PRELIMINARY

REFERENCE MANUAL FOR THE CRMS SYSTEM APL PROCESSOR

Paul McJones

July 21, 1972

Center for Research in Management Science
Systems Group

Technical Document

TABLE OF CONTENTS

saves time for the processor and the resultant addresses are in general more compact than the identifiers they replace.

Programs exist for the SIMPLE processor which translate external to internal APL and perform the concomitant editing, debugging, and "system" functions. Thus the sole responsibility of the APL processor is to execute internal APL programs and indeed this processor is capable of nothing else. Since the two processors share core memory, programs and data prepared by the SIMPLE processor are directly accessible by the APL processor.

This manual assumes familiarity with APL\360; for example see APL\360 Reference Manual by Sandra Pakin (Science Research Associates, 1972).

2. APL Objects.

APL is a language for expressing algorithms which manipulate numbers, characters, and n-dimensional arrays. In this section we will describe how these objects are represented internal to the APL processor.

Every value ~~in Meta-APL~~ occupies a single 32-bit word. Because of the dynamic nature of APL, stemming from its lack of compile-time type declarations, each value must carry explicit type information at run-time. Thus type bits and value are packed together into a 32-bit word or descriptor. In Figure $\overset{11}{1}$ are illustrated the formats of the various types of values; some further explanation is in order.

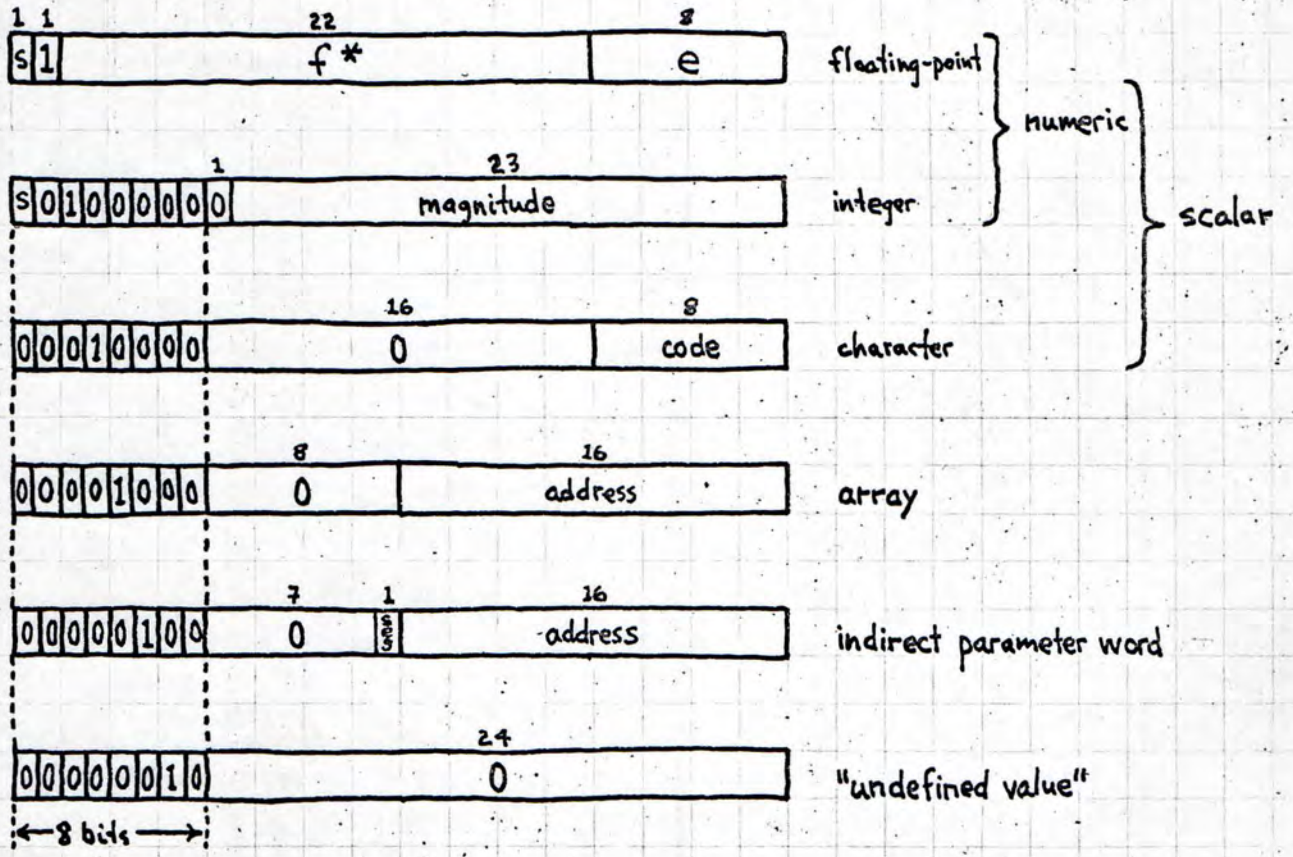
2.1. Floating-point Numbers.

A floating-point number consists of a signed, fixed-point coefficient (with the radix point lying between the first and second significant bits) and an "exponent" or power-of-two scale factor. If F is a floating-point number whose sign, fraction, and exponent are respectively s , f and e (all taken as $\overset{\text{unsigned}}{\wedge}$ integers) then

$$F = (1 - 2s) \cdot (1 + \frac{f}{2^{-22}}) \cdot 2^{(e-128)} .$$

no '-' ?

As can be seen from Figure 1, all floating-point numbers ~~in Meta-APL~~ are "normalized," that is, if $c = (1 + \frac{f}{2^{-22}})$ is the coefficient of a floating point number F , then $1 \leq c \leq 2 - 2^{-22}$. Equivalently, the "leading" bit of F (the first bit after the sign of F) is a one. The bit patterns corresponding to unnormalized numbers are thus available to encode other types of values; Figure 1 shows how the type field of a value not a floating-point number is used.



* The coefficient of a floating-point number consists of a leading "1" bit followed by the f (fraction) field.

APL Descriptor Formats

Figure 1.1

2.2. Integers.

An alternative representation for integers of magnitude less than 2^{23} is available. In particular, zero must be represented in this format since it is not a normalizable floating-point number. A point concerning zero is that it is always represented with the sign bit $s = 0$; "minus zero" is not allowed.

2.3. Characters.

A scalar character is represented by an 8-bit code chosen from an extended-ASCII character set which includes the special characters necessary for APL.

2.4. Arrays.

A descriptor for a scalar (numeric or character) might be termed an "immediate descriptor" since it completely specifies the identity of its value. In contrast, an array descriptor merely points to an array block stored elsewhere which contains the elements and shape information of the array. This scheme has two benefits:

(1) All descriptors are a uniform 1 word in size;

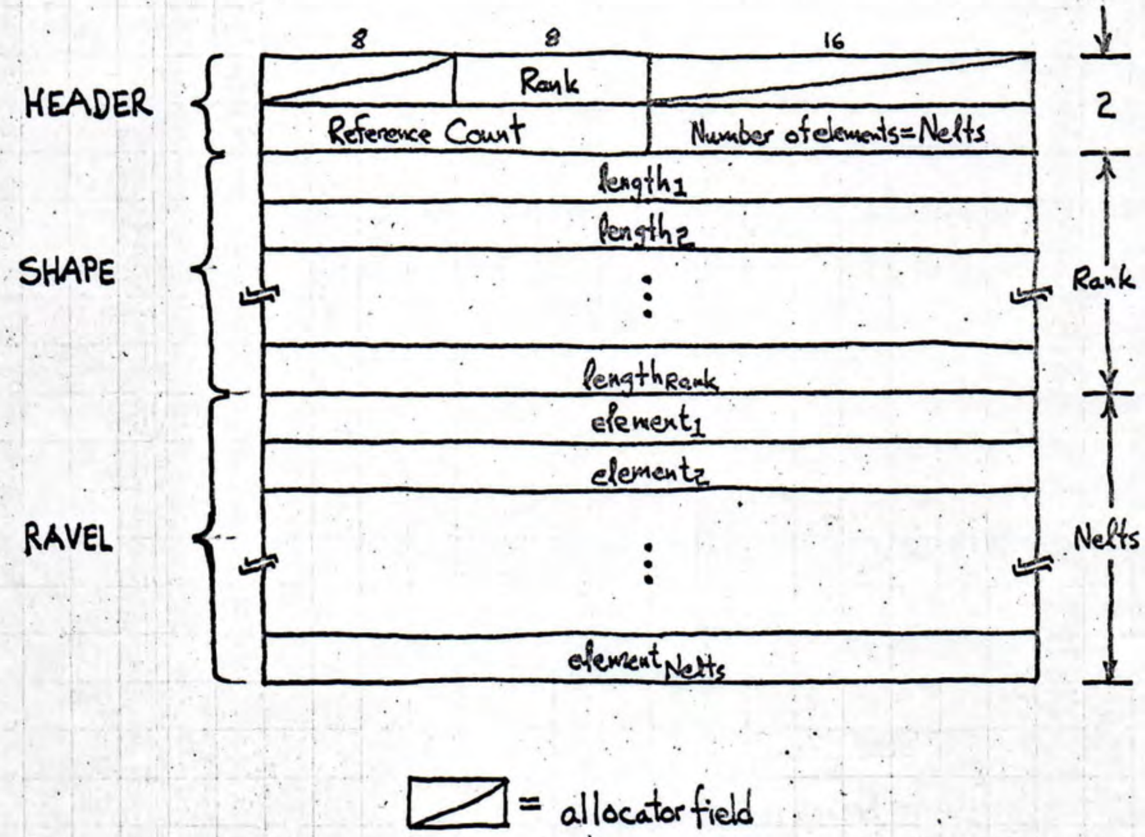
(2) Several distinct array descriptors may point to the same array block.

The form of an array block is given in Figure ~~2~~ 1.2. Following is a description of the various fields.

The rank of an array is the number of dimensions it has; the numbers length_1 ~~width_1~~, length_2 ~~width_2~~, ..., $\text{length}_{\text{rank}}$ ~~$\text{width}_{\text{rank}}$~~ in the array block are called the shape of the array. and must be regular integer type descriptors for nonnegative numbers. If the value of length_i ~~width_i~~ of an array equals n , then there are n positions ~~in~~ along the i^{th} coordinate of the array; the total number of elements in the array is elements

$$\text{Nelts} = \prod_{i=1}^{\text{rank}} \text{width}_i \quad .$$

12 January 73



APL Array format

Figure 1.2

may be any scalar descriptors, and

The cases Nelts = 0 and Nelts = 1 are both possible.

The elements of the array are stored in a linear sequence known in APL as the ravel of the array. The distance, in ravel order, between two neighboring elements of the ith coordinate is called the ith stride. A formula for the strides of an array with widths $w_1, w_2, \ldots, w_n$ is:

$$\text{stride}_n = 1$$

$$\text{stride}_i = \prod_{k=i+1}^{n} w_k , \quad \text{for } 1 \leq i < n .$$

APL arrays must be homogeneous, that is either all elements of an array must be numbers or else all elements must be characters. (In the former case, mixed floating-point and integer numbers are permissible.) To gain compactness, the elements of a character array are packed four to a word. A bit called the element-type in an array block distinguishes character and numeric arrays.

Since it is possible for several array descriptors to point to the same array block, a reference count is used to determine when the storage for an array block may be released. This count is a field in the array block giving the number of array descriptors currently pointing to this array block; it is updated whenever such a descriptor is created or destroyed. If a reference count is reduced to zero, then the storage occupied by that array block ~~may be reused.~~ is returned to a free pool.

(For further details, see section 3.4)

2.5 Indirect Parameter Word.

This descriptor is not for an ordinary value, but is used in conjunction with call-by-reference parameters to functions. The descriptor contains the identity of a local or global variable, and itself resides in some formal parameter of a function; references by load and store instructions directed at the formal variable instead affect the variable identified by the indirect parameter word. The variable to be affected is identified by one bit signifying local or global (0⇒local, 1⇒global) and sixteen bits specifying an address. For a local variable, this address is relative to the state variable SBASE (see section 3.3); for a global variable this is just the ordinary address of that variable in the data segment.

"state variable" seems like wrong term.

2.6 "Undefined Value".

This descriptor (which we are defining!) is used to initialize global and local variables so that references to variables which haven't been explicitly assigned a value will cause a value error trap. The processor automatically initializes local variables with a descriptor for the "undefined value"; such initialization for the global variables must be done in software on the SIMPLE processor before starting the APL program.

3. The data segment

A program for the APL processor consists of a pair of procedure segments together with a data segment. The procedure segments (described in section 4) contain reentrant code, and express the invariant algorithm of the program. The data segment, on the other hand, contains all the storage used to run the program: that occupied by the variables of the program as well as that used by the processor for its internal data structures.

Figure 3.1 shows the overall layout of the data segment. (Higher addresses are toward the bottom of the figure, as is the case throughout this document.) The region labeled "global storage holds" information accessible throughout the execution of the program, and is statically allocated before execution begins. "Local storage" consists of information for each of a sequence of function activations. It is allocated using a last-in-first out discipline dictated by the nested activations of functions resulting from the flow of control through the program. Finally, array block storage contains arrays each of which is referenced by one or more global and/or local variables. Space in this area is dynamically allocated.

3.1 Processor State Area

An assortment of state variables used by the processor are gathered together at the beginning of the data segment (see Figure 3.2). The individual state variables will be described in the sequel: SBASE, LTOP, and LBASE in section 3.3 (runtime stack); TRAPCLASS and TRAPNUMBER in
→ section 5.? (traps); saved-NELTS, -OPNO, -APTR, -BPTR, and
→ -RPTR in section ?.? (interruption of array operations); and ROVER and BLOCKPTR in section 3.4 (array block storage). This leave FLAGS, which is really a collection of one-bit variables, as shown in Figure 3.3. IORG is the APL index origin (see section 5, instructions). FCNCTRC and FCNRTRC control function call and return tracing (see section 5, call and return). STEP controls step tracing, another debugging tool (see section
→ 5.?, step trap). INTRPT and the "temporary" flag bits are discussed in section
→ ?.? (interruption of array operations).

This cell contains a descriptor for value of the variable, be it scalar, array, or "undefined".

3.2 Global Variables

Each global variable in the program is allocated a single cell in a special area of the data segment. (Note that global variables are addressed by non-negative integers (0, 1, 2, etc.). The processor automatically maps the global variable addresses appearing in instructions into data segment addresses. This transformation consists merely of adding 8 (the length of the ~~preceding~~ processor state area). When execution of a program begins, each of the global variables should contain the the standard "undefined" value (see section 2.?, the undefined value); the processor does not itself initialize these variables.

all fixed?

These two state variables contain addresses relative to SBASE, easing relocation of the stack within the data segment.

3.3 Runtime Stack

As was mentioned earlier, the processor maintains a pushdown stack for storage local to each function activation. As shown in Figure 3.4, a stack frame (entry) contains one word for each formal parameter or other local variable, two words of local processor state information, and a variable number of words containing descriptors for temporary values. At any moment the most recently activated function is being executed, and its stack frame, called the current frame, is on top of the stack.

Several state variables are used to delimit the stack. SBASE contains the address in the data segment of the first word of the first (oldest) stack frame, and is never changed by the processor. LBASE always points to the current stack frame, and LTOP points to the topmost temporary value cell of the current frame. The processor updates LBASE for each function call or return. LTOP is decremented by the various processor instructions when they remove their operand(s) and is incremented when they stack their result(s).

LBASE points not to the first word of the current frame, but to the first word of local state information. The local variables are addressed by negative integers (-1, -2, etc.), which the processor interprets relative to LBASE. If a function has a result variable, it is always assigned the address -1 (so the result variable is in the cell immediately preceding the first word of local state information).

The local state information in a stack frame merits some discussion. Since stack frames are not of constant length, some form of backpointer is necessary; this need is filled by the "last LBASE" field, which contains the value of LBASE for the function activation immediately preceding in the stack. The "function descriptor" and "PCTR" fields in a stack frame refer not to the preceding activation, but to the given one. These fields identify, respectively, the function activated and the location in that function of the next instruction to be executed. In the context of a given program, a function is identified by one bit to select between the two procedure segments together with an 11-bit integer designating one of the functions in that segment. The location of an instruction is always specified as an address, in bytes, relative to the containing function body, of the first byte of the instruction (which may be one or more bytes long).

3.4 Array Block Storage

Descriptors for array values exist in various parts of the data segment: in global variables, in local (and formal parameter) variables, and in temporary storage cells. The actual arrays, however, are gathered together in array block storage. Space in this region is dynamically allocated using a scheme of "boundary tags and roving free pointer" [Knuth 1968].

Array block storage consists of a sequence of reserved and free blocks. Each reserved block contains an array, whose reference count field is by definition nonzero. (See section 2.? for the description of the structure of an array per se.) As the name suggests, free blocks contain space available for future use. In the interest of avoiding fragmentation of storage, no free block is allowed to exist adjacent to another free block (see Figure 3.5 for a typical configuration).

4

When the array in a reserved block is no longer needed, there are four cases to consider:

(1) Both the preceding and following blocks are free; the block being freed should be coalesced with both its neighbors into one big free block.

(2) Only the preceding block is free; it should be extended to include the following newly freed space.

(3) Only the following block is free; it should be extended to include the preceding newly freed space

(4) Neither the following nor the preceding blocks are free. The block being freed can't be coalesced.

fixed?

Might be better to say "coalesced" or "extended" for not both

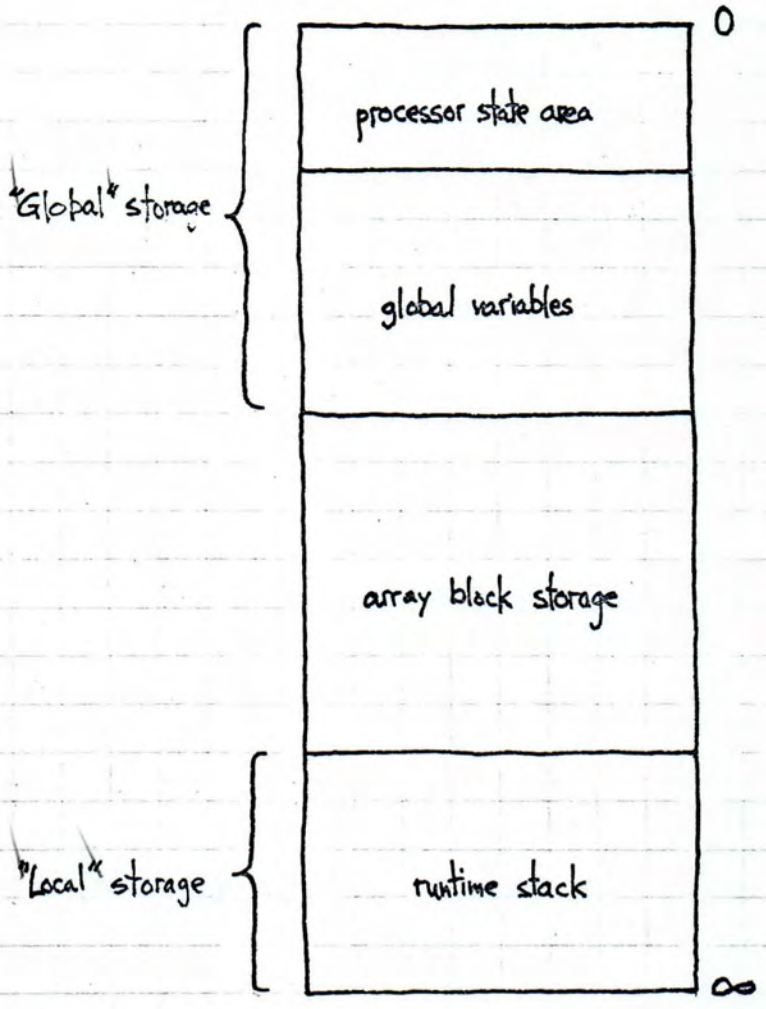
Figure 3.6 shows the format of "reserved and free" blocks from the allocator's point of view (thus leaving out the details of the array in a reserved block). Note that given the starting location of a block, it is possible to tell whether the block is free or reserved (since the leftmost bit of the first word of every block is the boolean field THISFREE). Also, every <u>reserved</u> block has another boolean field, PREVFREE, indicating whether the preceding block is free. With these fields and the SIZE fields it is possible to discriminate between the cases (1) through (4) listed above.

All of the free blocks in the array block storage area are joined into a doubly linked ring using the NEXT and LAST fields appearing in every free block. The state variable ROVER always points to some member of this ring. To ensure that the ring is always nonempty (so the value of ROVER is well defined), there is a dummy free block with an indicated size of zero which can never be allocated. This dummy free block serves another purpose. It lies at the very end of array block storage, and has its THISFREE bit set false (!) to inhibit coalescing with the last real block. Figure 3.7 gives a typical configuration of the free block chain.

Note (in Figure 3.6) that a reserved block may end with several words of "slop" which are not necessary to hold the actual array. A field named SLOP in the first word of the array holds the number of slop words, which incidentally are not counted in the SIZE field. The main reason for slop words is that a free block must be at least three words long; when a smaller packet of unused words ~~are~~ is created by breaking a larger free block, it must be attached to the <u>preceding</u> reserved block ? as slop.

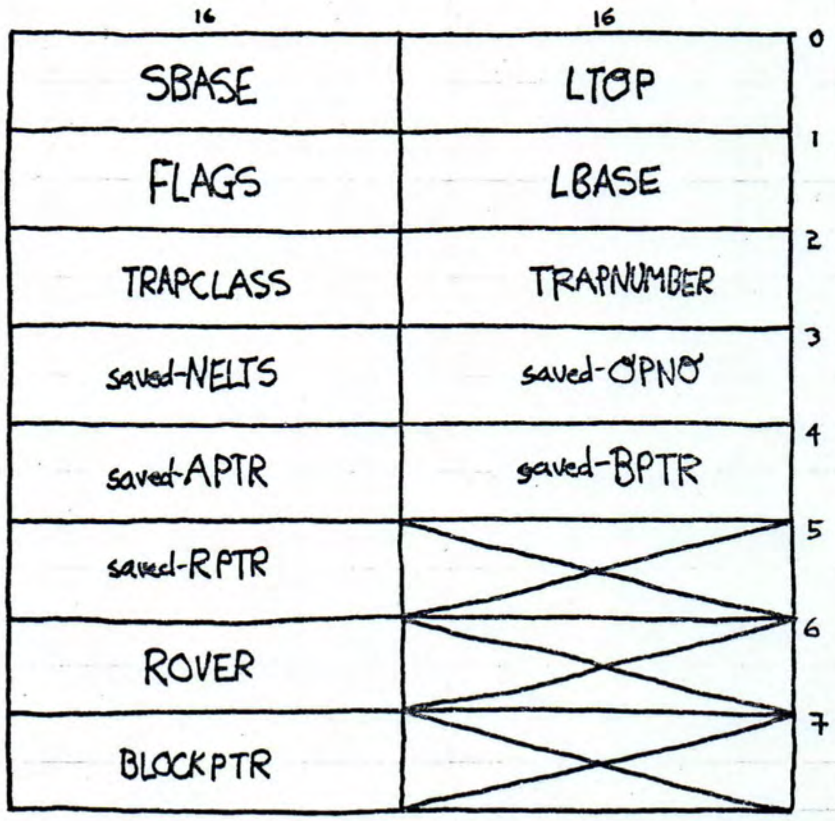
? or new block?

11 January 73



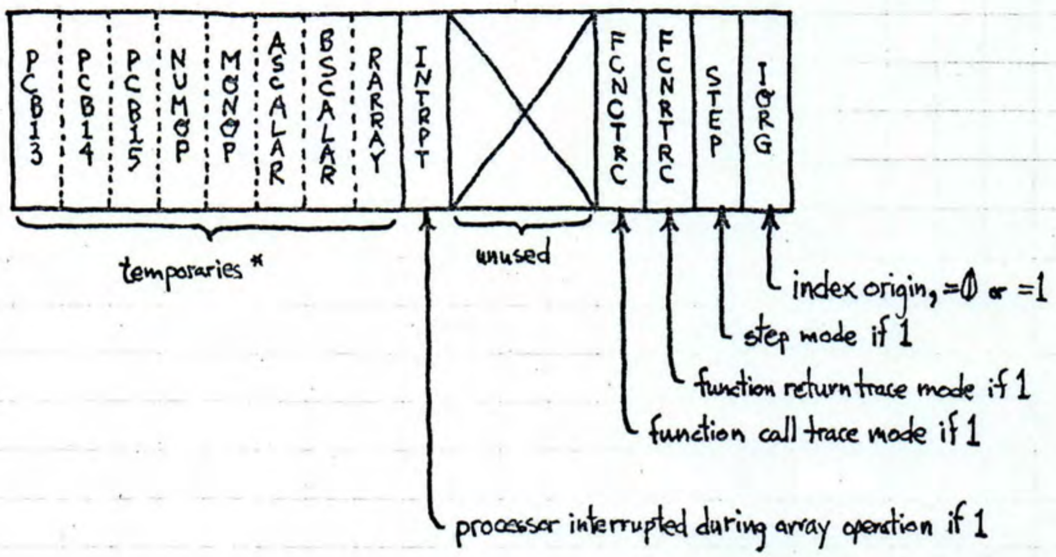
APL Data Segment

Figure 3.1



APL Processor State Area

Figure 3.2

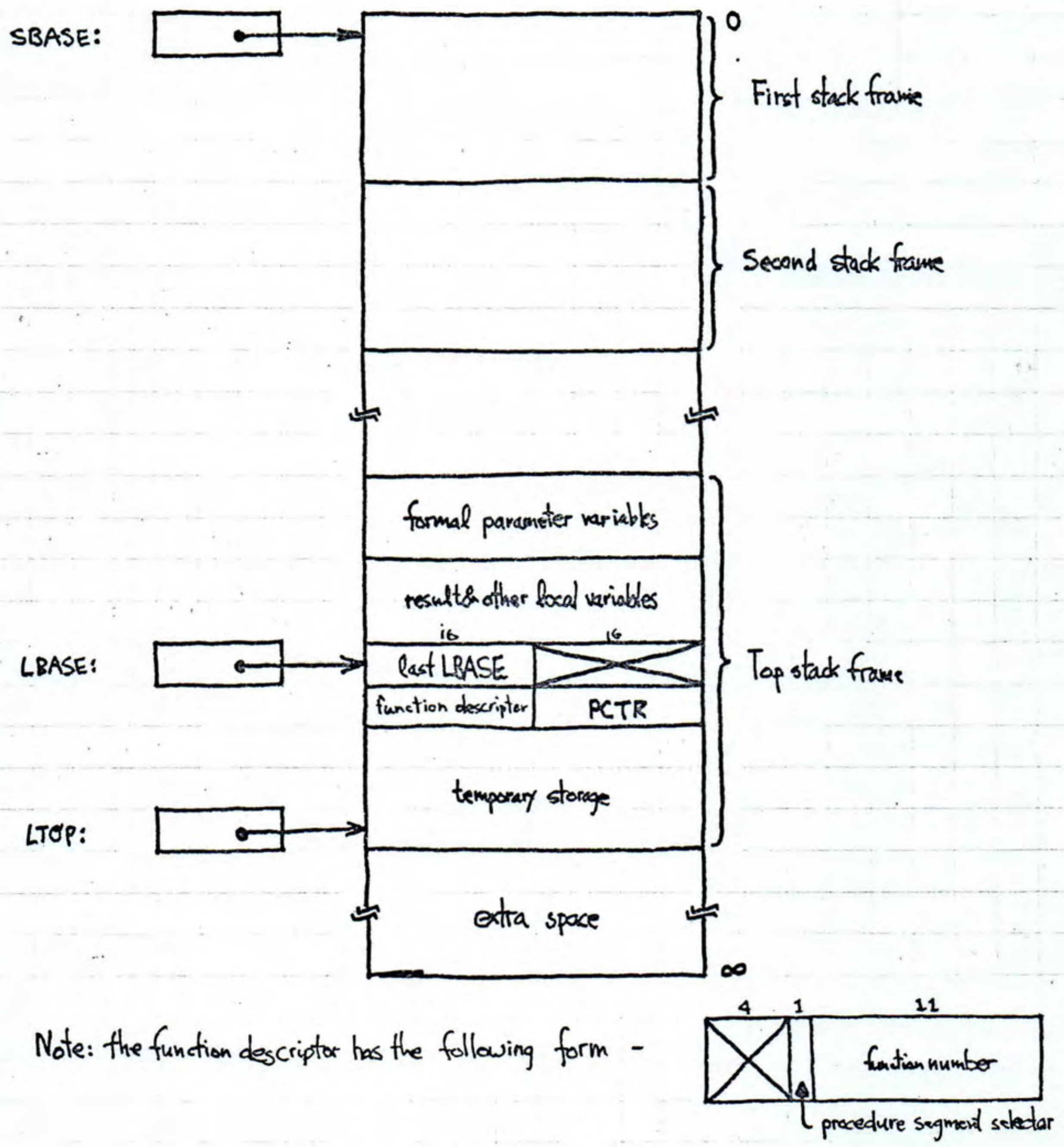


* If the processor is stopped with FLAGS$INTRPT=1, then these temporary bits must not be changed.

APL Processor State FLAGS

Figure 3.3

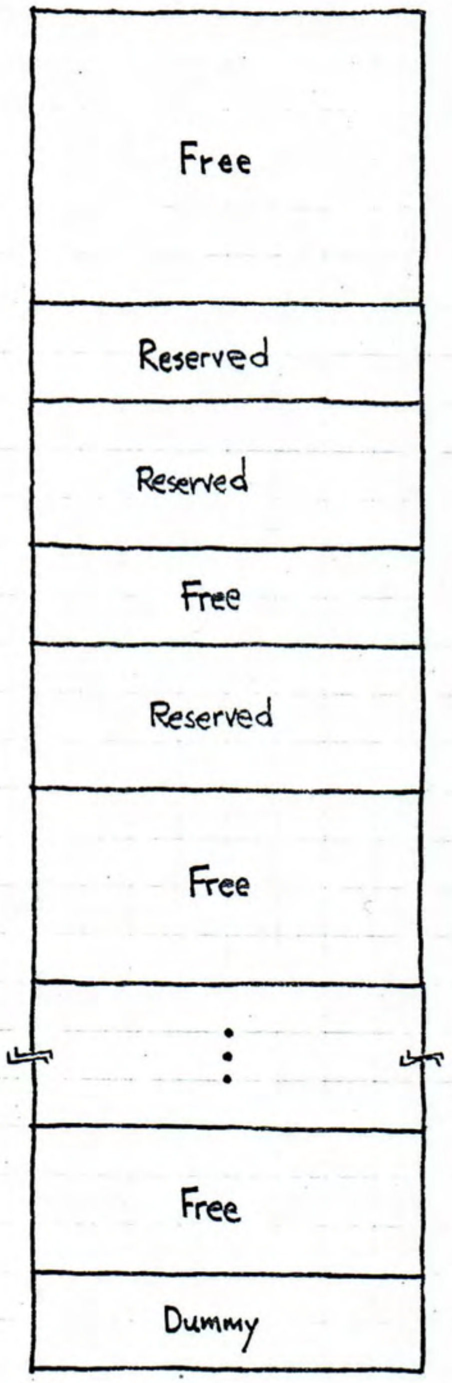
11 January 73



APL Runtime Stack

Figure 3.4

12 January 73



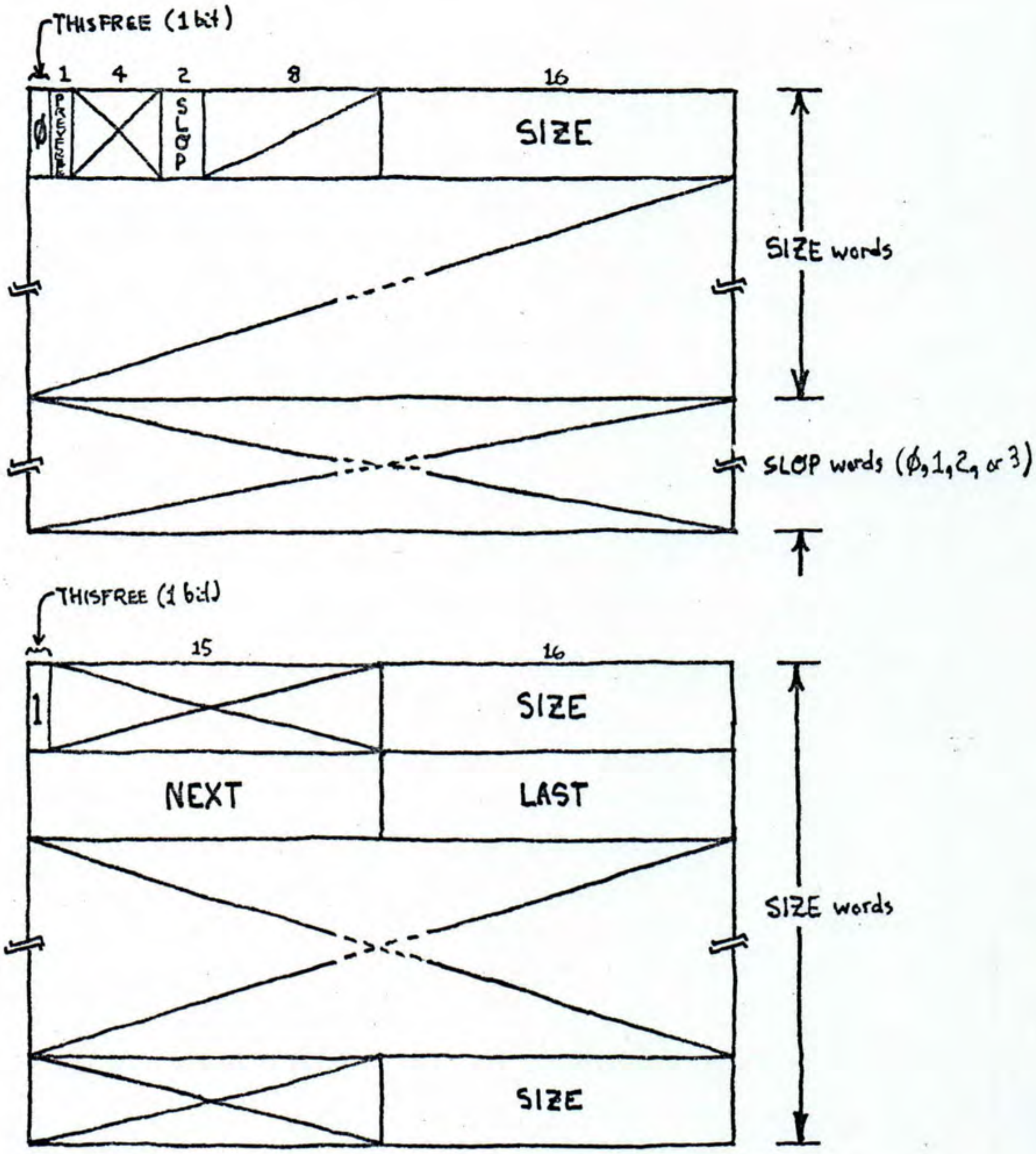
Typical APL Array Storage Configuration

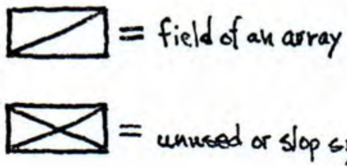
Figure 3.5

17 January 1973

Reserved block:

Free block:





APL Array Storage Reserved & Free Block Formats

Figure 3.6

1. Introduction.

One of the major goals of the new CRMS computer system is to provide efficient, time-shared APL. The system contains two processors with common (core) and auxiliary (disk) memories, as well as controllers for a variety of peripheral devices. One processor (the primary subject of this document) is specially tailored for the execution of APL programs. The other processor is optimized for executing programs in the SIMPLE programming language and is described elsewhere.

Rather than directly interpreting source language character strings, the APL processor executes an internal "machine language" form adapted for that purpose. Three types of changes are involved in the mapping from external to internal APL:

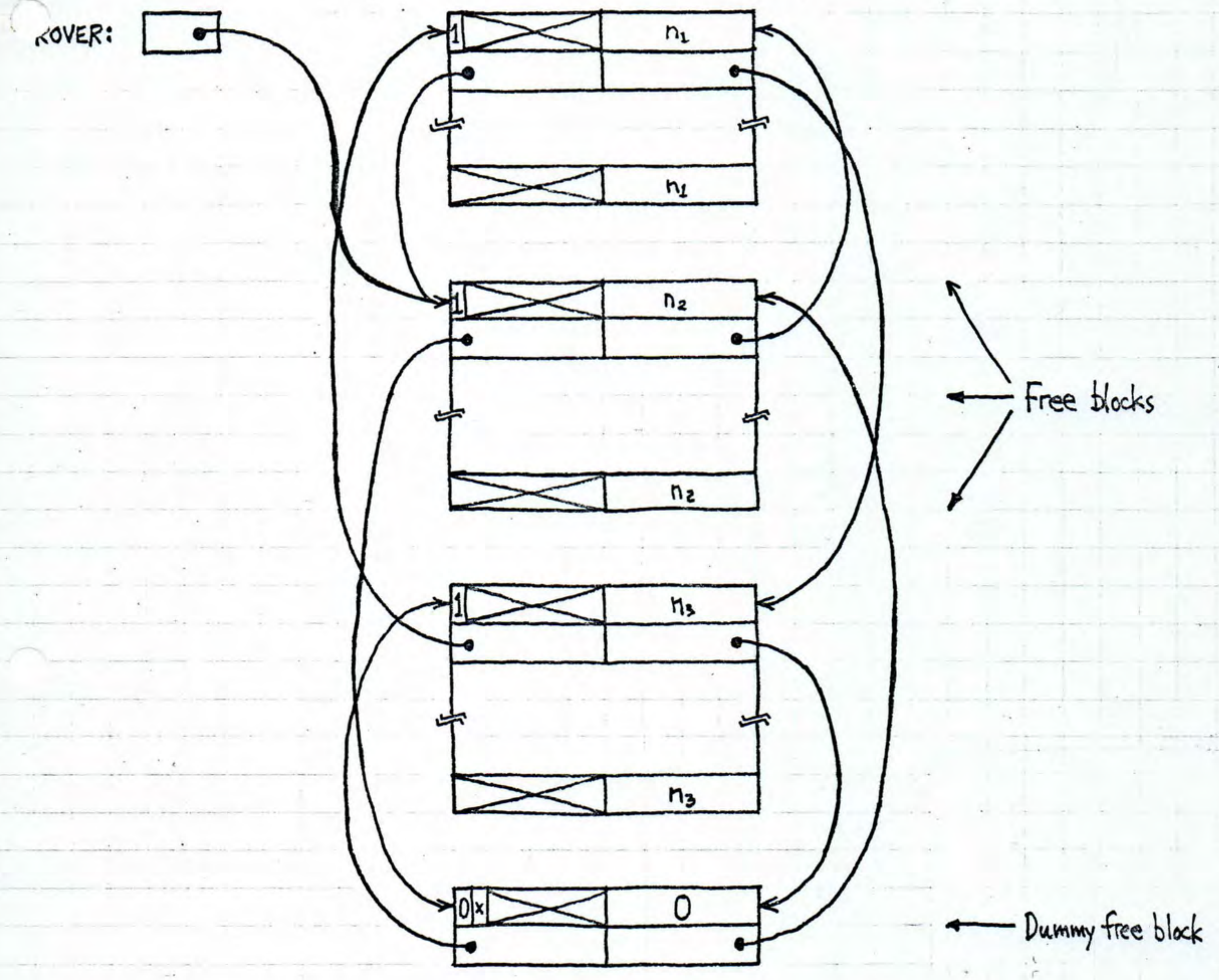
(1) (literal notation => binary notation) Each numeric or character literal is replaced by a binary encoding of the number or character.

(2) (identifier => variable address) An arbitrary correspondence between the identifiers of a program and unique consecutive integers is formed; then all occurrences of an identifier are replaced by its number (address).

(3) (infix => postfix) The expressions are reordered so that an operator follows its operands instead of preceding (monadic) or separating (dyadic) them.

These changes are made for technological rather than logical reasons. Arithmetic can be performed much faster on operands in binary notation than in decimal notation. Similarly, binding identifiers to addresses in advance

11 January 73



APL Array Storage—
Typical Free Chain Configuration

Figure 3.7

Dave

typist will type this as "some"

4. The Procedure Segment

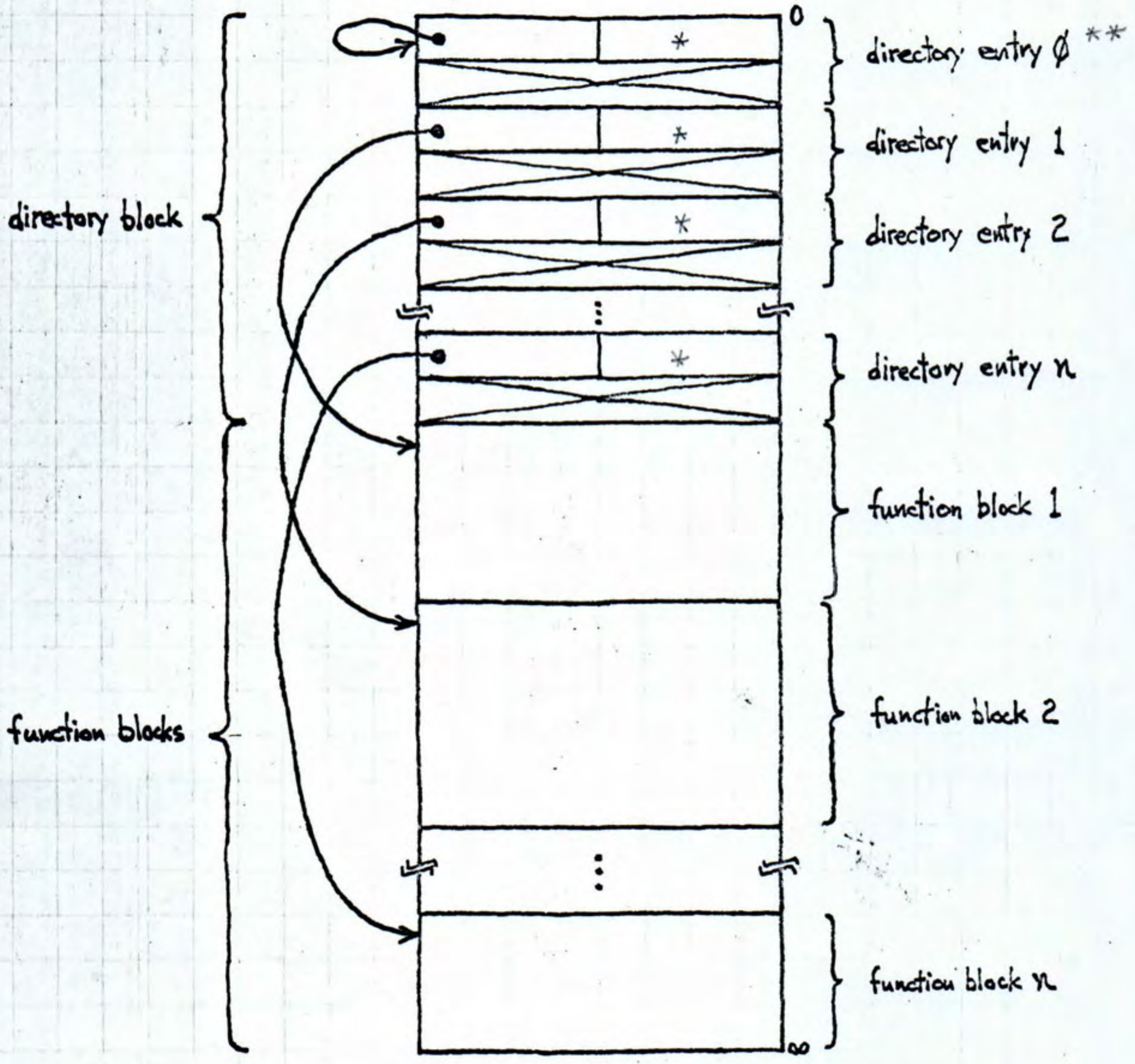
4.1. Function Blocks and the Function Directory

Just as an APL source program consists of a set of function definitions, a procedure segment contains a sequence of function blocks. In the interest of function block relocatability, the procedure segment begins with a directory each of whose entries contains the starting address in the segment of the corresponding function block. Thus each function can always be referred to by a unique ordinal number, even if the lengths of functions change. Figure 4.1 shows the overall layout of a procedure segment.

4.2 Components of a Function Block

A function block is the compiled version of an APL source function definition, and is diagramed in Figure 4.2. The two-word header contains various count and length fields; it corresponds to the header or "line zero" of a source function. Next in a function block is the actual code, a sequence of variable-length instructions. Completing a function block is its line table, which serves to map APL source statement numbers into addresses in the code.

12 January 73

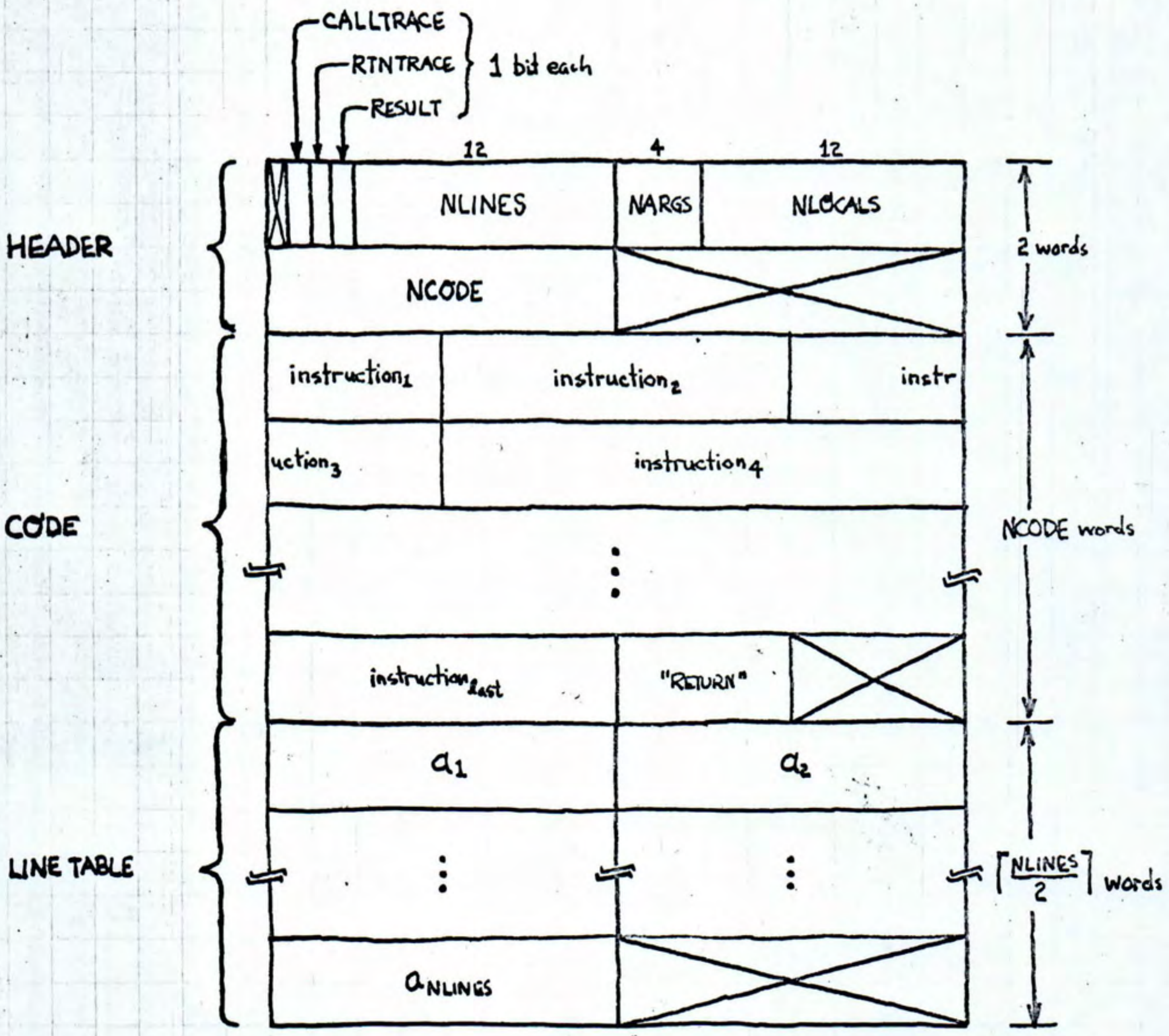


* This field, not used by the APL processor, contains the length of the corresponding block.
** Directory entry ∅ corresponds to the directory block itself.

APL Procedure Segment

Figure 4.1

18 January 73



APL Function Block

Figure 4.2

4.2.1 Function Block Header

The fields in the header of a function block have the following meanings:

CALLTRACE -- if this bit is one and the state flag FCNCTRACE is also one, the processor will trap just *after* executing a call instruction designating this function;

RTNTRACE -- if this bit is one and the state flag FCNRTRACE is also one, the processor will trap just *before* returning from this function;

RESULT -- if this bit is one the function returns a value (the contents of the local variable with address -1);

NLINES -- the number of lines in the source function from which this function block was compiled (also the number of entries in the line table);

NARGS -- the number of actual parameters with which this function should be called;

NLOCALS -- the number of local variables (including the result variable, if any) required by this function;

NCODE -- the number of 32-bit words occupied by the code in this function block.

4.2.2 Code

Instructions for the APL processor are not of fixed length. Most instructions are one, two, or three bytes long. One instruction is five bytes long, and another is 2+4n bytes long (for any $n \geq 0$). Detailed descriptions of each of the instructions is the subject of section 5, but it would be appropriate here to state that the last instruction in the code for every function should be a "RETURN" instruction (see section 5.?). This way a function will return if the flow of control "runs off the end". Following this last instruction will be zero to three bytes of garbage to fill out the last word of code.

4.2.3 Line Table

The sole use of the line table is made by the "BRANCH" instruction (i.e. APL's monadic right arrow). This table consists of a sequence of 16-bit elements, ostensibly one per line of source program. Each element is the function block-relative byte address of an instruction (presumably the first instruction compiled from the corresponding source line). Note that the function block-relative byte address of the very first instruction in a code body is 8, due to the preceding two words (at four bytes per word) of header.

Dave

Section 5 The Instructions

SHORTCONST

This instruction takes no operands from the stack, and places one integer type result on the stack. The sign and magnitude of the result, say s and m respectively, together constitute the 13 rightmost bits of the SHORTCONST instruction itself (see Figure ?.?).

Traps:

(1, 1) -- Runtime stack overflow if room for one more cell on the stack doesn't exist.

Questions

1. Does "ith operand" mean "ith from top of stack" throughout?
2. Is there a general statement anywhere that an inst. which traps (except step trap) is equivalent to a no-op? ("Consults oracle")

The GENSCALAR family

The instructions of this family correspond to (a subset of) the APL "primitive scalar functions". Some are monadic, and the rest are dyadic. Operands of any rank are allowed; the APL\360 rules of conformity are exactly followed [Pakin 1972]. Since operands of rank greater than zero are treated elementwise, the descriptions of the individual GENSCALAR instructions are in terms of scalar operands.

Normally the APL processor obeys stop commands from the SIMPLE processor (see section 3.?). only between instruction executions. The only exceptions are the GENSCALAR instructions, some of which spend as long as 15 microseconds per element of the result (which could have thousands of elements). If a command from the SIMPLE processor arrives during the execution of a GENSCALAR instruction, the APL processor sets the INTRPT state flag and stores some internal quantities in the state variables saved-NELTS, ..., and saved-RPTR (see Figure 3.2). Then the processor stops just as it would between instructions. If the interrupted program is to be restarted later, then the "saved-XX" state variables must be preserved unmodified, as must the state flag INTRPT and the state flags marked as temporaries in Figure 3.3.

Traps from GENSCALAR instructions (also see descriptions of individual instructions):

insert a from the next here

(5,0) -- Rank error if the ranks of the operands for a dyadic operator differ and neither operand is a one-element array.

(6,0) -- Length error if the ranks of the operands for a dyadic operator are the same, but the operand shapes are different and neither is a one-element array.

(3,5)--Missing operands if LTOP<LBASE+3 for dyadic operators, or if LTOP<LBASE+2 for monadic operators.
(3,4)--Unknown descriptor type if either operand is not a scalar and not an array.

α
insert into
previous page

(1,2)--Array block storage overflow if insufficient free space to create result array.

IDENTITY

This monadic instruction returns its (integral or floating-point) numeric operand unchanged.

Traps:

(7,0)--Type error if the operand is of character type.

NEGATIVE

This monadic instruction returns the same value as its numeric operand, except with the opposite sign.

Traps:

(7,0)--Type error if the operand is of character type.

FLOOR

This monadic instruction returns the greatest integer not larger than its numeric operand. The result is of integer type unless the operand is a floating-point number greater than or equal to 2^{23} in magnitude, in which case only a floating-point representation is possible.

Traps:

(7,0)--Type error if the operand is of character type.

CEILING

This monadic instruction returns the least integer not smaller than its numeric operand. The result is of integer type *unless* the operand is a floating-point number greater than or equal to 2^{23} in magnitude, in which case only a floating-point representation is possible.

Traps:

(7,0)--Type error if the operand is of character type.

MAGNITUDE

This monadic instruction returns the same value as its numeric operand, except with positive sign.

Traps:

(7,0)--Type error if the operand is of character type.

NOT

The single operand for this instruction must be logical, i.e. equal to zero or one (either integer or floating-point). The result is equal to one minus the operand, and is always of integer type.

Traps:

(7,0)--Type error if the operand is of character type.

(8,5)-- Domain error if the operand is a number other than zero or one.

TESTNUM

This monadic operator takes any scalar for an operand. The result, always of integer type, is one for a numeric (integer or floating-point) operand and zero for a character operand.

Traps: none.

CONVERT

This monadic operator takes both character and numeric operands. When given a character, it returns the corresponding character code as an integer type value. If given an integral number between zero and 255 inclusive (in either integer or floating-point representation), CONVERT returns the character type value possessing that code.

Traps:

(8,5) - Domain error if the operand is a number which is nonintegral, negative, or greater than 255.

SUM, DIFFERENCE, PRODUCT

These dyadic instructions return respectively the sum, difference, or product of two numeric operands. The result is of integer type if both operands are of integer type, or if the result is zero. In every other case the result is of floating-point type.

Traps:

(7,0)-- Type error if either of the operands is of character type.

(8,1)-- Integer overflow if both operands are of integer type and the magnitude of the result is not less than 2^{23}.

(8,2)-- Floating-point underflow if at least one operand is floating-point and the magnitude of the result is less than 2^{-128} (but is greater than zero).

(8,3)-- Floating-point overflow if at least one operand is floating-point and the magnitude of the result is not less than 2^{128}.

QUOTIENT

This dyadic instruction returns the quotient of its first operand divided by its second operand, both of which must be numeric. The result is always of floating-point type.

(7,0)-- Type error if either operand is of character type.

(8,2)-- Floating-point exponent underflow if the magnitude of the result is less than 2^{-128} (but is greater than zero).

(8,3)-- Floating-point exponent overflow if the magnitude of the result is not less than 2^{128}.

(8-4)-- Zero division if the second operand is zero.

AND, OR

These dyadic instructions return respectively the logical conjunction and disjunction of their operands, as shown in the truth table below. Each operand must be either zero or one (integer or floating point); the result will always be of integer type.

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Traps:

(7,0)--Type error if either operand is of character type.
(8,5)-- Domain error if either operand is a number other than zero or one.

LESS

This dyadic instruction returns one if its first operand is smaller than its second operand, and returns zero otherwise; the result is always of integer type. Execution of the LESS instruction involves finding the difference of its operands, hence the same traps as listed under DIFFERENCE are possible.

Traps: same as DIFFERENCE.

EQUAL

This dyadic instruction returns one if its operands are equal, and zero otherwise. The result is always of integer type, but the operands may be any scalars (character, integer, or floating-point). The definition of equality used is as follows. A character scalar is equal only to another character scalar possessing the same character code. . . Two numbers are equal if their difference (as would be returned by the DIFFERENCE instruction) is zero.

Traps:

(8,1)-- Integer overflow if both operand are of integer type and the magnitude of their difference is not less than 2^{23}.

(8,2)-- Floating-point underflow if both operands are of numeric type and at least one operand is floating-point and the magnitude of their difference is less than 2^{-128} (but is greater than zero).

(8,3)-- Floating-point overflow if both operands are of numeric type and at least one operand is floating-point and the magnitude of their difference is not less than 2^{128}.

Dave

integer or integral floating-point

INDEX

This instruction corresponds to the indexing operation of APL\[illegible] restricted to scalar subscripts. The first operand is the array to be indexed; its rank must be equal to the value of the RANK field of the INDEX instruction descriptor (see Figure ?.?). The rest of the operands are the subscripts in left to right order. If S_i is the ith subscript and L_i is the length of the ith coordinate of the first operand, then $0 \leq (S_i - \mathrm{IORG}) < L_i$ must hold for $1 \leq i \leq \mathrm{RANK}$. The result returned by INDEX is the value of the ravel of its first operand indexed by

$$\sum_{i=1}^{\mathrm{RANK}} \left[(S_i - \mathrm{IORG}) \cdot \prod_{j=i+1}^{\mathrm{RANK}} L_j \right].$$

Traps:

(3,4)--Missing operand(s) if $\mathrm{LTOP} < (\mathrm{LBASE} + \mathrm{RANK} + 2)$.

(3,5)--Unknown descriptor type if the first operand is not a scalar or an array, or if any other operand is not a scalar or an array or the "undefined value".

(5,0)--Rank error if the first operand is a scalar, or if any other operand is an array or the "undefined value".

(9,1)--Wrong number of subscripts if RANK $\neq$ rank of first operand.

(7,0)--Type error if any subscript is a character.

(8,5)--Domain error if one of the subscript operands is a nonintegral floating point number.

(9,2)--Index error if $0 \leq (S_i - \mathrm{IORG}) < L_i$ fails to hold for some i, $1 \leq i \leq \mathrm{RANK}$.

UNDEFINED

This instruction takes no operands. Its sole effect is to place a descriptor for the undefined value on top of the stack.

Traps:
(1,1)-- Runtime stack overflow if room for one more cell on the stack doesn't exist, i.e. if SBASE+LTOP = address of last word in data segment.

EAT1

This instruction takes one operand, of any type, and produces no result. It is useful at the end of a statement to remove the value of the statement from the stack.

Traps:
(3,4)-- Missing operand if LTOP< LBASE+2.

INTERCHANGE

This instruction takes two operands, of any type, and returns two results by simply interchanging its operands on the top of the stack. The main use for INTERCHANGE is to reverse the operands to LESS to get the effect of a "GREATER" instruction.

Traps:
(3,4)-- Missing operand(s) if LTOP< LBASE+3.

SETORIGIN

This instruction takes one operand, produces one result, and has a side effect. The operand must be either zero or one (either integer or floating-point); the state flag IORG (see section 3.1) is set accordingly. The result of this instruction is the previous value of IORG, zero or one, and is always of integer type. As a special case the operand to SETORIGIN may be a suitable one-element array.

Traps:

(3,4)--Missing operand if LTOP<LBASE+2.
(3,5)--Unknown descriptor type if the operand is not a scalar or an array.
(6,0)--Length error if the operand is an array with other than one element.
(7,0)--Type error if (the element of) the operand is a character.
(8,5)--Domain error if (the element of) the operand is a number not equal to zero or one.

GETORIGIN

This instruction takes no operands; it returns as its single result the current value of the IORG state flag, zero or one. The result is always of integer type.

Traps:
(1,1)--Runtime stack overflow if room for one more cell on the stack is lacking.

SHAPE

This instruction corresponds exactly to the "monadic rho" operator of APL\360. Its single operand may be any scalar or array. The result is always an array of rank one whose length is equal to the rank of the operand; the integer type elements of this array are just the lengths of the coordinates of the operand. For example, SHAPE returns the empty vector when applied to any scalar.

Traps:
(3,4)--Missing operand if LTOP<LBASE+2.
(3,5)-- Unknown descriptor type if the operand is not a scalar or an array.
(1,2)-- Array block storage overflow if insufficient free space to create the result array exists.

RESHAPE

This instruction corresponds exactly to the "dyadic rho" operator of (special case: single subscript) APL\360. The first operand is a vector of non-negative integral numbers; the result is an array with these numbers as its shape and elements taken from the ravel of the second operand.

Traps:
(3,4)--Missing operand if LTOP<LBASE+3.
(3,5)--Unknown descriptor types if either operand is other than a scalar or an array.
(5,0)--Rank error if the first operand has rank larger than one.
(7,0)--Type error if (any element of) the first operand is a character.
(8,5)--Domain error if (any element of) the first operand is a number which is nonintegral, negative, or not less than 2^{16}, or if the product of the elements of the first operand is not less than 2^{16}.
(6,0)-- Length error if the second operand is an empty array but the shape for the result (as given by the first operand) is nonempty.
(1,2)--Array block storage overflow if insufficient free space exists to [illegible] the result array.

RAVEL

This instruction corresponds exactly to the "monadic comma" operator of APL\360. Its single operand may be scalar or array; the result is always a vector (rank one array) having the same number of elements as the operand has. The elements of the result are the same as the elements of the operand, taken in the standard APL ravel order.

Traps:

(3,4) -- Missing operand if LTOP<LBASE+2.

(3,5) -- Unknown descriptor type if the operand is not a scalar or an array.

(1,2) -- Array block storage overflow if insufficient free space exists to allocate the result array.

CATENATE

This instruction corresponds to the "dyadic comma" operator of APL restricted to operands of rank zero or one. The result is always a vector (rank one array) whose length is equal to the sum of the number of elements in both operands. The elements of the result are just the elements of the first operand followed by the elements of the second operand. Note that the elements of the operands for CATENATE may be any mixture of characters and numbers.

Traps:

(3,4) -- Missing operand(s) if LTOP< LBASE+3.

(3,5) -- Unknown descriptor type if either operand is other than a scalar or an array.

(5,0) -- Rank error if either operand has rank greater than one.

(6,0) -- Length error if the sum of the lengths of the operands is not less than 2^{16}.

(1,2) -- Array block storage overflow if insufficient free space exists for the result array.

INDEXGEN

This instruction corresponds exactly to the "monadic iota" operator of APL. Its operand should be a nonnegative integral number or one-element array; the result is a rank one array whose length is equal to the value of the operand. The elements of the result are consecutive integers beginning with the current index origin (i.e. the value of the state flag IORG).

Traps:

(3,4) -- Missing operand if LTOP<LBASE+2.

(3,5) -- Unknown descriptor type if the operand is not a scalar or an array.

(6,0) -- Length error if the operand is an array with length other than one.

(7,0) -- Type error if the operand (or its element if it is a one-element array) is a character.

(8,5) -- Domain error if the operand (or its element) is a number which is negative, nonintegral, or not less than 2^{16}.

(1,2) -- Array block storage overflow if sufficient free space is lacking.

CONSCALAR

This instruction is five bytes long (see Figure ?.?). The first byte is the opcode and the other four bytes (32 bits) constitute a descriptor for some scalar (character, integer, or floating-point) to be returned on top of the stack. (CONSCALAR takes no operands from the stack.)

Traps: (1,1)—— Runtime stack overflow if (SBASE + LTOP)=address of last word of data segment.

CONVEC

This is the only instruction which must be aligned with respect to the word boundaries in the code of a function block. Its format is as follows. (see Figure ?.?):

(1) The ETC-class opcode for CONVEC;
(2) A byte containing a nonnegative integer count (less than 256);
(3) Zero, one, two, or three bytes which are completely ignored to fill out the current code word;
(4) "Count" 32-bit descriptors (of any scalar type), one to a code word.

The effect of this instruction is to return a vector (rank one array) with these scalars as elements. (CONVEC takes no operands from the stack).

Traps: (1,2)-- Array block storage overflow if insufficient contiguous free space exists.
(1,1)-- Runtime stack overflow if (SBASE+ LTOP)= address of last word of data segment.

BRANCH

This instruction exactly corresponds to the "monadic right arrow"
operator of APL\360. The instruction returns no result value, but has
a side effect which depends on the value of its single operand:

(1) If the operand is a nonnegative integral number (integer or floating point)
not greater than the value of the current function block header's
NLINES field, then PCTR in the current
stack frame is set equal to the contents of the line table entry in the
current function block indexed by the value of the operand;

(2) If the operand is an integral number outside the range mentioned
in case (1) above, the processor returns from the current function
just as if it were executing a RETURN instruction (q.v.);

(3) If the operand is a nonempty vector the first element of which
satisfies case (1) or case (2) above, then the action specified for that
case is taken;

(4) Else if the operand is an empty vector, the processor takes no special action.

The BRANCH operator does not return a value in the usual sense that a call
to it could be nested within an expression (but of course in case (2) above
the current function may return a value).

Traps:
(3,4)--Missing operand if LTOP<LBASE+2.
(3,5)--Unknown descriptor type if the operand is not a scalar or an array.
(5,0)--Rank error if the the operand has rank greater than one.
(7,0)--Type error if the operand (or its first element if the operand is a vector) is
a character.
(2,5)--Domain error if the operand (or its first element) is not integral.
Also--also in case (2) above, any trap listed under RETURN.

GO

This instruction does not correspond to any construct of APL\360. It has no operands or results, but sets the contents of the PCTR field in the current stack frame equal to the byte address contained in the second two bytes of the three-byte GO instruction (see Figure ?.?.).

Traps:

(3,7)--Procedure segment address too large if the given byte address lies past the end of the current procedure segment.

GOTRUE, GOFALSE

These three-byte instructions each take a single operand, the value of which should be a numeric zero or one (either integer or floating-point). Execution of GOTRUE or GOFALSE is the same as for GO, except that setting PCTR is conditioned upon the value of the operand being one or zero, respectively. As a special case these instructions accept a suitable one-element array as operand; neither returns a result on the stack.

Traps:

(3,4)-- Missing operand if LTOP<LBASE+2.

(3,5)-- Unknown descriptor type if the operand is not a scalar or an array.

(6,0)-- Length error if the operand is an array with other than one element.

(7,0)-- Type error if the operand (or its solitary element) is a character.

(8,5)-- Domain error if the operand (or its element) is a number other than zero or one.

(3,7)-- Procedure segment address too large if PCTR is to be set but the given byte address lies past the end of the current procedure segment.

RETURN

This instruction of no operands causes the processor to return from the current function, just as would be caused by executing a BRANCH to a nonexistent line number. The return is accomplished by resetting the state variables LBASE and LTOP to the values which were in effect when the current function was called, and additionally pushing a result value on the stack if the RESULT flag in the header of the current function is one (see Figure 4.2). The new value for LBASE is taken from the "last LBASE" field in the stack frame for the returning function activation (see Figure 3.4). LTOP is set equal to the old value of LBASE minus the quantity (NARGS+NLOCALS+1), where NARGS and NLOCALS are fields in the header of the current function. The result value, if any, is taken from the first local variable (the one with address -1) of the returning function.

Traps:

(3,12)-- Temporary value(s) still on stack if LTOP>(LBASE+1).

(2,3)-- Return trace if both the global state flag FCNRTRC and the RTNTRACE flag in the current function block's header are equal to one.

(3,13)-- Attempt to return from bottom function activation in stack if value of "last LBASE" field in the current stack frame is not less than the current value of LBASE.

(10,0)-- Value error if the function is to return a result but its result variable contains a descriptor for the "undefined value".

CALL

Execution of this instruction causes the processor to call a function, i.e. to establish a new frame on top of the stack and make it the current one, updating LBASE and LTOP. The three-byte CALL instruction takes a variable number of operands on the stack, the actual parameters to the function being called. Rather than being removed by CALL, these operands become the formal parameter variables for the new stack frame. The other local variables of this frame are automatically initialized to the "undefined value". The "function descriptor" field in the new frame is set equal to the field of the same name in the CALL instruction itself (see Figure ?.?); the PCTR field in the new frame is always set to 8, which is the byte address of the first instruction in any function block. After the call is complete, the state variable LTOP will be equal to (LTOP+1). LBASE

A function is in call trace mode if both the global state flag FCNRTRC and the CALLTRACE flag in the header of the function are equal to one. As mentioned in section ?.?, execution of the first instruction after the call to a function in call trace mode always causes a call trace or (2,2) trap.

Traps:

(3,4)-- Missing operand(s) if (LBASE+NARGS+1)>LTOP, where NARGS is a field of the CALL instruction.

(1,1) -- Runtime stack overflow if (SBASE+LTOP+NLOCALS+2) > the last address in the data segment (NLOCALS is a field in the header of the called function block).

SHORTLOCAL, LONGLOCAL, LONGGLOBAL

Each of these instructions pushes a copy of the descriptor which is the value of a designated variable onto the runtime stack. The first two forms, SHORTLOCAL and LONGLOCAL, always designate a local variable (or formal parameter variable) of the current function activation; a LONGGLOBAL instruction always refers to a global variable. A local variable is addressed by a negative integer, which the processor interprets relative to the value of the state variable LBASE. As shown in Figure -.-, this negative integer, in two's complement representation and truncated to six or twelve bits, is explicitly contained in the instruction, SHORTLOCAL or LONGLOCAL respectively. A global variable is addressed by an integer larger than seven, which the processor interprets relative to the start of the data segment. (Addresses zero through seven are invalid since they correspond to the words of the processor state area). Each LONGGLOBAL instruction has a twelve-bit field holding the address of the variable to be affected.

As an aid in implementing call-by-reference function arguments, one level of indirect addressing is possible. If the variable addressed by a SHORTLOCAL, LONGLOCAL, or LONGGLOBAL instruction contains an indirect parameter word descriptor (see Figure -.-), then the instruction loads the value of the variable indirectly addressed by this descriptor. Indirect parameter word descriptors are produced by the REFERENCE instruction (q.v.).

Traps:

(3,6)-- Illegal indirect chain if an indirectly addressed variable contains another indirect parameter word descriptor.

(10,0)-- Value error if the variable directly or indirectly addressed by a SHORTLOCAL, LONGLOCAL, or LONGGLOBAL instruction contains a descriptor for the "undefined value".

(1,1)-- Runtime stack overflow if SBASE+LTOP = address of last allocated data segment.

The MEMREF family.

These instructions share a common format; the execution of each involves a variable designated by a field of the instruction itself. As shown in Figure -.-, a MEMREF-family instruction is two or three bytes long. The first byte contains the opcode for the MEMREF family together with a subopcode selecting a member of the family. The last byte or pair of bytes addresses the variable to be affected; this address takes a form identical to the SHORTLOCAL, LONGLOCAL, or LONGGLOBAL instruction which would be used to load the value of the same variable on to the stack.

One level of indirect addressing is provided by the MEMREF instructions, just as with the SHORTLOCAL, LONGLOCAL, and LONGGLOBAL instructions. If the variable directly addressed by a MEMREF instruction contains an indirect parameter word descriptor, then the instruction (ASSIGN, etc.) applies to the variable indirectly addressed by this descriptor.

Traps from MEMREF instructions (see also the descriptions of the individual instructions):

(3,7) -- Malformed address field.

(3,6) -- Illegal indirect chain if an indirectly addressed variable contains another indirect parameter word descriptor.

(3,8) -- Undefined suboperation code.

ASSIGN_INDEXED

The first sixteen of the thirty-two possible MEMREF suboperation codes all stand for the ASSIGN_INDEXED operation, which is compiled for the APL assignment operator when the variable on the left-hand side is subscripted. Let RANK be the value of the suboperation code of the instruction, $1 \leq RANK \leq 15$. Then the instruction takes (RANK+1) operands. The first RANK operands are the subscripts, which are limited to scalars. The last operand is the value computed by the code compiled from the right-hand side of the assignment, and must also be scalar. The variable addressed by the ASSIGN-INDEXED instruction must hold a descriptor for an array of rank RANK. If S_i is the value of the ith subscript operand and L_i is the length of the ith coordinate of this array, then $0 \leq (S_i - IORG) < L_i$ must hold for $1 \leq i \leq RANK$. The execution of this instruction substitutes the value of its last operand for the element in the array indexed by

$$\sum_{i=1}^{RANK} \left[(S_i - IORG) \cdot \prod_{j=i+1}^{RANK} L_j \right] .$$

The new value of this array is also returned on top of the stack.

Traps:

(3,4)-- Missing operand(s) if LTOP<(LBASE+RANK+2).

(3,5)-- Unknown description type if the value of the addressed variable or of any of the operands is not a scalar or an array.

(5,0)-- Rank error if the value of the addressed variable is a scalar, or if any of the operands is an array.

(9,1)-- Wrong number of subscripts if RANK≠ rank of the array which is the value of the addressed variable.

(7,0)-- Type error if any of the subscripts (first RANK operands) is a character.

(8,5)-- Domain error if any of the subscripts is a nonintegral floating-point number.

(9,2)--Index error if $0 \leq (S_i - \text{IORG}) < L_i$ fails to hold for some i, $1 \leq i \leq \text{RANK}$.

ASSIGN, ASSIGN_NORESULT

These two instructions perform assignment to an unsubscripted variable. Each takes a single operand from the stack, the value to be assigned. Execution of an assignment stores this value into the variable directly or indirectly addressed by the instruction (see The MEMREF instructions). ASSIGN also returns on top of the stack the value assigned to the variable. ASSIGN_NORESULT, as its name implies, returns no value. It is equivalent to an ASSIGN instruction followed by an EAT1 instruction, and could be compiled for an assignment which is not nested in a larger expression.

Traps:

(3,4) -- Missing operand if LTOP<(LBASE+2).

REFERENCE

This instruction returns on the stack an indirect parameter word descriptor for the variable it directly or indirectly addresses. It is compiled for a variable reference used as an actual parameter to a user-defined function specifying call-by-reference for the corresponding formal parameter. → The indirect parameter word descriptor (see Figure -.-) contains one bit to designate a local or global variable, and a sixteen bit address field. This address is just a data segment address for a global variable, and is an SBASE-relative address for a local variable. REFERENCE takes no operands from the stack.

Traps:

(3,1)-- Runtime stack overflow if (SBASE+LTOP) = address of last word in data segment.

TEST_DEFINED

This instruction returns an integer type value of zero or one depending on whether the variable it directly or indirectly addresses contains or does not contain a descriptor for the "undefined value". TEST_DEFINED takes no operands from the stack.

Traps:

(3,1)-- Runtime stack overflow if (SBASE+LTOP) = address of last word in data segment.

NO_OPERATION

This instruction takes no operands, returns no results, and has no side effects. It could be used, for example, to patch over an instruction used for debugging that is no longer needed.

Traps: None.

BREAKPOINT

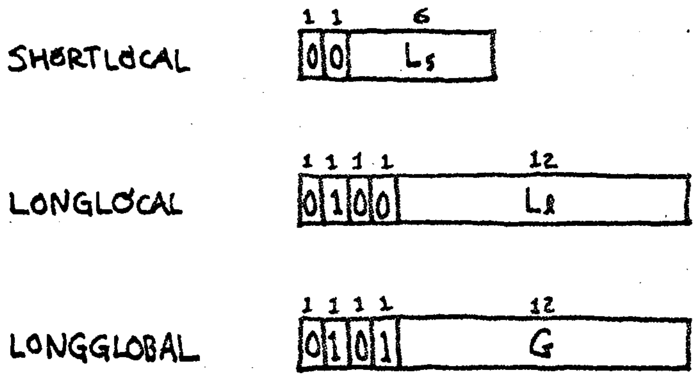
This instruction takes no operands, returns no results, and in fact never finishes execution; its sole effect is to cause a "breakpoint encountered" (2,4) trap. Note that the PCTR field (in the current stack frame) is left pointing to the BREAKPOINT instruction; to enable the program to continue past the breakpoint, the PCTR must be increased by one (the length of this instruction) before the program is restarted.

Traps:
(2,4) -- Breakpoint trap (unconditional).

ATTENTION

This instruction is similar to BREAKPOINT (q.v.): it takes no operands, returns no result, and always generates a trap. In this case the trap is an "attention" (4,0) trap, and is intended as part of an escape mechanism so that APL programs can call functions running on the SIMPLE processor.

Traps:
(4,0) -- Attention trap (unconditional).

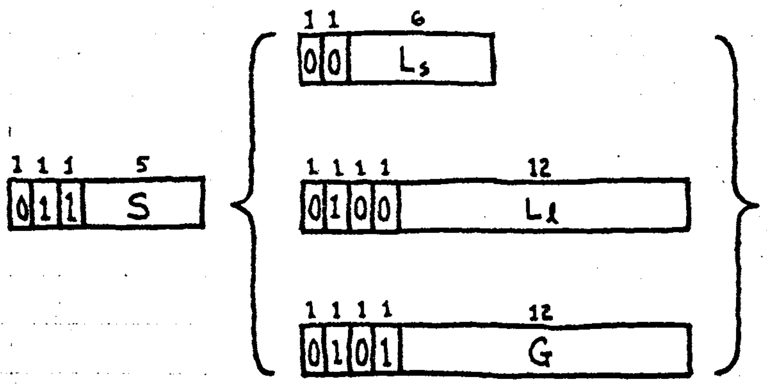


L_s and L_ℓ are local addresses: negative integers in two's complement notation, truncated to six and twelve bits, respectively. Thus $-64 \leq L_s \leq -1$ and $-4096 \leq L_\ell \leq -1$ are satisfied.

G is a global address: a twelve-bit unsigned integer, which should be larger than seven.

SHORTLOCAL, LONGLOCAL, and LONGGLOBAL Instructions

Figure



S is a suboperation code:

ASSIGN_INDEXED, rank S,	if $1 \le S \le 15$
ASSIGN	if $S=16$
ASSIGN_NORESULT	if $S=17$
REFERENCE	if $S=18$
TEST_DEFINED	if $S=19$

L_s, L_ℓ, and G are local or global addresses, as defined in Figure (SHORTLOCAL, LONGLOCAL, and LONGGLOBAL Instructions).

The MEMREF Family

Figure

1	1	1	1	4
1	0	0	0	S

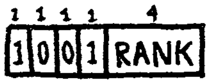
S is a suboperation code:

	S	operator
monadic	0	IDENTITY
	1	NEGATIVE
	2	FLOOR
	3	CEILING
	4	MAGNITUDE
	5	NOT
	6	TEST_NUMBER *
	7	CONVERT *
dyadic	8	SUM
	9	DIFFERENCE
	10	PRODUCT
	11	QUOTIENT
	12	LOGICAL_PRODUCT
	13	LOGICAL_SUM
	14	LESS
	15	EQUAL *

* These operators accept character operands.

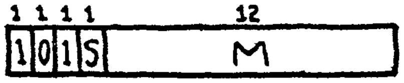
The GENSCALAR Family

Figure



The INDEX Instruction

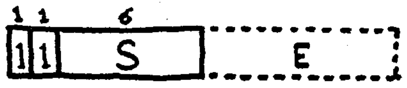
Figure



S is the sign (1⇒ negative, 0⇒ nonnegative).
M is the magnitude, an unsigned integer.

The SHORT_CONSTANT Instruction

Figure

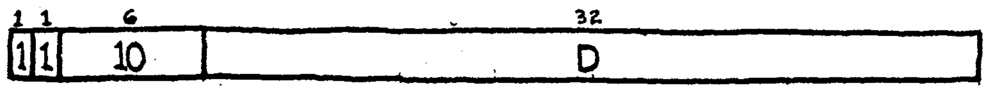


S is a suboperation code, and E is extra information present for some values of S:

S	operator	E present ?
0	UNDEFINED	No
1	EAT1	No
2	INTERCHANGE	No
3	SETORIGIN	No
4	GETORIGIN	No
5	SHAPE	No
6	RESHAPE	No
7	RAVEL	No
8	CATENATE	No
9	INDEX_GENERATOR	No
10	CONSTANT_SCALAR	Yes, see Figure
11	CONSTANT_VECTOR	Yes, see Figure
12	BRANCH	No
13	GO	Yes, see Figure
14	GO_TRUE	Yes, see Figure
15	GO_FALSE	Yes, see Figure
16	RETURN	No
17	CALL	Yes, see Figure
18	NO_OPERATION	No
19	BREAKPOINT	No
20	ATTENTION	No

The ETC Family

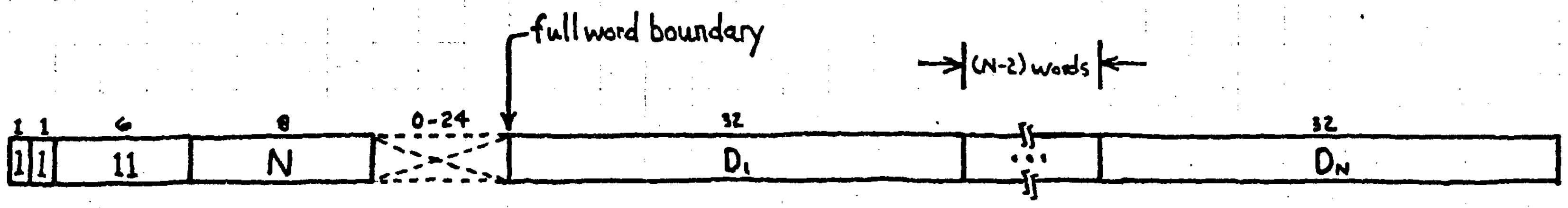
Figure



D is any scalar descriptor (see Figure   ).

The CONSTANT_SCALAR Instruction.

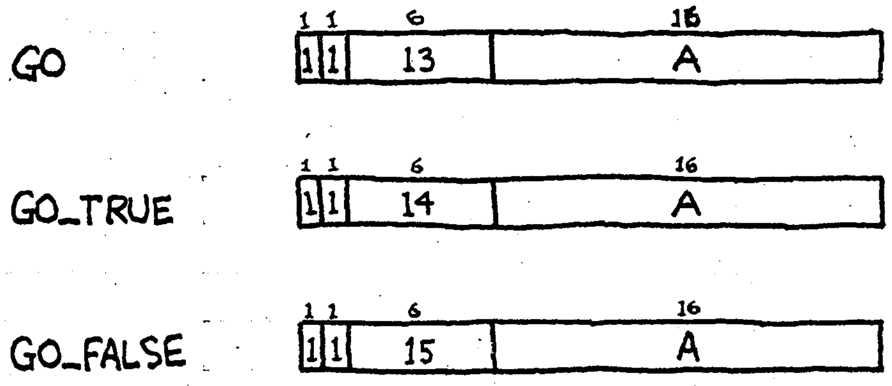
Figure



N is a count, $0 \leq N \leq 255$.
D_i is any scalar descriptor, for $1 \leq i \leq N$.

The CONSTANT_VECTOR Instruction

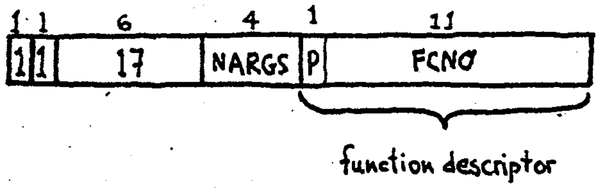
Figure



A is a byte address in the current function block of an instruction to transfer to.

GO, GO_TRUE, and GO_FALSE Instructions

Figure



NARGS is the number of arguments the caller has pushed onto the stack before executing this instruction.

P selects the procedure segment, regular or alternate, containing the function being called.

FCNO is the function number of the function being called.

The CALL Instruction

Figure

1 of 2

→· Appendix ?: Summary of Trap Classes and Numbers

Class	Number	Description
1		Storage overflow:
	1	Runtime stack overflow
	2	Array block storage overflow
2		Debug trap:
	1	Instruction completed in step mode
	2	Function call in call-trace mode
	3	Function return in return-trace mode
	4	Breakpoint encountered
3		System error (processor and/or compiler):
	1	Procedure segment address too large
	2	Stack address too large
	3	Global variable or array block address too large
	4	Missing operand(s) for instruction.
	5	Unknown descriptor type for instruction operand.
	6	Double indirection encountered
	7	Malformed MEMREF-family instruction
	8	Undefined MEMREF sub-operation
	9	Undefined GENSCALAR sub-operation
	10	Processor state OPNO field too large on interrupt restart
	11	Undefined ETC suboperation

Class	Number	Description
(3)	12	Temporary value(s) still on stack upon function return
	13	Attempt to return from bottom function activation in stack
4		Attention trap
5		APL rank error
6		APL length error
7		Type error—character operand(s) where numeric expected
8		Domain error:
	1	Integer overflow
	2	Floating-point exponent underflow
	3	Floating-point exponent overflow
	4	Division by zero
	5	Other domain error:
9		Index error:
	1	Wrong number of subscripts
	2	Subscript too large or small (APL index error)
10		APL value error

13 December 1972
Paul McJones

Appendix

The APL Processor As an I/O Device

As the above title suggests, the APL processor appears as an i/o device to the SIMPLE processor. Five preassigned cells in central memory and the two external communications flipflops are associated with the APL processor "device." The flipflops will be referred to as FF_{SA} and FF_{AS}, and have the following properties:

(i) The APL processor can read (sense state = ∅ or = 1) and reset (set state to ∅) FF_{SA}, while it can set FF_{AS} (to state 1);

(ii) The SIMPLE processor can set FF_{SA}, and can read and reset FF_{AS}.

Here are the preassigned memory cells and their usage (see "System Parameters" document by Charles Grant for the actual value of these symbols):

(i) APLCOMWD - the SIMPLE processor places a command (either "start" or "stop") here for the APL processor;

(ii) APLSTATWD - the APL processor places a status code here whenever it stops;

(iii) APLDSEGWD, APLPSEG∅WD, APLPSEG1WD - before the SIMPLE processor commands the APL processor to start, it sets up these words to describe the data segment, procedure segment, and alternate procedure segment, respectively, of the program to be run. Figure 2 gives the format of these "segment descriptors."

The APL processor is always in one of two states, <u>idle</u> or <u>running</u> a program--see figure 1. A transition from idle to running occurs only when the SIMPLE processor gives a start command. A transition from running to idle occurs either when the SIMPLE processor gives a stop command or spontaneously when the APL processor encounters a trap condition in the program being executed, whichever happens first. These start and stop commands are given in the following way: First, the SIMPLE processor sets the low order (rightmost) bit of APLCOMWD: zero means stop; one means start. Then, the processor sets FF_{SA} . Some time later the APL processor will notice that FF_{SA} is set, will reset it, and will obey the command indicated by the current setting of APLCOMWD.

Whenever the APL machine goes idle, it issues a status code. These codes are issued with a method like that used for giving start/stop commands. The APL processor stores the status code in APLSTATWD and sets FF_{AS} . Eventually the SIMPLE processor will notice that FF_{AS} is set, reset it, and then look at the current value of APLSTATWD. Figure 3 lists the status codes.

Some notes on this interface to the APL processor are in order. First of all, when the processor is running and is given a stop command, it could take as long as a few milliseconds before the shutdown occurs and the "ok, stopped" status is issued. (Sometimes the APL processor really gets wrapped up in its work!) Second, the SIMPLE processor should be sure the APL processor is idle before it writes into the segment descriptors APLDSEGWD, APLPSEGØWD, and APLPSEG1WD. This is because the APL processor looks at these cells from time to time as long as it is running.

⟨commandgiven⟩/⟨status returned⟩

$$\begin{tikzcd}[column sep=huge]
\text{IDLE} \arrow[loop above, "\text{"stop"/none returned}"] \arrow[r, "\text{"start"/none returned}"] & \text{RUNNING} \arrow[l, bend left=20, "\text{"start"/"whoops"}"] \arrow[l, bend left=50, "\text{"stop"/"ok, stopped"}"] \arrow[l, bend left=80, "\text{none given/"trap"}"]
\end{tikzcd}$$

Figure 1

segment descriptor:

16	16
base	length

Note: base is the absolute central memory addess of the first word of the segment.

Figure 2

Commands (stored in APLCOMWD by the SIMPLE processor):

$xx \cdots x\emptyset_2$ = stop
$xx \cdots x1_2$ = start

Status codes (stored in APLSTATWD by the APL processor):

1 = "whoops" (start command received while running)
2 = "ok, stopped"
3 = "trap" (APL processor encountered a trap condition in program being run)

Figure 3