open call-blocks adds overhead to most stack operations. Our guidelines in designing VM2 function-calling were:

* Keep all the state of the current running function in registers.

* Invert the stack: replace the open-call-block/active-call-block distinction with the simpler scheme of pushing all the arguments first, then doing the call.

* Do not copy the arguments (this guideline had an important influence on the timing of operations in the microcode).

* Let the compiler do all the hard work: non-NIL initializations of optional variables, special-bindings.

* Simplify the FEF: for the common cases, keep the information needed by the microcode in the header word; for the less-common cases, use one other word.

* Simplify the state: most of the important call and return state is kept in one word.

The remaining sections will describe VM2 function-calling in detail.

11.4 STACK LAYOUT

A snapshot of the PDL is shown in Figure 11-1. There are several call-frames shown. C is the current running function and its frame is shown in most detail. C was called by B, which was

itself called by A. Note how the state for B is between C's arguments and locals. C's state is not shown, because it is in the five registers M-CALL-INFO, M-ARGUMENT-POINTER, M-LOCAL-POINTER, M-FEF, and LOCATION-COUNTER. The values of M-ARGUMENT-POINTER and M-LOCAL-POINTER are indicated in Figure 11-1 by the two arrows marked "argument-pointer" and "local-pointer." LOCATION-COUNTER is described in the section on Macroinstructions; note that when the registers are pushed on the stack, its offset from the beginning of the function is saved instead.

A function is called by pushing its arguments and executing a CALL instruction. The CALL instruction pushes NILs for any unsupplied optional arguments, pushes the state of the calling function, pushes NILs for any locals of the called function, sets up the state registers, then causes macroinstruction execution to continue at the first instruction of the new function.

There are a few areas of added complexity:

* If there are any optional arguments, the CALL instruction leaves the count of the supplied optionals on top of the stack when it finishes. The intent is that the function's first macroinstruction will use this count to initialize the unsupplied optionals (this macroinstruction is frequently a DISPATCH instruction). The function is free to ignore this count, since it will not interfere with anything; it is likely to be ignored if all optional arguments default to NIL.

* If the function is lexpr-funcalled (applied, in Common Lisp terms), the last argument is a list that is spread into its individual elements at call time. In VM1, the list was spread unconditionally by a (MISC) %SPREAD D-LAST, which spread the list and activated the call-block at the same time. In VM2, we try to be more efficient by spreading the list during the CALL instruction: If the function has a &REST argument and enough arguments are supplied that part or all of the lexpr-list would be collected in that &REST argument, we do not spread that part of the list, storing it directly into the &REST argument instead.

* If the object being called is a lexical closure, the closure's environment object is temporarily pushed on the stack between the time the closure is taken apart and the time the environment is stored into its reserved local (see the section on Closures).

* If the object is being called with a self-mapping-table (see the section on Instance-Calling) supplied, the self-mapping-table is supplied on the stack above the arguments and is picked up before any optional-argument

processing.

* If the object is being called with a macroinstruction call-destination of D-TAIL-REC, it means that this call is destructively tail-recursive.

NOTE

Non-destructive tail recursion uses the destination D-RETURN, which has little special effect at call time, though much at return time.

In destructive tail-recursion, the callee's arguments are copied down over the arguments of the caller, obliterating the caller's frame, and the callee's state is modified so that it returns directly to its caller's caller. This kind of tail recursion is known colloquially as "frame-eating" tail recursion, and is enabled by setting compiler optimization levels to emphasize size and speed and deemphasize safety.

NOTE

In contrast, VM1 destructive tail-recursion was decided at runtime, controlled by a user-settable variable and some instructions that allowed or prohibited tail-recursion when it might not otherwise appear possible. It did not work very well, and added the overhead of checking to every return.

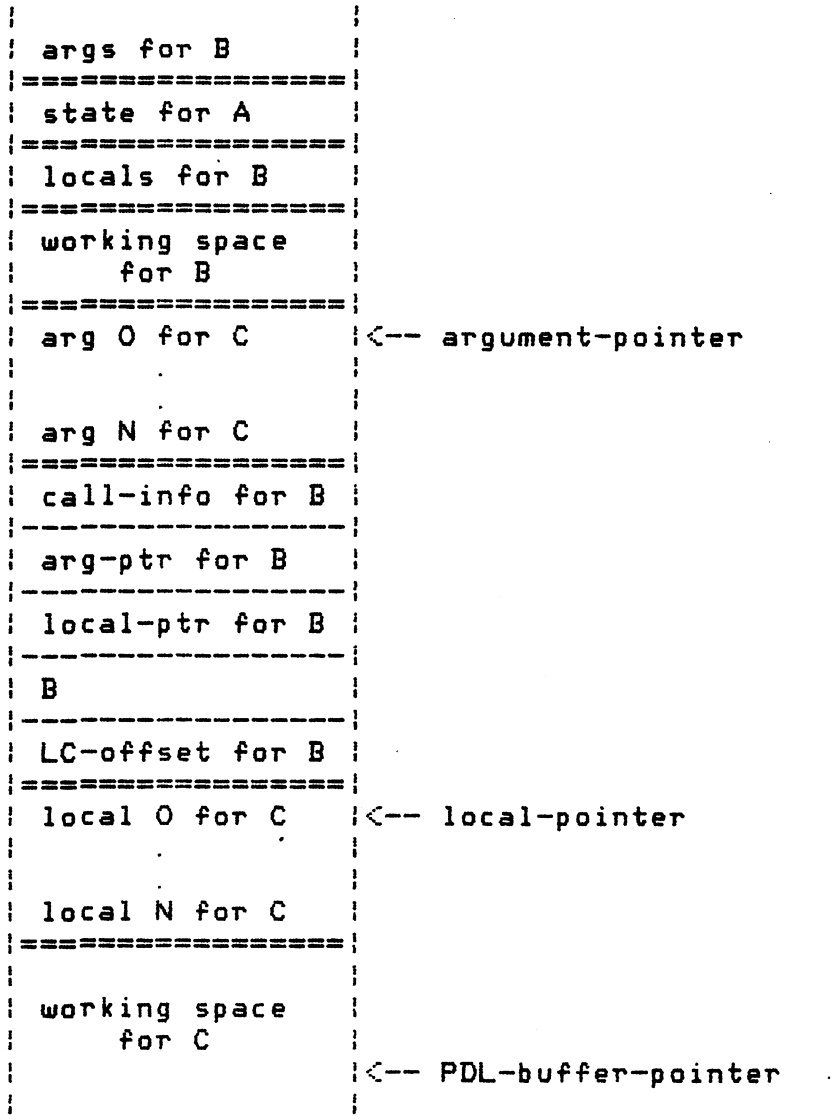


Figure 11-1 Call-Frame Layout

11.5 THE CALL-INFO WORD

Following the design guidelines shown in the section on motivations, most of the call and return information is supplied and kept in one word, the call-info word, a fixnum whose pointer field is laid out as pictured in Figure 11-2. In the generic CALL, the call-info word is pushed on the stack after the arguments, though there are several short-form call instructions that use a call-info word derived from the instruction. There

are three types of fields in the call-info word: call fields, return fields, and state fields.

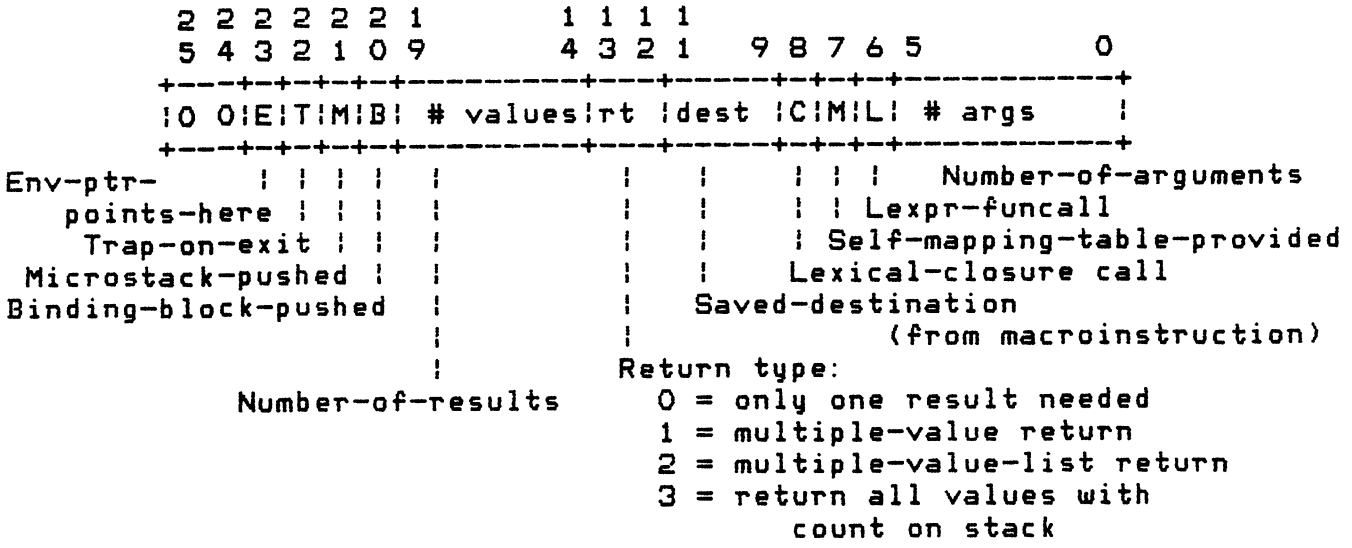


Figure 11-2 Call-Info Word Layout

11.5.1 Call Fields.

Call fields are those containing information about the kind of call. These are usually supplied by the compiler when it generates a call-info word, or by the short-form call instructions in their constant call-info words. Call fields are all in the low nine bits of the call-info word, so most call variations can use a call-info word generated by a PUSH-NUMBER instruction, as long as multiple values are not needed. The call fields are:

* The number-of-arguments field, %%call-info-number-of-arguments, which allows for up to 63 arguments (though more actual arguments can be supplied using lexpr-funcall).

* The lexpr-funcall flag, %%call-info-lexpr-funcall-flag, set when doing a LEXPR-FUNCALL or APPLY.

* The self-mapping-table-provided flag, %%call-info-self-map-table-provided, set when supplying the self-mapping-table on the stack.

* The lexical-closure-call flag, %%call-info-lexical-closure-call, which, strictly speaking, is not a call field, because it is set by the microcode at call time.

This flag indicates that a lexical closure is being called and that its environment object is on the stack.

11.5.2 Return Fields.

Return fields contain information about the kind of return that the caller expects from the call. These fields are supplied at call time, but are copied from the caller's call-info word if the macroinstruction destination is D-RETURN or D-TAIL-REC. There are two fields:

* The number-of-results field, %%call-info-number-of-results, which allows for up to 63 values.

* The return-type field, %%call-info-return-type, which indicates the form of returned values:

  - %only-one-result-needed (0): the caller only wants one result. This is special-cased for speed.

  - %normal-return (1): 0 to 63 values, as a block. When the function returns, there will be this many values left on the stack for the caller (or ultimate caller). Excess returned values will be trimmed and missing values will be filled in with NILs.

  - %multiple-value-list-return (2): all the returned values will be collected into a list, which will be returned as a single value.

  - %return-all-values-with-count-on-stack (3): the number-of-results field will be ignored and all values will be returned. The number of values will be left on the stack. This form of return is needed for situations involving indefinite numbers of values, usually associated with throws and unwind-protects, where, due to tail-recursion, it is not possible to determine how many values to return at call time. Calls with this return-type are often followed by RETURN-N instructions.

11.5.3 State Fields.

State fields are fields used by the microcode to keep track of the state of the frame, usually to preserve information from the call or from the execution of the function until return time, when it will be needed. The state fields are:

* The saved destination, %%call-info-saved-destination:

the destination field from the macroinstruction is copied here; the field is in the same position as it is in the macroinstruction so it can be copied in one microinstruction with selective-deposit. The field is three bits wide instead of two because the microcode recognizes a fifth destination not available from macrocode: D-MICRO. This destination is used when "calling out" from microcode to macrocode (See the section on Calling "Out").

* The "marked-frame" flags, including:

  - Environment-pointer-points-here, %%call-info-env-ptr-points-here: set when a lexical-closure object is created that uses this frame in its environment (See the section on Closures); indicates that this frame must be "unshared."

  - Trap-on-exit, %%call-info-trap-on-exit: manipulated by the debugger commands C-X, M-X and C-M-X and the function breakon; causes an exit-trap when returning from this frame.

  - Microstack-pushed, %%call-info-microstack-pushed: set when the microstack is saved on the special PDL; usually set by calling out to macrocode, so that returning from the macrocode function will restore the microcode's state.

  - Binding-block-pushed, %%call-info-binding-block-pushed: set when a special-binding is done; indicates that there is a block of bindings to be undone when returning.

11.6 FEF LAYOUT

The compiled-function object, the FEF, is shown in Figure 11-3. Every FEF contains both boxed and unboxed words. The boxed words are collectively called the "FEF header" and contain information about the function and any constants that it needs. The unboxed words contain the macroinstructions.

The words in the FEF header are:

1. Word 0 has type DTP-FEF-HEADER and contains the following fields:

   a. Special form flag (%%FEF-HEADER-Special-Form, 1 bit) --- indicates whether there are any &QUOTE or &FUNCTIONAL arguments.

b. SUBST flag (%%FEF-HEADER-Subst, 1 bit) --- set if the function was defined by DEFSUBST.

c. Self-mapping table required (%%FEF-HEADER-Self-Mapping-Table, 1 bit) --- when set, a self-mapping-table must either be provided or fetched during the call.

d. Call type (%%FEF-HEADER-Call-Type, 3 bits):

...%fef-call-simple (0) = simple call (no optionals or locals)

...%fef-call-optionals (1) = optional arguments

...%fef-call-lo als (2) = local variables

...%fef-call-rest (3) = &REST arg (with or without other locals)

...%fef-call-optionals-and-locals (4) = optional args and local variables

...%fef-call-optionals-and-rest (5) = optional and &REST args

...%fef-call-unused (6) = [unused]

...%fef-call-long (7) = use the longs-args word.

e. Number of optional arguments (%%FEF-HEADER-Number-Optional-Args, 3 bits).

f. Number of required arguments (%%FEF-HEADER-Number-Args, 4 bits).

g. Number of local variable slots (%%FEF-HEADER-Number-Locals, 4 bits).

h. Offset of the word containing the first instruction (%%FEF-HEADER-Location-Counter-Offset, 10 bits).

2. Word 1 is a fixnum specifying the length of the FEF in words (used by the storage management code, see the section on Storage Management).

3. Word 2 points to the debugging info, which is a structure of type SYS:DEBUG-INFO-STRUCT. This structure has five slots: the name of the function, its true arglist, its interpreted definition if there

is one (often useful for functions which are declared INLINE), its local-map (which describes the layout of the local variables in the frame), and a plist which contains other fields, such as :MACROS-EXPANDED, :DOCUMENTATION, and :VALUES.

4. The long-args word is optional. It is needed when the counts will not fit in word 0, and its presence is indicated by the Call Type (%FEF-CALL-LONG). It contains:

   a. Optional arguments flag (%%fef-long-args-optionals, 1 bit) --- set if there are optionals.

   b. Local variables flag (%%fef-long-args-locals, 1 bit) --- set if there any locals.

   c. &REST arg flag (%%fef-long-args-rest-arg, 1 bit) --- set if there is a &REST argument.

   d. Minimum number of arguments (%%fef-long-args-min-args, 6 bits) --- the number of required arguments.

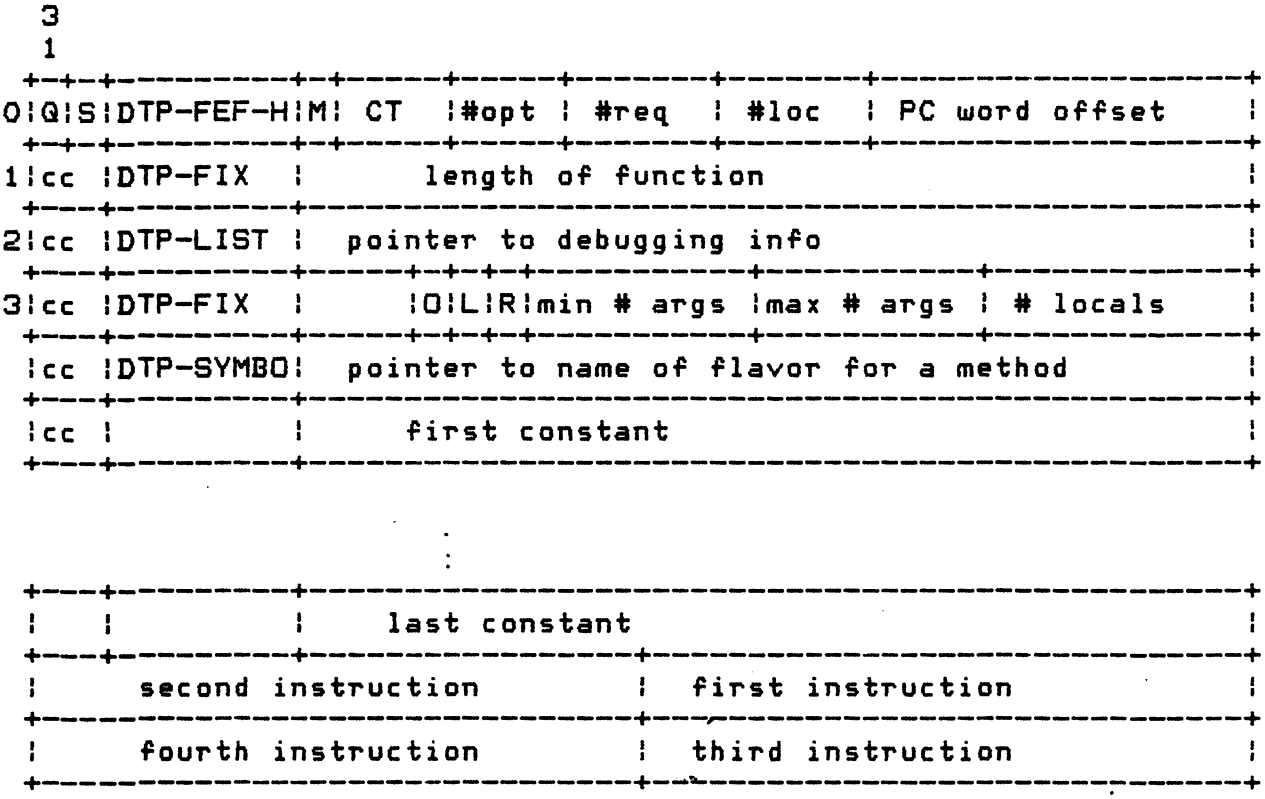
   e. Maximum number of arguments (%%fef-long-args-max-args, 6 bits) --- the sum of the number of required arguments and the number of optional arguments. The &REST argument, if present, is not counted here.

   f. Number of local variable slots (%%fef-long-args-number-of-locals, 7 bits).

      The maximum/minimum representation of the arguments (instead of the optional/required form in word 0) is a historical holdover, as is the seven-bit locals field (only 6 bits worth are accessible, but the old stack-closure code used to allocate some of its storage using this field).

5. The flavor name is also optional, and is present when the self-mapping table bit is set in word 0. If the flavor-name word is present, but the long-args word is not, the flavor-name word will be word 3; when both are present, it will be word 4.

The remaining header words are constants used in the function, indicated in disassembled code by FEF!n, where n is the offset from the beginning of the FEF. Many of these constants will be external-value-cell pointers to value-cells and function-cells of symbols.



Q - Special Form Flag
S - SUBST Flag
M - Self Mapping Table Required
CT - Call Type
O - Optional Arguments Flag
L - Local Variables Flag
R - &REST Arg Flag

Figure 11-3 The Layout of a FEF

11.7 CALLING A FEF

11.7.1 The Instructions.

There are several kinds of CALL macroinstructions, depending on the grouping chosen. They may be divided into mainops and auxops (see the section on Macroinstructions) or general and special-case, and there are shadings in each division. The instructions

are:

CALL-0 through CALL-6:
These seven instructions all call a function with the number of arguments specified in the instruction, to return one result, nothing fancy. They all take the standard "register" and offset for the function, but are coded to favor FEF-relative arguments that are external-value-cell pointers.

CALL-N:
Similar to CALL-0 and friends, but takes the number of arguments from the top of the stack.

(AUX) COMPLEX-CALL:
The most general of the call instructions, COMPLEX-CALL takes both the function and a call-info word on the stack. This instruction actually exists in four forms to express the four call destinations: COMPLEX-CALL-TO-INDS, COMPLEX-CALL-TO-PUSH, COMPLEX-CALL-TO-RETURN, and COMPLEX-CALL-TO-TAIL-REC.

(AUX) APPLY:
This special-case instruction takes the function from the stack and lexpr-funcalls it with one argument, to return one result. It exists in four forms just like COMPLEX-CALL.

(AUX) LEXPR-FUNCALL-WITH-MAPPING-TABLE:
This special-case instruction takes the function and a self-mapping-table from the stack, and lexpr-funcalls the function with one argument and self-mapping-table provided, to return one result. It exists in four forms just like COMPLEX-CALL.

Other instructions may be added later.

11.7.2 The Actions.

The pattern of actions taken by the various call instructions resembles a tree: there are several beginnings, all of which merge at various points into a single main path, which itself branches out into several subroutines. All the auxop calls follow the same path after five or six microinstructions to pick up the function and figure out the call-info word. The mainop calls have the FEF-relative path broken out separately, but merge quickly. They do not handle the complex situations that the auxop calls do. The auxop calls merge with the mainop calls before dealing with the arguments and locals. With all that merging and separating in mind, here is the generalized sequence of operations of the call macroinstruction:

1. Find the function, the new argument-pointer, the new

call-info word, and the number of supplied arguments; all must be set up before dispatching on the data-type of the function. If the self-mapping-table-provided bit is set in the call-info word, pick up the new self-mapping-table.

2. Dispatch on the data-type of the function to the appropriate handler. At this point, the function is guaranteed to be in MD, the new argument-pointer in M-G, the new call-info word in M-F, and the number of supplied arguments in M-H. Most functional objects will eventually return here, having led to a compiled-function (DTP-FUNCTION) object; this dispatch pushes its own address, and falls through on DTP-FUNCTIONs. The other routines that extract functions from objects must not disturb most registers.

3. Calculate the location-counter offset from the previous function. Since the location-counter always points at the next instruction, this offset will be used to continue the previous function when this one returns. Read the header word of the FEF. Check for PDL-buffer overflow. Check for lexical-closure calls, if the bit is set in the call-info word, pick up the environment object from the stack and save it for later. If metering is turned on, record this function entry. Check for lexpr-funcalls, and spread the last argument if necessary; if part of the lexpr-list matches up with a &REST argument in the function, do not spread it only to collect it again, but make the list the &REST argument directly. If the trap-on-calls bit is set, trap now. note If the call-type of the function is %FEF-CALL-LONG, we do not try this nicety, because determining the existence of a &REST argument requires another memory reference that is not convenient at this point.

4. Handle the call-destination: If the call-destination is D-INDS (D-IGNORE) or D-PDL (D-STACK), do nothing. If the call-destination is D-RETURN, copy the return fields (see the section on Return-Fields) from the previous function's call-info word. If the call-destination is D-TAIL-REC (D-REALLY-TAIL-RECURSIVE), copy the return fields like D-RETURN, copy the call-destination and all the state from the previous function, and copy the arguments down over the previous function's frame, so that a return from this function will return to the previous function's caller instead of the previous function itself. Note that the call-destination is a two-bit field that is in two different places in the instruction, one for the mainops and one

for the auxops.

5. Take care of the arguments, checking for too few or too many, pushing NILs for any unsupplied optional arguments, and setting up the stack-list for any &REST argument (all arguments are pushed with cdr-code CDR-NEXT, so &REST-argument handling also must change the last one to CDR-NIL (see the section on Cdr-Codes)).

6. Advance the instruction stream. Call and return instructions do not use the mechanisms of the main macroinstruction loop, but instead include those mechanisms in the code, so that the next step can be hidden under the memory reference here. Note that only page-faults and interrupts may be checked for here, not sequence-breaks, because of the incomplete state of the call frame.

7. Push the five state words of the previous function. The old argument-pointer, local-pointer, and location-counter offset are all pushed as fixnums; the previous function and its call-info word are pushed as is, except that the call-info word is pushed with cdr-code CDR-ERROR to make call-frame tracing easier.

8. Take care of the locals, which involves pushing NILs for them, storing any lexical-closure environment object into its reserved local, placing any &REST argument in local 0, setting up the new local-pointer, and leaving the number of supplied optionals, calculated in step argument-handling, atop the stack. If there are non-NIL defaults, the first instruction of the function will probably be a dispatch-type instruction that will use that number to decide which optionals to initialize.

9. Pick up the first macroinstruction word and decode it, starting execution of this function.

11.8 CALLING ANYTHING ELSE

11.8.1 Calling the Interpreter.

Whenever a CALL instruction encounters a list, stack-list, or locative, it treats it as an interpreted function. The microcode builds a faked call-frame for the list-call, then collects a list of all the arguments and passes the function list and the argument list as the two arguments to the function SYS:APPLY-

LAMBDA, which it finds through the support vector (see the section on Support-Vector). It calls SYS:APPLY-LAMBDA with a destination of D-RETURN, so the result of the call will also be the result of the list-call. The only unusual details involve the possible need to diddle the cdr-codes of the arguments on the stack to make a stack-list of the arguments.

11.8.2 Calling an Instance.

An instance (DTP-INSTANCE) is a pointer to a word whose data type is DTP-INSTANCE-HEADER. The instance itself is the words following the header, while the header word points to the instance descriptor. All instances of the same flavor have headers pointing to the same instance descriptor, which contains the information common to all those instances.

Calling an instance first binds the variable SELF to the transported instance then looks in the instance descriptor for any other special-bindings to do, then looks in the instance descriptor for the method's function. If the function is not an array, it simply calls it (in case of a non-hash-table implementation).

NOTE

We may eventually make SELF and SYS:SELF-MAPPING-TABLE into lexical variables.

If it is an array, it is treated as a hash-array, and the first argument is the key. The modulus of the hash-array must be a power of two for the simple hashing algorithm used, which simply looks at the low bits of the symbol that is the key. Each entry has three words: key, locative to function, and mapping-table. If the key does not match, the microcode looks at the next entry, etc, until it either finds a match or an entry whose key is unbound (DTP-NULL) (this is different from the usual hashing algorithm). When it finds a match, it binds SYS:SELF-MAPPING-TABLE if the mapping-table slot is non-NIL, then dispatches on the data type of the new function.

If no matching entry is found, or if the key or the hash-array turns out to be gc-forwarded, the microcode calls out to SYS:INSTANCE-HASH-FAILURE, which will rehash the hash-array if necessary, or signal an error if the key is truly not in the table.

11.8.3 Calling a Microcode-Entry Function.

The pointer field of a DTP-U-ENTRY function is an index into the MICRO-CODE-ENTRY-AREA, a table of microcode entries. Entries in the table are either fixnums, indicating true microcode entries; NIL, indicating invalid entries; or other objects, which are the definitions for microcode entries that are not presently microcoded. The fixnums that indicate true microcode entries are themselves indices into the MICRO-CODE-LINK-AREA, a table of information about the microcode routine indicated by the microcode entry. The MICRO-CODE-LINK-AREA is built by the microassembler and lives in the microcode band.

The microcode builds a faked call-frame on the stack, then checks the arguments using two fields in the link-area data:

* The %%microcode-entry-args field indicates how many required arguments the microcode entry takes. This field is exactly equivalent to the number-of-requireds fields in the FEF header and long-args word.

* The %%microcode-entry-rest bit is 1 if the microcode-entry function takes a simulated &rest argument. If so, any number of arguments may be supplied; if not, the maximum number of arguments is the number in %%microcode-entry-args.

One complication arises from the fact that microcode-entry functions are under no compulsion to obey the normal conventions for dealing with arguments. In particular, they are likely to pop the arguments off the stack during their actions. To cope with this behavior while preserving a call-frame in case of error, the arguments are copied to the top of the stack before calling the microcode routine.

The final step in calling a microcode-entry function is to jump to the beginning of the microcode routine. The starting address is in the %%microcode-entry-index field in the link-area data. Before the jump itself, the microcode sets up return addresses so that the end of the microcode routine causes a function return. At the moment, microcode-entry functions may only return one value (while some miscops do return multiple values, they are not microcode-entry functions).

11.8.4 Calling "Out".

"Calling out" is the term used for any function call that is originated from the microcode. The microcode calls "out" from the microcode level to the macroinstruction level, hence the name. The main difference between calling out and normal calling is that since calling out is done from microcode, the microcode state must be preserved. The most important part of the microcode is the micro-pc return stack (microstack or UPCS, for short), which is saved as a block on the special PDL during the

call. The %%call-info-microstack-pushed bit in the call-info word is used to keep track of which calls saved the microstack.

The call destination D-MICRO exists for the use of calling out. Calls with this destination do not read a macroinstruction when they return; they simply do a microcode return to the routine that called out, and continue executing microinstructions. When the microcode routine finishes, only then is the next macroinstruction fetched and executed. This destination makes it possible for the microcode to call out to Lisp, receive results, and continue, possibly indefinitely. Calling out can also use the other call destinations, though D-RETURN is the most common.

The most common reason to call out is that the calculation is too complicated to do in microcode; for example, arithmetic on rational and complex numbers is handled by the two functions SYS:NUMERIC-ONE-ARGUMENT and SYS:NUMERIC-TWO-ARGUMENTS, which the arithmetic microcode calls out to. Another example is EQUAL and EQUALP, which are recursive in microcode. Since the microstack is only 64 words, there is a real danger of overflowing it when comparing deep structures. EQUAL and EQUALP solve this problem by calling out to themselves, thus saving the old microstack and obtaining more room to work. The function-calling microcode also occasionally calls out. When calling an interpreted function, it calls out to SYS:APPLY-LAMBDA.

There are other ways that the microcode uses calling out. The most unusual is that at boot time, the initial function is called out to by the boot microcode. This way, there is always a valid frame at the bottom of the initial stack-group's stack, and if this function is returned from (though unlikely, it can be forced from the debugger), the microcode will loop back and call it again. It is thus also unnecessary to make a special check for the bottom of the stack when returning.

11.8.5 The Support Vector.

Calling out is inextricably linked with the support vector. The support vector is an area of memory known by the microcode that contains some symbols needed by the microcode. Table 11-1 lists the contents of the support vector. Many of these symbols are functions that are called out to.

SYS:NAMED-STRUCTURE-INVOKE is used as described in the section on Array-Call when calling a named-structure as a function. SYS:APPLY-LAMBDA is used when calling interpreted functions (see the section on Interpreted Functions). SYS:EXPT-HARD, SYS:NUMERIC-ONE-ARGUMENT, SYS:NUMERIC-TWO-ARGUMENTS, SYS:LDB-HARD, and SYS:DPB-HARD are used for the difficult cases of some arithmetic functions and for byte fields too large for the microcode LDB and DPB. EQUAL, EQUALP, and EQUALP-ARRAY are used when the microcode recursion becomes too deep, as described in

the section on Equal-Call-Out. SYS:DEFSTRUCT-DESCRIPTION is used when doing TYPEP of named-structures; it is the property of the named-structure-symbol that contains all the internal information about the structure. SYS:INSTANCE-HASH-FAILURE is the function that calling an instance will call out to if the key is not found or if the hash-table needs rehashing (see the section on Instance-Calling). PRINT, *PACKAGE*, and the unbound marker are currently unused.

SYS:INSTANCE-INVOKE-VECTOR is an array of keyword symbols that represent messages to send to an instance when trying to do several of the basic Lisp operations to that instance. For example, trying to take the CAR of an instance will cause the :CAR message to be sent to that instance. The contents of the instance-invoke vector are :GET, :GETL, :GET-LOCATION-OR-NIL, :CAR, :CDR, :SET-CAR, and :SET-CDR. While :GET and its cousins are relatively useful in that they make GET a more generic function, the others seem to be remnants of an attempt to do everything using message-passing, since almost no flavor accepts the messages.

Table 11-1 Contents of the Support Vector

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

11.9 RETURNING

11.9.1 Basics.

Returning is governed by the return instruction itself, and by three fields in the call-info word,

* Number-of-results field
* Return-type field
* Saved-destination field

The return instruction indicates how many values are being returned, the number-of-results field indicates how many are wanted, the return-type field indicates how the values are to be returned, and the saved-destination field indicates where the values are to be returned. The return-type field and the saved-destination field are adjacent in the call-info word so the return microcode can dispatch on the combination of the two and decide quickly what is to be done. There are five call destinations, four of them accessible from compiled code. D-PDL is the "normal" destination; the results of the call will be left on the stack, first one farthest from the top. D-INDS (called D-IGNORE in VM1) means that no values are to be received; the function was either called for effect or just to set the "indicators." The first value is placed in the "indicators" (the register M-T), and a succeeding conditional branch can look at that value in deciding whether or not to branch.

NOTE

In VM1, multiple-value calls allocated a block on the PDL to receive the values and arranged that this block would be on top of the stack after the return. In VM2, there is no reserved block; the values are simply copied over the frame of the function that is returning.

D-RETURN and D-TAIL-REC are the two forms of tail-recursion. D-RETURN is non-destructive tail-recursion; it behaves exactly like a D-PDL call until return time. The return microcode will scan back through the stack until it finds a D-INDS or D-PDL frame, and return directly to that frame, through all intervening D-

RETURN frames. D-TAIL-REC is destructive (colloquially, "frame-eating") tail-recursion: at call time, it assumes the state of the caller's caller and copies the arguments over the caller's frame, thus obliterating it. It actually splices out intervening D-TAIL-REC frames at call time, so that at return time it does not have to do the scanning that D-RETURN returns do. This offers substantial space savings and some time savings, at the expense of making the code difficult to debug. The compiler will use D-TAIL-REC instead of D-RETURN when it is safe to do so and the compiler optimizations are set such that speed and space are emphasized and safety is deemphasized. Both D-RETURN and D-TAIL-REC calls copy the return fields (see the section on Return-Fields) from the current call-info word into the new call-info word at call time, D-TAIL-REC because that is part of assuming its caller's caller's state, D-RETURN because it is handy to have that information available before it starts scanning.

The fifth call destination is D-MICRO, used when the microcode calls out to Lisp (see the section Calling-Out). It acts exactly like D-PDL, except that where returns involving the other four destinations restore state and continue at the next macroinstruction, a D-MICRO return will restore state and continue at the next microinstruction.

There are two basic return instructions, known in the microcode as RETURN-1 and RETURN. RETURN-1 is special-case code for handling single-value returns, because they are by far the most common, while RETURN handles the multiple-value cases. All the return macroinstructions (which will be discussed later) are built on these two. The return-types are described in the section on Return-Fields, but they can be grouped into pairs as far as returning is concerned. Only-one-result-needed and multiple-value-list situations are similar in that only one value has to be handled, while normal-return and return-all-values situations may require the movement of several values. As a result, RETURN-1 will end up handling returns of the first pair, even though from the actual macroinstruction it may not appear so.

Adding to the complexity of returning is the necessity of cleaning up after the function. This breaks up into two parts, cleaning up after the frame and making sure the values are safe to return. Cleaning up after the frame means looking at the marked-frame flags (see the section on State-Fields) and acting accordingly. If the environment-pointer-points-here flag is set, some lexical closure has included part of this frame in its environment, and the values must be copied out of the stack before the frame can be removed. The binding-block-pushed flag indicates that this frame has a matching frame on the special PDL whose bindings must be restored. The trap-on-exit flag is a debugging aid that allows the user to examine the frame being exited and the values being returned. The microstack-pushed flag indicates that a block on the special PDL contains some saved

microstack words that must be restored.

To make sure the values are safe to return, we pass them through the "return barrier." This barrier makes sure that stack-lists are copied out to the heap and that lexical closures do what they have to do, which involves setting the environment-pointer-points-here flag in appropriate frames and copying out the values from this frame that are needed in the closure's environment object. Returning passes every value through the return barrier for every frame returned from.

11.9.2 Details.

Returning a single value is fairly straightforward. RETURN-1 dispatches on the combination of the return-type and the saved call-destination, picking up the top of the stack into M-T along the way (every return sets the indicators). It then passes the value through the return barrier and checks the marked-frame flags, and restores the state of the preceding frame. Next, it advances the instruction stream to pick up the macroinstruction at which it will continue (see the step on instruction-stream in the Call-Action section), and during the memory reference checks for PDL-buffer overflow. Finally, it picks up the macroinstruction and macro-decodes it while pushing the returned value on the stack.

The destination and return-type differences are small:

* D-INDS returns do not push the result.
* Multiple-value-list returns make a list of the result or results first, then treat the list as a single value.
* D-RETURN returns continue the steps above for successive frames until one of them has a call-destination of D-INDS or D-PDL.
* D-MICRO returns back up the location-counter and re-advance the instruction stream to restore the contents of the macroinstruction buffer to what it was at the time of the call-out, but do not macro-decode it.
* If the return-type was normal-return, NILs are pushed if necessary to match the number of results wanted.
* If the return-type was return-all-values-with-count, a 1 is pushed above the single value.

Multiple-value returns are slightly more complex. All the simple cases that happen to be single-value returns, such as multiple-value-list or only-one-result-needed situations, use the single-value code described above. The harder cases have to handle

passing more than one value through the return barrier, trimming excess values if more are being returned than are wanted, and pushing the count if the return-type is return-all-with-count. The hardest case is D-RETURN, because the obvious is not the most efficient. Multiple-value D-RETURN returns leave the values right where they are until the ultimate frame is found, then copy them all at once. They also do the trimming of values early and the augmentation with NILs late, because that minimizes the number of values that have to be passed through the return barrier and copied. Otherwise, the algorithm is the same as the single-value return.

There are a lot of return macroinstructions:

* RETURN is a mainop that returns one value and taking it from any mainop source.

* The auxops RETURN-0 through RETURN-63 return the indicated number of values, taking them from the top of the stack, first value deepest.

* The auxop RETURN-N is like RETURN-0, etc, but it takes the number of values from the top of the stack before taking the values from there.

* The auxop RETURN-LIST takes a list of values and returns them as if they were individual values. It could be emulated by spreading the list, pushing the length of the list, and using RETURN-N, but it tries to be smart about multiple-value-list returns and excess values.

* The auxops RETURN-T and RETURN-NIL return T and NIL, respectively. We seem to do that a lot, so they are special-cased.

* The auxop RETURN-PRED returns T if the indicators are non-NIL, else it returns NIL. The auxop RETURN-NOT-INDS is the opposite, returning T if the indicators are NIL and T otherwise.

11.10 CATCH AND THROW

Catches and throws are among the most radically redesigned parts of VM2, relative to VM1. In VM1, catches were open call-blocks for the function *CATCH, with special ADI to record the restart-PC and other things, whose first argument was the catch tag. Throws would scan the list of all open call-blocks to find the open catches. Multiple-value throws would essentially do a multiple-value return on all but the first value, then do a single-value throw with it. In VM2, things are different. Catch-blocks are entirely divorced from call frames. Throws scan

a list of the open catch-blocks, and multiple-value throws are handled cleanly. This section will describe the VM2 catch and throw mechanisms.

11.11 CATCH

One of the design goals in VM2 catch and throw was to make catches cheap and push the expense onto throws, figuring that many catches are set up, but few throws are taken. Another was to clearly delimit the scope of each catch (in VM1, catches are frequently closed by popping the open call-block off the stack, just like discarding any other data). These goals led to the following rules:

1. All catches are opened with the auxop %OPEN-CATCH or a variant.

2. All catches are closed with the auxop %CLOSE-CATCH. It may not be assumed that returning from a function will close any open catches remaining within its frame, for example. The only implicit closing of a catch is when it is thrown through on the way to another catch.

3. Unwind-protects are exactly like any other catch, except that (a) they are closed with the auxop %CLOSE-CATCH-UNWIND-PROTECT, and (b) the restart-PC of normal catches points at the matching %CLOSE-CATCH, while the restart-PC of an unwind-protect points at the first instruction of the undo-forms.

4. The %CLOSE-CATCH-UNWIND-PROTECT of an unwind-protect will detect if the catch was thrown to, and if so will continue the throw. The throw does not retain control throughout the throw; rather, there are a series of throws that add up to the complete one, each bounded by the intervening unwind-protects.

A catch-block contains six words:

1. %catch-block-catch-tag, a Lisp object that is the tag for this catch. Tags are matched using EQ, so it is usually a symbol, but is occasionally a fixnum or stack-list (condition-handling frequently uses stack-list tags).

2. %catch-block-restart-pc, a fixnum that is the relative PC of the instruction at which execution is to resume if this catch is thrown to.

3. %catch-block-number-of-results, which is either a fixnum indicating the number of results that will be accepted by this catch if thrown to, NIL meaning that it wants all the values with the count on top, or T

meaning to collect all the thrown values into a list. The function of this word is analogous to a combination of the return-type and number-of-results field in the call-info word (see the section on Return-Fields) when returning from a function. This field does not come into play unless this catch is the target of a throw.

4. %catch-block-special-pdl-level, a locative recording the level of the special-binding PDL at the time the catch was opened. This word is necessary to undo any special bindings that may have been done inside the catch, but it is only used if the catch is thrown to.

5. %catch-block-saved-catch-pointer, a locative or NIL that indicates the previous open catch. A special microcode register, M-CATCH-POINTER, points to the most recent open catch, and the catch-blocks form a singly-linked list. If there are no open catches, M-CATCH-POINTER is NIL. M-CATCH-POINTER is saved in each stack-group.

6. %catch-block-tag-being-thrown, initially NIL, but filled with the tag being thrown to when this catch is involved in a throw. It is this word that enables %CLOSE-CATCH to decide whether to continue the throw.

The first two words, the tag and the restart-PC, are pushed before opening the catch. %OPEN-CATCH pushes a 1 as the number of results and %OPEN-CATCH-MV-LIST pushes T, but %OPEN-CATCH-MULTIPLE-VALUE requires that its number of results be pushed for it. The other three words are set up by the open-catch microcode.

%CLOSE-CATCH is designed to be fast. It assumes that if there is no need to continue the throw, the preceding instructions have left the values in the desired state, and it just splices the catch block out of the stack, leaving the values on top. It also assumes that any specials bound within the catch have been unbound, so it does no unbinding. Its algorithm is:

First check the tag-being-thrown word. If it is NIL, there has been no throw. If it is non-NIL, but the catch-tag is not T, there has been a throw, but this catch is not an unwind-protect, so the throw does not need to be continued. In these two cases, simply update M-CATCH-POINTER and copy everything above the catch-block down over it, splicing it out. It is thus very fast to open and close catches if they are not needed.

NOTE

NIL is therefore not allowed as a tag argument to THROW.

If there was a throw and the catch-tag is T, then this catch is an unwind-protect, and the throw needs to be continued. If the tag-being-thrown word is T, then the throw was actually an *UNWIND-STACK (see the section on Unwind-Stack).

NOTE

T is also not allowed as a tag argument to THROW.

If the number-of-results word in the catch-block was NIL, then the number of thrown values is atop the stack; otherwise, all values above the catch block are taken to be thrown values. In any case, the throw is continued by jumping into the throw code, which is described next.

11.12 THROW

Throwing is a two-pass process. In the first pass, the throw microcode searches the list of catch-blocks for one whose tag matches the throw-tag. In the second pass, the throw microcode unwinds the stack, cleaning up frames as if it were returning through them (see the section on Returning), leaving the thrown values above the target catch-block.

The simple description above leaves out all the complexity of the operation. For instance, the first pass also looks for unwind-protects, because although it is required that a matching catch exist, the throw may actually be to an intervening unwind-protect. Also, a catch with a tag of NIL will catch any throw; such a catch is produced by the macro CATCH-ALL. CATCH-ALL differs from UNWIND-PROTECT in that there are no undo-forms and the throw is not continued. Unlike VM1, CATCH-ALL does not receive any extra values like the throw-tag; it is simply a catch that catches everything.

Another area of complexity is hardware PDL buffer cache management. Returns via D-RETURN (see the section on Returning) also unwind one or more frames as they return through them. Adjacent call frames are known to be relatively close together, and D-RETURN chains are likely to be short, so is makes sense to refill the PDL-buffer whenever needed in the course of the

return. Catch-blocks, by contrast, tend to be more widely separated, so frequent PDL-buffer refills are not usually a good idea. The alternative is to access the PDL-buffer through the virtual-memory interface, which is much slower but which does not involve reading so many words. The compromise that the microcode actually uses is to work through the PDL-buffer for the frames that are there, but to use the virtual-memory path for the deeper frames that are not, refilling only at the end to bring the target frame into the PDL-buffer.

A third complexity is the handling of values. Once the throw has unwound to the target frame, it must check the catch-block for the number of values wanted and the form they are to take. It is at this point that missing values are augmented with NILs, excess values are trimmed, and lists are built for the multiple-value-list case. Values are passed through the return barrier at the beginning of the second pass, after it is known where they are going to be copied to, and it is only done once (unlike returning, which passes them through for each frame).

NOTE

Multiple-value returns, when the return-type is multiple-value-list, build the list first and treat it as a single-value return. While it might also be a good idea for throws, it is not done at this time.

Just like returning, there are two kinds of throw instructions, THROW, which throws a single value, and THROW-N, which throws any number of values, taking the count from the top of the stack. Both instructions are auxops, and the compiler generates both from the Lisp THROW special form, choosing the appropriate one from context. In structure they are very similar, the difference being that THROW can pick up its single value into M-T, while THROW-N must leave its values on the stack undisturbed while it unwinds. THROW can therefore restore all the state of the target frame and refill the PDL-buffer before putting its value on the stack, while THROW-N must copy its values, possibly through the virtual-memory path, to their destination before refilling.

11.13 UNWIND-STACK

*UNWIND-STACK is a mating of throw and return. It exists for stack-manipulating programs such as the debugger, and as such is a subset of the *UNWIND-STACK of VM1. In VM1, *UNWIND-STACK was the general form of THROW: THROW was *UNWIND-STACK with the count and action arguments NIL. *UNWIND-STACK takes four

arguments: tag, value, count, and action. The tag argument is for partial compatibility with the VM1 *UNWIND-STACK; it must be T. The value argument is the same as the value argument to THROW; it may be any Lisp object. The count argument is the number of call frames to unwind; it must be a fixnum or NIL. The action argument is called when the unwinding is finished; it must be a functional object or NIL. Either the count or the action must be non-NIL; the case of both NIL was equivalent to THROW in VM1 but is an error in VM2.

NOTE

VM2 *UNWIND-STACK behaves exactly as the VM1 version did when this argument was T; the functionality that is not implemented in VM2 is the use of other tags.

*UNWIND-STACK will unwind the frames on the stack, doing all the cleanup it would do if it were throwing through them. It also cleans up after catches along the way. There are three possible situations:

* If the count is a fixnum, it is the number of frames to unwind. If the action is NIL, then the value will be returned from the last frame unwound.

* If the count is NIL, it indicates that all frames in the stack should be unwound. The action must then be non-NIL, and will be called with one argument, the value, after all the frames have been unwound. The action is not permitted to return in this situation. It is often useful for the action to be a stack-group; the debugger frequently uses it in this way.

* If both the count and the action are non-NIL, then the action will be called, with the value as its argument, after count frames have been unwound. The action may return, and its values will be returned as if from the last frame unwound. This case is rarely used.

*UNWIND-STACK honors unwind-protects like throws do. It stops at the unwind-protect catch, leaves the value, the count, and the action atop the stack, and writes T into the tag-being-thrown word. Since T is not permitted as the tag argument to a throw, CLOSE-CATCH recognizes it as the sign of an *UNWIND-STACK that needs to be continued, and calls *UNWIND-STACK. Since the tag-being-thrown slot is the top slot of the catch-block, the four arguments are therefore right on top of the stack.