IBM PALO ALTO SCIENTIFIC CENTER TECHNICAL REPORT NO. ZZ20-6428

FEBRUARY 1975

THE APL ASSIST

(RPQ S00256)

A. HASSITT

L. E. LYON

IBM SCIENTIFIC CENTER
2670 HANOVER STREET
PALO ALTO, CA 94304

[redacted]

This report describes the APL Assist, RPQ S00256, for the IBM System/370 Model 145. This RPQ was announced on May 13, 1974 (blue-letter P74-22) and was made available on September 30, 1974 (P74-50).

This document obsoletes and supersedes the earlier IBM Confidential P.A.S.C. Technical Report Number ZZ20-6417 by the same authors (An APL Emulator, July 1972).

ABSTRACT

The APL Assist is a hardware feature which enhances the performance of APL systems by providing direct execution of a major subset of the APL language. The feature can be installed on an IBM/370 model 145. The feature implements a new IBM/370 instruction called APLEC. The APL Assist does not modify any other IBM/370 instruction. The assist feature and the APLEC instruction may be used under standard operating systems such as VS and VM/370. APL execution is initiated by loading a general purpose register with the base address of an APL workspace and then issuing the APLEC instruction. This report defines the format of the APL workspace required by the assist feature, it gives the form of the APLEC instruction and it describes the results to be expected from using the instruction.

INDEX TERMS FOR THE IBM SUBJECT INDEX

Microprogramming
Machine Language
APL
Performance
IBM System 370/145
07 - Computers
21 - Programming

CONTENTS

FIGURES

SECTION I: INTRODUCTION AND BACKGROUND

INTRODUCTION

The APL systems which are currently in use provide interpretive execution of the APL language. Interpretive execution offers many advantages in producing a powerful, safe and elegant programming language, however, interpretation is typically much slower than direct execution. There are several aspects of the APL language which make it impossible to provide direct execution of APL using the machine language of existing computers. The only way of directly executing APL programs is to provide hardware specifically designed for that purpose.

The effective utilization of any hardware requires that it be supported by appropriate software. The APL Assist feature is installed on an IBM/370 model 145 and it is invoked by a special IBM/370 instruction. The feature can provide direct execution of a major portion of the APL language and it can be invoked from IBM/370 software. The feature supplies an interface so that the remainder of the APL system can be implemented in software routines supplemented by the services of a standard operating system.

The first section of this report gives an introduction to various aspects of APL execution, and it gives an overview of topics which will be discussed in detail in subsequent sections. The second section gives a definition of the workspace format which is required for use of the assist feature. The third section describes the APLEC instruction and the way in which it interacts with the operating system and with the APL system.

EXECUTION AND MACHINE LANGUAGE

Consider the running of a conventional assembler language program. The program is written in assembler language, for example:

```
L    1,R
A    1,Q
ST   1,P
```

The program is entered into the machine and is processed by an assembler. The assembler converts the mnemonic instructions such as 'L' into machine language instructions such as hexadecimal '58'. It also assigns memory locations to variables such as R. The ouput of the assembler is sent to the loader which puts the binary instructions into memory, and which resolves any cross references between external symbols. The result is a machine language program which might, for example, be:

58102400 5A102404 50102408 hexadecimal

The loader now initiates the execution of the user's program by branching to the first executable instruction. When the user's program has finished execution, or when certain errors occur, control is returned to the operating system.

As we can see from the above example, the execution of an assembler language program requires the use of an assembler, a loader, an operating system, and of course a machine which can execute IBM/370 machine language instructions. The execution of an APL program requires some analogous features. An APL system contains a translator, a supervisor and a mechanism for executing the translated form of the APL program. The supervisor may control the operation of the complete machine, or more typically, the supervisor controls the operation of the APL sub-system which is itself under the control an operating system such as VS or VM/370. An APL program is, of course, written in APL, for example:

$P \leftarrow Q + R$

The program is entered into the machine and is processed by the APL translator. The translator converts symbols such as '+' into internal codes such as 1021 hexadecimal. It converts names such as R into internal names. Internal codes and internal names are discussed in detail later; for the moment it is sufficient to know that internal names are 16 bit (4 hexadecimal digit) integers which are multiples of four. The internal form of the above APL statement might be:

0108 1021 0104 7001 0100 A001

where 0100, 0104, 0108 are the internal names of P, Q, and R respectively. 1021, 7001 and A001 are the internal codes for '+', '←' and 'end of statement'. The translator puts the internal code directly into memory. Notice that the tranlator reverses the order of the items within a statement since this facilitates execution. The translator stores the address of the first byte of the internal form in a memory location called NEXTINST and it initiates APL execution. In a conventional APL system, APL execution is initiated by branching to the APL interpreter. The APL interpreter is an IBM/370 program which does interpretive execution of the internal form of the APL program. On a machine which has the APL Assist feature installed, APL execution is initiated by issuing the APLEC instruction. The APL Assist feature gets the address from NEXTINST, finds the first byte of the APL code string, and directly executes the internal form of the APL statement.

It should be noted that the APL translator is similar to an assembler, it is not a compiler. The major part of the translation process is a one for one substitution of internal names, constants and codes for external APL names, constants and operators or primitive functions. The translation for Q+R, for example, is independent of the properties of Q and R. The translated form is the same, irrespective of whether Q and R are scalars, vectors, arrays, global variables, local variables, shared variables, or defined functions. The reversal of items within an APL statement is similar to the way in which the IBM/370 assembler re-arranges the fields in a 'TM' instruction.

We have made an analogy between the IBM/370 assembler and the APL translator. It will simplify the description of the APL Assist feature if we continue the analogy between the execution of IBM/370 and APL programs. The output of the assembler is an IBM/370 machine language program. We can regard the output of the APL translator as APL machine language. An IBM/370 system contains hardware which can execute IBM/370 machine language instructions. The major function of the APL Assist feature is execution of APL machine language instructions. The PSW contains the address of the next IBM/370 machine language instruction. The location NEXTINST contains the address of the next half-word of APL machine language. An IBM/370 with the assist feature installed can execute IBM/370 or APL machine language instructions; the APLEC instruction causes the machine to switch from IBM/370 to APL mode. There is no explicit instruction for switching from APL to IBM/370 mode, but as we shall see, there is an automatic return to IBM/370 mode if the APL program terminates, if a page fault occurs, or if an interrupt is pending.

The APL Assist feature does have several minor functions, in addition to its main function as an emulator; details are given in section III. We will usually be concerned with the execution of APL statements so we will often use the term 'APL emulator' in place of 'APL Assist'.

THE ENVIRONMENT

The execution of IBM/370 programs is determined by the particular instructions in the program and by the current environment. The environment is specified by the PSW, the 370 registers (both general purpose registers and floating point registers) and by the contents of memory. In the execution of any instruction only a small part of the environment is used. For example, the result of the next operation might be determined by the fact that the instruction address is 1234, the contents of location 1234 is 58102400, and register 2 contains 62350. In this environment, the machine will put the contents of location 62750 into register 1.

The environment of the APL emulator is the APL workspace plus the general purpose and floating point registers. The workspace is a contiguous piece of memory which contains APL programs, APL data and status information. The system may contain many different workspaces, however only one workspace is active at any one time. The address of the active workspace is specified by the contents of the WORKBASE register; WORKBASE is in fact general purpose register 3. The execution of a typical IBM/370 instruction utilizes and modifies a small part of the environment. The execution of a typical APL expression may use and modify a significant part of the workspace.

The IBM/370 machine requires a particular format for machine language instructions and for a limited number of data items such as integers, floating point numbers and decimal digits. It does not utilize any specific format for collections of items; it does not, for example, specify the ordering of the elements of an array. APL is a high level language and the emulator directly executes the statements of the language. The APL emulator requires a particular format for instructions and data items, and also for statements, defined functions, scalars, vectors, arrays and status information. The emulator relies on this format when carrying out such operations as statement scan, syntax analysis, function call and return, subscripting of arrays and dynamic allocation of memory.

The workspace is divided into four parts, namely

Control words
Address table
Stack
Free space

The control words area contains certain fixed constants as well as current status information. The address table contains a one word entry corresponding to each internal name; it also contains some empty words which will be used when new names are created. The address table entry at location WORKBASE+r is a word which describes the properties of the variable whose internal name is r; thus if WORKBASE contains 123400 and variable R has internal name '0108' then the word at location 123508 is the address table entry for R. An address table entry has one of two forms:

```
+-----------------------+
| S P | D |      V      |
+-----------------------+

+-----------------------+
| S P |        A        |
+-----------------------+
```

S specifies whether the entry is a variable, a function, a group name, or a shared variable. P specifies which form of the address table entry is being used. It also specifies whether a variable has a value or not. If a variable has a value the value may be specified by D and V or the value may be specified by the block of memory beginning at location A. The first form is used for scalars which are character, logical or small integers.

Free space contains the current values of variables and functions as well as some unused space. The address 'A' in the previous paragraph is a free space address. If R is an array, its address table entry will point to a block of the form:

```
+--------------------------------------+
|  D  :  r  |           V              |
+--------------------------------------+
```

where D is a sixteen bit descriptor specifying that R is an array and indicating whether it is logical, integer, real or character. The half word shown as r contains the internal name of R. V contains the internal representation of: the ravel of R, the size of R, the rank of R, and the size of the ravel of R (see the APL\360 User's Manual <3> for the meaning of size and ravel). Later sections of this manual provide further details on the representation of functions and variables.

The stack is used to hold temporary values or names during the execution of a statement, and it is used to keep a record of function calls and the global values of local variables. The APL system commands)SI and)SIV give a display of most of the information on the stack.

APL EXECUTION

This report describes how the APL emulator may be used; it does not describe the inner working of the emulator. It might, however, be helpful if we give some details of the execution of a particular statement. Consider the APL statement 'P←Q+R' which was discussed above. The internal form of the statement was:

0108 1021 0104 7001 0100 A001

The emulator obtains the first two bytes (0108) and examines the last two bits; in this case these are 00 which indicates an 'internal name'. The emulator forms WORKBASE+0108 and finds the appropriate S bits (see 'THE ENVIRONMENT'). Assuming that the S bits show that 0108 is a variable and not a function, the emulator notes this fact and selects the next item (1021). The low order bits of '1021' indicate that it is an operator. The emulator now selects the 0104, finds its S bits, and, assuming that 0104 is a variable, the emulator starts to perform the addition of variable 0108 and variable 0104. The first action is to examine the P bits of 0108 and check that the variable has a value (if not, to signal 'value' error), then check that it is numeric (if it is character then signal 'domain' error). Similar actions are performed for '0104'. Next the emulator checks to see if 0104 and 0108 are scalars, vectors, or arrays, and that their sizes conform. It then decides on the type of the result (integer or real), obtains space for the result, does the additions, checks for 'range' errors (exponent overflow), stores the result (which may be a scalar, vector, or array) and finally proceeds to the next item in the statement, which is the '7001'.

The execution of this expression has been described in some detail in order to demonstrate that the APL emulator does execute APL statements directly and is fully cognizant of the properties of the APL language.

THE APL SYSTEM

We continually reference 'the APL system'. The APL Assist feature does not presuppose any particular APL system; the system designer has almost complete freedom in choosing the facilities which the APL user will see. The completeness and reliability of the feature has been established in its extensive use by the APL/CMS system (program number 5799-ALK, PRPQ MF2608). Details of APL/CMS are given in the APL/CMS User's Manual <4> and the APL/CMS Installation Manual <5>. We will outline the relationship between the assist feature and a typical APL system.

An APL system will usually contain a supervisor, a translator, an interpreter, a shared variable processor and auxiliary processors. The supervisor controls the terminal input/output, disk I/O, scheduling of users, management of APL libraries, and so on. The APL Assist feature is designed to improve the performance of APL execution. It has a major impact on the interpreter, in fact it can replace a large part of the interpreter. It has a minor effect on the translator and very little effect on the rest of the system. The translator must of course be designed to support the workspace organization specified in this report, but it makes little use of the assist feature itself.

The APLEC instruction is indeed an IBM/370 instruction, it is interruptable, it does check the protect key before storing into memory, it does not initiate I/O operations and it does not make any unusual assumptions about the behavior of the operating system. The APLEC instruction may be used in a multi-programming environment, and it may be used in a virtual machine running under VM/370.

The APL Assist feature is implemented as a microprogram designed to operate in the control store of the IBM/370 model 145. The control store is a valuable resource and any use of the store must be considered carefully. On the one hand, the implementation of a feature in microcode may improve performance. On the other hand it may reduce the number of other features which can be added and it may reduce the amount of real program memory. APL is a very powerful and extensive language with primitive functions ranging from '+' (integer or real addition of scalars, vectors or arrays) to '⌹' (matrix division). It would be uneconomical and impractical to implement the whole of the language in microcode. We have chosen to implement a major subset of the language in microcode. The remaining functions may be implemented in IBM/370 or APL machine language. It was decided, for example, that there would be little performance advantage to putting matrix division in microcode. If the emulator is required to execute a statement such as:

```
A←(B+C-D)⌹E
```

The emulator evaluates the expression B+C-D and gives it a temporary name such as T. T may be a scalar, a vector or an array. The emulator checks that E has a value. It puts the internal names of T and E into general purpose registers, reverts to IBM/370 mode and branches to part of the APL system. The APL system may now compute the value of T ⌹ E, put the name of the result in a general purpose register and execute an APLEC. Alternatively, the APL system may put the name of an APL function into the register and execute the APLEC. In either case the emulator will continue APL execution. In one case it assigns the result into A. In the other case it calls the named APL function and assigns its result to A.

THE APL ASSIST RPQ

The primary purpose of the APL Assist RPQ is to provide direct execution of APL programs. The feature is intended to be used by an APL system which is written in (or has been assembled or compiled into) IBM/370 machine language. The APL system must prepare the workspace in the correct format and load the registers with appropriate information before using the APLEC instruction. The macro defintions in a later section of this report provide a convenient way of using APLEC. There are a number of different uses of the APLEC; they may be grouped into four categories. The APLEC may be used to initialize or check that the feature is loaded; the appropriate macro is APLCSL. The APLEC may be used for service functions such as getting or freeing space, etc.; the macros are APLFIND, APLFREE, etc. The APLEC may be used to return control to the emulator after an interrupt, quantum end or use of an external function; the macros are APLRESM, APLSRTN or APLRTN. The APLEC may be used to execute APL programs, in which case the macro is APLSCAN. At this point we are mainly concerned with the use of APLSCAN.

Assumming that the workspace is in the correct format and the registers have been loaded correctly, the use of APLSCAN will cause the assist feature to begin execution of the APL statement to which NEXTINST points. There are several ways in which the assist feature may relinquish control so we will give a number of examples. Assume that NEXTINST points to the internal form of the statement:

$$P \leftarrow Q + R$$

Assume that the APLEC is at location X and that general purpose register 3 contains W (the workspace base). Each example begins by stating the properties of P,Q and R which it will assume.

a) P, Q and R are variables. Q and R have numeric values. Execution proceeds to completion. The next IBM/370 instruction to be executed is at location X+4. The APLEC sets the condition code to zero. P will have the value Q+R, which is to say that the address table entry corresponding to P will have the value or a pointer to the value specified by Q+R. NEXTINST will have been updated to reflect the new status.

b) Same as (a) except that Q has no value. The next IBM/370 instruction to be obeyed will be at location X+4. The condition code will be one to indicate that an APL error has occurred. There will be a code in general purpose register 5 which indicates a 'value' error. NEXTINST will point to the half-word past Q.

c) Same as (a) except that a page fault occurs. The assist feature sets the PSW to point to a certain word in the workspace, it switches to IBM/370 mode and a storage access exception occurs. The operating system follows its normal course, it eventually loads the page and executes the word specified by the PSW; that word is an APLRESM which returns control to the assist feature. APL execution is resumed as though the page fault had not occurred. This process is transparent to the APL system.

d) Same as (a) except that Q is an APL function. The assist feature will do the call of Q according to the rules of APL. Q may call other APL functions. This is similar to case (a).

e) Same as (d) except that Q uses ⌹ (the matrix division function). The assist proceeds as far as the matrix divide, it then reverts to IBM/370 mode with the next instruction to be taken from location (T+contents of CALL370F). CALL370F is a control word at location W-X'90'. T is an offset whose value is given in section III of this report. The IBM/370 instruction should branch to a routine which does the matrix division or supplies the name of an APL routine which can do the matrix division. In either case the routine uses an APLRTN to return control to the assist feature. The value of X (the location of the APLSCAN macro call) has been preserved so the instruction at location X+4 will eventually be reached.

The APL Assist feature executes a major subset of the APL language. The language is described in the APL\360 User's Manual <3>, modified and supplemented by the APL/CMS User's Manual <4>. The 'EXTERNAL FUNCTIONS' section of this report delineates which APL functions are implemented in the APL Assist and which should be handled by the APL system.

SECTION II: FORMAT OF THE WORKSPACE

THE WORKSPACE

A computing machine works in an environment and the execution of the machine causes the environment to change. In an IBM/370 operating in non-privileged mode the environment is essentially the PSW, the registers (16 fixed point and 4 floating point) and a piece of the main memory. The non-privileged program can not change anything outside this environment but it can make a supervisor call in order to get information into and out of its environment. In the APL emulator the environment is the workspace plus the 370 registers. One of the registers specifies the location of the workspace. The other registers and the contents of the workspace specify the current status of APL execution. The APL emulator has no memory of its own; it simply operates on a workspace in the manner specified by the status and by the programs in that workspace. When the APL emulator gives up control (for example in order to allow an interrupt to be serviced) it does not assume that when it regains control it will still be operating on the same workspace.

The main areas of the workspace are shown in figure 1. The system areas are used only by the APL system and are not discussed in this document (with minor exceptions). The other areas are summarized below and discussed in detail in the following sections. The 370 registers are also discussed in a following section.

Free space contains the values of APL variables, APL functions and a block of unused space. The address table contains a complete description of variables which have no value and of some scalar variables. For other variables and for all functions, the address table contains a partial description and an address. The address points to a block in free space. The execution stack, or simply the stack, is a pushdown list used by the APL emulator. The control words contain status information, constants, save areas, and so on. Some control words have the same format as address

```
+-------------------------------+
|                               |   <---  LOW MEMORY ADDRESS
| SYSTEM AREA A                 |
|                               |
|-------------------------------|
|                               |
| CONTROL WORDS                 |
|                               |
|-------------------------------|
|                               |
| ADDRESS TABLE                 |
|       |                       |
|       |                       |
|       V                       |
|                               |
| - - - - - - - - - - - - - -   |
|                               |
|       A                       |
|       |                       |
|       |                       |
| (EXECUTION) STACK             |
|                               |
|-------------------------------|
|                               |
| FREE SPACE (BOTTOM)           |
|       |                       |
|       |                       |
|       V                       |
|                               |
|       A                       |
|       |                       |
|       |                       |
| FREE SPACE (TOP)              |
|                               |
|-------------------------------|
|                               |
| SYSTEM AREA B                 |
|                               |   <---  HIGH MEMORY ADDRESS
+-------------------------------+
```

FIGURE 1: WORKSPACE FORMAT

table entries, so there is an overlap between the end of the control words and the beginning of the address table.

Free space is used from both the bottom and top as shown. When there is insufficient space left, the emulator invokes a software routine which performs a garbage collection to reclaim any unused blocks. The stack and the address table both grow towards a definite boundary between them. Should one of them require additional space, however, the boundary may be dynamically moved. Note especially that the stack grows from high memory addresses to low memory addresses. When we speak of the top item on the stack we refer to the item most recently placed on the stack. Thus the top stack item is the stack item with the lowest memory address.

THE CONTROL WORDS

The control words contain constants, addresses, and so on, which help specify the current status of the workspace. The only things not included which are necessary to completely describe the status of a workspace are contained in the registers (see 'GETV' in section III). A map of the control words is given in figure 3; figure 2 gives a listing of the codes used in the map. Below is an alphabetic list of the control words and their definitions. The emulator instructions can conveniently use only small displacements (<256). These can, however, be either positive or negative and thus GPR3 is used to point not to the beginning of the workspace, but higher up (at TMPSAV). In the codes CBYT refers to the control byte which is the first byte of the word. In the definitions the phrase 'Address table entry for ...' means that the item is in free space (or in the system or is an immediate) and the control word follows the conventions described in 'THE ADDRESS TABLE'. The control word is thus like a reserved name for a variable which will be used by the emulator or the system. In a paging system all the control words must reside within a single page.

RELO	A	Absolute value - no base required
	D	Displacement - absolute needing a base
	R	Relocatable - an address in the workspace
	S	System address - treated like 'A'
	$	System address - treated like 'R'
	X	Save area - specialized treatment
USED	B	Used by both the emulator and the system
	E	Primarily used by the emulator
	S	Primarily used by the system
CBYT	A	Control byte uses address table conventions
	U	Control byte is unused
	V	Control byte contains part of the value
	Z	Control byte is zero
	*	See the detailed writeup
WRDS	-	Actual number of storage words
R03	-	Displacement from GPR3

FIGURE 2: CODES USED IN FIGURE 3

R E L O	U S E D	C B Y T	W R D S	R03	CONTROL WORDS
-	S	-	1	-A8	SYSTEM
-	-	-	1	-A4	UNUSED
A	B	V	2	-A0	FUZZER
-	S	-	1	-98	SYSTEM
-	-	-	1	-94	UNUSED
S	E	U	1	-90	CALL370F
S	B	V	1	-8C	QEND
S	E	U	1	-88	SCANRTN
S	E	U	1	-84	SERVRTN
A	E	V	1	-80	INTRTN
X	E	V	9	-7C	SAVELS
-	S	-	20	-58	SYSTEM
A	E	V	1	-08	CHKWRD
-	S	-	1	-04	SYSTEM
X	E	V	2	00	TMPSAV
-	-	-	5	+08	UNUSED
A	B	A	1	+1C	XARGO
A	E	A	1	+20	BLANK
A	E	A	1	+24	ZEROVAR
A	E	A	1	+28	ONE
$	E	A	1	+2C	REAL1
$	E	A	1	+30	PI
$	E	A	1	+34	E
$	E	A	1	+38	MIN
$	E	A	1	+3C	MAX
-	S	-	1	+40	SYSTEM
$	E	A	1	+44	NULNUMVC
$	E	A	1	+48	NULCHRVC
-	S	-	1	+4C	SYSTEM
A	B	A	1	+50	NOVALUE
R	E	A	2	+54	TMPNAM
D	B	A	1	+5C	FUNCTION
R	B	*	1	+60	NEXTINST
R	E	A	1	+64	TSADR
R	E	A	1	+68	BNDATS
R	B	A	1	+6C	$CT
R	B	A	1	+70	$IO

FIGURE 3: CONTROL WORD MAP

BLANK	Address table entry for the blank character scalar.
BNDATS	Address of the current boundary between the address table and the stack. This actually addresses byte zero of the first word below the stack words.
CALL370F	Address of the transfer vector for the external functions.
CHKWRD	Word used on entry to the emulator to check that a workspace is properly addressed by GPR3. A copy of CHKWRD is assembled into the emulator. Originally this word contained X'3D8942BC'. Bytes 0 and 1 are the workspace check pattern and are permanent. Byte 2 is the workspace/emulator check pattern; it is bumped by one whenever an emulator change requires workspace reformatting. Byte 3 is the system/emulator check pattern (the system MVI's this to the workspace on ')LOAD'); it is bumped by one whenever an emulator change requires a system change.
E	Address table entry for 2.718...
FUNCTION	The internal name of the current APL function.
FUZZER	This double word contains the comparison tolerance represented as a floating point number with an exponent of X'40' (unless the current comparison tolerance has a value which is non-meaningful).
INTRTN	This contains an APLRESM macro. When the emulator takes an interrupt, it points the 370 instruction location counter at INTRTN.
MAX	Address table entry for the largest possible real number (X'7FF...').
MIN	Address table entry for the smallest possible real number (X'FFF...').
NEXTINST	Address table entry for the next APL instruction half word. Byte 0 of this

	control word is unused but is not preserved by the emulator. Thus it must be given special attention during workspace relocation.
NOVALUE	Address table entry for a variable with no value.
NULCHRVC	Address table entry for a null character vector.
NULNUMVC	Address table entry for a null numeric vector.
ONE	Address table entry for 1 (logical).
PI	Address table entry for 3.141...
QEND	Quantum end control word. Byte 0 contains the switches (see 'GETV'). Bytes 1-3 contain the address of the system quantum end routine. On entry to the quantum end routine the emulator registers are active. If bit 31 of this word is on they have also been stored (see 'APL SYSTEM/APL EMULATOR INTERFACE').
REAL1	Address table entry for 1 (real).
SAVELS	Save area for the non-370 registers used by the emulator at interrupt (or other checkpoint) times.
SCANRTN	Location of the 370 instruction following the last APLSCAN.
SERVRTN	Location of the 370 instruction following the last APLxxxx where xxxx specifies some service function (FIND, FREE, etc).
SYSTEM	Reserved for use by the APL system.
TMPNAM	Address table entries reserved for temporary use by the emulator during stack extension, function call, etc. These two words are sometimes referred to individually as TMPNAM0 and TMPNAM1.
TMPSAV	Temporary save area for the emulator.

TSADR	Address table entry for byte 0 of the next available word on the stack.
UNUSED	Currently unused.
XARGO	Extra argument (i.e., 'global') for APL coded system functions.
ZEROVAR	Address table entry for 0 (logical).
$CT	Address table entry for QUADCT.
$IO	Address table entry for QUADIO.

THE ADDRESS TABLE

The address table consists of a series of single word entries for the various internal names. Any of these internal names may correspond to a user's external name, such as 'A' or 'FUN3', or it may be a name that the APL system is using for another purpose, such as pointing to the 'print name' for some internal name. The APL emulator may be making temporary use of a name to identify an intermediate result such as A+B or a name may not be in use at all. The full details of the address table entries are given in figures 4 to 7.

The first byte of the address table entry consists of four syntax bits and four primary descriptor bits. The syntax bits might, for example, identify the named item as a function of two arguments or as a variable (see 'STATEMENT SCAN AND SYNTAX ANALYSIS' for a description of the syntax bits and their use). The primary descriptor bits distinguish between permanent and temporary items, tell whether or not a variable has a value, and if it does, identifies it as an addressed value or an immediate value. Entries with addresses point to byte 0 of the DN word (see 'FREE SPACE').

A variable with an immediate value is called an 'address table immediate' and is a scalar character, logical, or small integer. Character immediates have their value in the last byte; the next to last byte is unused. Logical immediates have their value in the last bit; the

```
SSSS  P0PP  AAAA  AAAA  AAAA  AAAA  AAAA  AA00
SSSS  P1PP  MUUU  DDDD  VVVV  VVVV  VVVV  VVVV
```

SSSS	Syntax (see figure 5)
PPPP	Primary descriptor (see figures 6 and 7)
A..A	Absolute (virtual) address of the named block
V..V	Value
MUUU	Sign and unused
DDDD	Type descriptor (0=logical, 1=integer, 4=character, DDDD=UU1U is an 'escape' case)

FIGURE 4: ADDRESS TABLE ENTRY FORMS

SSSS=0	Unused name
SSSS=2	Variable, normal
SSSS=3	Function, dyadic
SSSS=9	Function, niladic
SSSS=B	Function, monadic
SSSS=C	Variable, shared
SSSS=D	Variable, system
SSSS=F	Group

FIGURE 5: POSSIBLE ADDRESS TABLE SYNTAX BITS

PPPP=0	Unused name not on unused name chain
PPPP=4	Unused name on unused name chain
PPPP=7	Permanent with no value
PPPP=9	Temporary with addressed value
PPPP=B	Permanent with addressed value
PPPP=F	Permanent with immediate value

FIGURE 6: POSSIBLE ADDRESS TABLE PRIMARY DESCRIPTOR BITS

(ALSO SEE THE NOTE TO FIGURE 7)

BIT 4	0=Has no value	1=Has value
BIT 5	0=Addressed value	1=Immediate value
BIT 6	0=Temporary	1=Permanent
BIT 7	(see note)	1=Normal setting

Note: The emulator does not normally use the P-bits of shared or system variables. The system need not follow the above conventions in these cases. For system variables P-bit 7 is 0 if there is an 'implicit error' associated with the variable.

FIGURE 7: ADDRESS TABLE P-BIT ASSIGNMENTS

```
                   +--------+
FPR2               |        |
           +-------|0400RRRR|
           |       +--------+
           |
           |       +--------+
           |       |IN USE  |
           |       | :      |
           |       |IN USE  |
NNNN       |   +---|0400QQQQ|<--+
           |   |   |IN USE  |   |
QQQQ       |   +-->|0400TTTT|---C--+
           |       |IN USE  |   |  |
RRRR       +------>|0400NNNN|---+  |
                   |IN USE  |      |
                   |IN USE  |      |
TTTT               |00UUUUUU|<-----+
UUUU               | :      |
                   |00UUUUUU|
                   +--------+
```

FIGURE 8: UNUSED NAME CHAIN EXAMPLE

remaining bits in the last two bytes are zero. Integer immediates have a 16-bit value in the last two bytes and a seventeenth sign bit which is replicated throughout the first two bytes when the value is extended to a full word. In figure 4 and other places, U is used to denote an unused bit. The emulator does not use the value of U, but it may change this value. The 'escape' case is discussed in 'THE GOTO PRIMITIVE FUNCTION'.

The second word of FPR2 is 0400NNNN where NNNN is the next available unused name. Whenever a name, say RRRR, is released to become 'unused', the FPR2 word is stored in the address table entry for RRRR and then the FPR2 word is changed to 0400RRRR. This yields a chain of unused names as shown in figure 8. When a name is next requested, RRRR will be given and the address table entry for RRRR read out. Since this entry is a link in the unused name chain it will replace the FPR2 word and we thus have restored FPR2 to 0400NNNN. If three more names are requested we will give NNNN and QQQQ in the same manner. Then we will give TTTT, but when the TTTT address table entry is read out it will be found to not be a link in the unused name chain. In this case four will be added to the FPR2 word to produce a next available unused name of UUUU. At the same time a check will be made to insure that UUUU is a valid name and not the lowest word in the stack area. This test consists of seeing that byte 0 of the UUUU entry is zero. Alternatively one could make a comparison with the contents of BNDATS.

THE STACK

THE USE OF THE STACK

The stack consists of four registers denoted by R1, R2, R3, R4 (actually these are the 370 registers GPR1, GPR9, GPR7, GPRE) and a sequence of memory locations M[TS+4], M[TS+8], ... M[BS]. M[BS] is the beginning of the stack. TS is contained in TSADR and its minimum allowable value is in BNDATS.

We would like to avoid repeated memory references so we keep the top stack items in registers and allow these registers to be marked 'empty'. The action of pushing an item onto the stack is as follows:

```
        If R4 is 'empty' then go to OK
        If M[TS] is 'end stack' then use an external routine
                                         to extend the stack
        M[TS] ← R4 and TS ← TS-4
   OK:  R4 ← R3, R3 ← R2, R2 ← R1 and R1 ← item
```

The end of stack marker is the same as an 'empty' marker, a zero first byte. An empty item can occur in the registers, but the emulator never puts one on the memory part of the stack. Hence an empty marker can be used to denote the end of the stack. At the beginning of execution TS=BS-4 and the stack setup is as follows ('U' denotes an unused half-byte):

```
   R1              undefined
   R2              07UUUUUU = 'null'
   R3              undefined
   R4              00UUUUUU = 'empty'
   M[BS]           08UUUUU2 = 'begin stack'

   M[BS-4]         anything but 'empty'
   :               :
   M[BNDATS+4]     anything but 'empty'
   M[BNDATS]       00000000 (hence 'empty')
```

We now begin execution with the sequence:

```
   BEGIN:     R3 ← R4
              R4 ← 'empty'
              R1 ← read next (first) APL token
```

Analysis and execution now proceed with the setup:

```
   R1   APL token
   R2   'null'
   R3   'empty'
   R4   'empty'
```

At the beginning of, for example, a dyadic operation the stack registers will be:

R1	left argument
R2	operator
R3	right argument
R4	next item on the stack

The emulator routine that executes the operator will leave the result in R2. It can then branch back to the above BEGIN. See 'STATEMENT SCAN AND SYNTAX ANALYSIS' for further details.

ITEMS ON THE STACK

This section describes operators, names and values on the stack. The stack can also contain blocks of information and special stop words (see 'FUNCTION INVOCATION').

Each item on the stack is a full word. Bits 0-3 are the syntax bits and identify the item as an operator, variable, separator, etc. A complete list of syntax codes is given in 'STATEMENT SCAN AND SYNTAX ANALYSIS'.

Operators go on the stack with hexadecimal form '1ABCUUUU' where '1ABC' denotes their opcode and 'UUUU' denotes unused. The opcodes may go through minor modification during processing, such as setting of the 'is indexed' bit. The various opcode bits are further specified in the 'OPERATORS AND SEPARATORS' section.

A name on the stack has the bit form

SSSS UUP1 UUUU UUUU NNNN NNNN NNNN NN00

where the U bits are unused, the N bits give the name, P is 0/1 for temporary/permanent names and the S bits give the syntax code. The only syntax codes that should occur with names on the stack are 2=variable, 3=dyadic function, 9=niladic function and B=monadic function. We do not stack all of the name's P-bits because they may be altered while the name is on the stack.

Immediate values may be on the stack with the bit form

0010 1110 MUUU DDDD VVVV VVVV VVVV VVVV

With the exception that the P-bits are 1110 rather than 1111, this is formatted exactly like an address table immediate. However, there is a fundamental difference. Address table immediates are always permanent variables; stack immediates are always temporary variables. In a statement like 'B←(A←1.5)+A' A may go on the stack when it is an address table immediate but it will change to a non-immediate before the stack entry is used. Because of this respecification problem, address table immediates must be put on the stack in the name form (as opposed to the immediate form). Temporary results like 2+3 cannot be respecified, so they are made into stack immediates if possible.

FREE SPACE

Free space is divided into a number of blocks. The formats of the various blocks are shown in figure 9 and are summarized in figure 10. The arrangement of these blocks in free space is

DB GA AB UB AB GA DB

where

DB	single word dummy block containing the integer 5
GA	any mixture of garbage or active blocks
AB	an active block
UB	the single unallocated block

FREEU (see the section on 'GETV') contains the address of the beginning of the unallocated block. When space is to be found for a new object, it will be taken from the bottom or top of the this block. The rightmost bit of FREEU determines the location selected with 0 indicating the bottom (low address) and 1 indicating the top (high address). If insufficient space is available then the APL system is called upon to perform a garbage collection. This causes all garbage blocks to be removed and all active blocks to be moved to the bottom of free space so that the

```
UNALLOCATED BLOCK
+-------+-----------------------------------+-------+
|       |                                   |       |
| X + 2 |       X-4 BYTES OF SPACE          | X + 2 |
|       |                                   |       |
+-------+-----------------------------------+-------+
```

```
GARBAGE BLOCK
+-------+-----------------------------------+-------+
|       |                                   |       |
|   X   |       X-4 BYTES OF SPACE          |   X   |
|       |                                   |       |
+-------+-----------------------------------+-------+
```

```
ACTIVE BLOCK
+-------+-------+---------------------------+-------+
|       |   :   |                           |       |
| X + 1 | D : N |   X-8 BYTES OF SPACE      | X + 1 |
|       |   :   |                           |       |
+-------+-------+---------------------------+-------+
```

FIGURE 9: BASIC FREE SPACE BLOCKS

```
+-------+-----------------------------------+-------+
|       |                                   |       |
|   C   |             INTERIOR              |   C   |
|       |                                   |       |
+-------+-----------------------------------+-------+
```

C — SPACE MANAGEMENT CONTROL WORD EQUAL TO B+T-4 WHERE B IS THE TOTAL NUMBER OF BYTES IN THE BLOCK AND T IS 0/1/2 ACCORDING TO THE TYPE BEING GARBAGE/ACTIVE/UNALLOCATED (IF ACTIVE THE INTERIOR MUST BEGIN WITH A DN WORD)

FIGURE 10: GENERAL FREE SPACE BLOCK

new configuration is

DB AB AB .. AB UB DB

If there is now sufficient space then execution continues, otherwise a 'workspace full' error exit is taken.

The second word of an active block is called the 'DN word'. N is the internal name of the block. Each active block is associated with a word at location GPR3+N. This word has the format SPAAAAAA (see 'THE ADDRESS TABLE' although this word is not necessarily located in the address table) where AAAAAA is the address of byte 0 of the DN word. D is a half word which describes the block. Further details about active blocks will be found in the sections specifically about them: 'VARIABLES IN FREE SPACE', 'AP VECTORS' and 'SYNONYMS'.

A garbage block is formed whenever an active block is freed. Whenever this happens the preceding and following blocks are also checked and, if either/both of them is/are inactive (garbage or the unallocated block), then it/they are merged with the newly freed block. Thus free space should never contain two adjacent inactive blocks (actually the APL system may generate this situation during cases, like editing, where it modifies free space). The first and last dummy blocks in free space contain an odd integer; this makes them look like active blocks so that the 'free a block' routine will never attempt to merge them with an adjacent block. When the garbage collector scans free space the dummy blocks look like active blocks, but with zero bytes for the interior of the block. Since this cannot occur for a true free space block the routine detects the end of the scan.

VARIABLES IN FREE SPACE

The very general form of variables in free space was described in the 'FREE SPACE' section. The more specific forms are shown in figure 11. All items are full words and are full word aligned. The various Vi represent the value words. We also have the element count in E, the rank in T, and the shape in R1 R2 .. RT. T must be less than 64; each Ri and their product, E, must be less than 2*24. U...U denotes an undefined number of undefined words. This is

NON-REAL SCALAR	C DN V0 U...U C
REAL SCALAR	C DN V0 V1 U...U C
VECTOR	C DN V0 V1 .. VN U...U E C
ARRAY	C DN V0 V1 .. VN U...U R1 R2 .. RT T E C

FIGURE 11: FORMAT OF VARIABLES IN FREE SPACE

BIT	MEANING IF ON
0	ESCAPE CASE
1	NOT SINGLE VALUED
2	ARRAY
3	ARRAY OR VECTOR
4	(ALWAYS OFF)
5	CHARACTER
6	REAL
7	REAL OR INTEGER

FIGURE 12: SECOND DESCRIPTOR BYTE DEFINITION

BITS 123	CASE
000	SCALAR
001	VECTOR, E is 1
011	ARRAY, E is 1
101	VECTOR, E not 1
111	ARRAY, E not 1

BITS 567	CASE
000	LOGICAL
001	INTEGER
011	REAL
100	CHARACTER

FIGURE 13: SECOND DESCRIPTOR BYTE CASES

BIT	MEANING
0	0 (CURRENTLY UNUSED)
1	0 (CURRENTLY UNUSED)
2	0 (CURRENTLY UNUSED)
3	0 (CURRENTLY UNUSED)
4	1 IF AND ONLY IF AP VECTOR
5	0 (MUST ALWAYS BE SO)
6	0 (MUST ALWAYS BE SO)
7	1 IF AND ONLY IF SYNONYM LINK

FIGURE 14: FIRST DESCRIPTOR BYTE DEFINITION

```
SCALAR:   100000
0000000D 0001nnnn 000186A0 0000000D

SCALAR:   .5
00000011 0003nnnn 40800000 00000000 00000011

VECTOR:   .5
00000015 0013nnnn 40800000 00000000 00000001 00000015

VECTOR:   NULL (CHARACTER)
0000000D 0054nnnn 00000000 0000000D

VECTOR:   'ABCDEF'
00000015 0054nnnn C1C2C3C4 C5C60000 00000006 00000015

ARRAY:    VALUES=1 0 1 0 0 1 1 1 1   SHAPE=3 3
0000001D 0070nnnn E5010000 00000003 00000003 00000002
00000009 0000001D
```

FIGURE 15: EXAMPLES OF VARIABLES IN FREE SPACE

usually null but an expression like 'A←,A' may produce a non-null case (see 'SYNONYMS'). The possibility of non-null U...U means that the location of E must be computed as follows: Let d be the address of the DN word. Then the address of E is d-9 plus the contents of d-4 (of the form X+1 because the block is active). T, RT, ... can be accessed by stepping backwards from E.

Integers are stored in full words and reals are stored in full word pairs (but not necessarily double words) using the standard IBM/370 representation. Characters are stored sequentially from left to right in bytes and padded on the right with undefined bytes if necessary to complete a word. The bit patterns used for character representation are defined by the APL system and are of no concern to the emulator. The emulator only needs to know the representation for a blank (for the 'expansion' and 'take' operators) and for this it uses the control word BLANK. Logical vectors are stored with eight values per byte and these bytes are stored sequentially as in the character case. Within a byte the values are stored from right to left. Hence a logical vector would begin with the elements E7 E6 E5 E4 E3 E2 E1 E0 in the first byte and E15 E14 E13 E12 E11 E10 E9 E8 in the second byte. The byte containing the last element will be padded with undefined bits on the left if necessary.

The descriptor is delineated in figures 12 to 14. It is a half word consisting of bytes D0 and D1. D0 is the 'escape' descriptor and is usually zero; the only exceptions are hexadecimal values of '01' for synonym links (see 'SYNONYMS') and '08' for AP vectors (see 'AP VECTORS'). D1 uses bit 0 to flag these escape cases. D1 has bit 4 always off. However, when the emulator is using a variable, a copy of the descriptor exists in the GPR's. In this copy, bit 4 of D1 may be used to flag initialization of the variable by some emulator routine, etc. The descriptor bits of most interest are bits 123 and 567 of D1. These are well described by figures 12 and 13. Particularly useful is bit 1, the 'pseudo scalar' bit. If this bit is on the variable is null or has more than one element. Thus if the bit is off, according to the rules of APL, it can frequently be used as a scalar, whether or not it is one.

Some examples are given in figure 15. Characters are shown in EBCDIC but the APL system may use a different code.

AP VECTORS

An AP vector is a vector of integers which form an arithmetic progression. Some examples are:

```
 1   2   3
10  13  16  19  22  25
17   3 -11 -25
```

Any AP vector can be represented in a compressed form: first element, step between elements, number of elements. The internal form for an AP vector is as shown below (all numbers are hexadecimal).

0000:0015	08D1:NAME	FIRST	DELTA	NUM ELM	0000:0015

Thus the above examples would become:

```
00000015 08D1xxxx 00000001 00000001 00000003 00000015
00000015 08D1yyyy 0000000A 00000003 00000006 00000015
00000015 08D1zzzz 00000011 FFFFFFF2 00000004 00000015
```

The APL emulator does not examine all vectors to see if they can be represented as AP vectors. But monadic iota always generates an AP vector if the element count is greater than one and the emulator will preserve AP vectors across many operations such as addition of a scalar (do one addition instead of n of them) and multiplication by a scalar (do two multiplications instead of n of them). When generating an AP vector the emulator does require that the following be true

```
1     <                          NUM_ELM
2*24  >                          NUM_ELM
2*28  >  |             DELTA  ×  NUM_ELM  -  1
2*31  >  |FIRST  +  DELTA  ×  NUM_ELM  -  1
```

The use of AP vectors reduces the memory requirements and the execution time of a number of APL expressions. Programs frequently use subscripts of the form '*A*+*B*×⍳*N*'. AP vectors allow very efficient processing of these subscripts. They also, in conjunction with subscript lists, allow the subscripting emulator routines to recognize many special cases for efficient evaluation. There are other instances of real use as well. For example, let TEXT be a string of N characters. Then the emulator will evaluate

(*TEXT*=' ')/⍳*N*

in less time and core space than would be possible without the use of AP vectors.

SYNONYMS

If B is a vector or an array, then A←B will usually cause A and B to become synonyms. In this case a single copy of the value block will be stored and both A and B will refer to this block. The use of synonyms will reduce the space and running time of most APL programs. Assuming that B is not already a synonym, figure 16 shows what happens for this assignment. T is a temporary name (but see below) and U is undefined. The quantities shown in the blocks (A, B, C, D, T, U and −1) are all half word items. The descriptors of A and B will have the synonym descriptor bits on (see 'VARIABLES IN FREE SPACE').

Several items can be synonymous; suppose A, B and C are synonyms. Then the last two items in the blocks which their address table entries point to are

A:	−1	B
B:	A	C
C:	B	−1

In other words these items show the neighboring items on the synonym chain with −1 (actually any half word with the low bit on) indicating the end of the chain. If the statement D←B occurs and a new synonym is formed then the synonym chain becomes A, B, D, C so that the link items become:

BEFORE THE ASSIGNMENT

```
+-------------+
|ADDRESS TABLE|
|-------------|
|             |
|             |
|ENTRY FOR A  |-----> ?
|             |
|             |
|             |      +-----+-----------------+
|ENTRY FOR B  |----->| D B | VALUES AND SIZE |
|             |      +-----+-----------------+
|             |
+-------------+
```

AFTER THE ASSIGNMENT

```
+-------------+
|ADDRESS TABLE|
|-------------|
|             |
|             |      +-----+-----+------+
|ENTRY FOR A  |----->| D A | U T | -1 B |
|             |      +-----+-----+------+
|             |
|             |      +-----+-----+------+
|ENTRY FOR B  |----->| D B | U T | A -1 |
|             |      +-----+-----+------+
|             |
|             |      +-----+-----------------+
|ENTRY FOR T  |----->| D T | VALUES AND SIZE |
|             |      +-----+-----------------+
|             |
+-------------+
```

FIGURE 16: ADDRESS TABLE AND FREE SPACE ITEMS BEFORE AND AFTER A←B (THE SPACE MANAGEMENT CONTROL WORDS HAVE BEEN OMITTED FOR SIMPLICITY)

A:	−1	B
B:	A	D
D:	B	C
C:	D	−1

A synonym is set up if B is a medium or large nonscalar (the emulator tests if the space management control word is greater than 64; see 'VARIABLES IN FREE SPACE') and a copy of B is required. Typical examples are

```
B   DF   E   where DF is a dyadic defined function
    MF   B   where MF is a monadic defined function
     A ← B   and the result will not fit in the old A
        ,B   where B is an array
```

The last case implies that the various synonym links may have different descriptors and that these descriptors, not the one in the value block, describe their associated variables. The last two cases imply that if B is an array then A←,B will usually set up a synonym block and that B←,B will simply change the descriptor of B.

If A and B are synonymous then A←X will cause the old value of A to be freed and the assignment to be done. If B was synonymous only with A the the synonym chain reduces to −1 −1 and in this case the synonym block is freed and B is made to point directly to the value block. Although the name of the value block is a temporary name, it is given permanent status in the address table. This protects it against freeing by the system in case of a user error (see 'ERROR RECOVERY'). The emulator will ignore this false permanent status and will free the name at the appropriate time (i.e., when the synonym chain has only one link).

OPERATORS AND SEPARATORS

Operators and separators are represented in 16 bits of the form:

SSSS DDDD DDDD DD01

The last two bits are zero-one and they specify that this is an operator or separator. The first four bits specify the syntax (see 'STATEMENT SCAN AND SYNTAX ANALYSIS'). The D-bits distinguish between the various operators. There are some special operators which have non-standard form.

OPERATORS

Operator codes are shown in figures 17 and 18. They have the form:

0001 CRZM DEFG HI01

The bit patterns for individual operators are arranged so that the emulator can quickly detect various groups of operations. The bits have the following significance:

C=1 for equal, unequal and fast mixed codes
R=1 for left slash, right slash (and their '-' overstrikes) and for period
Z=1 for operators overstruck with '-'
M=1 for mixed operators
E=1 for indexable operators

In the case of scalar operators (M=0) FG is 00 for comparisons and 01 for logical operations. Also, in the scalar operator case, character arguments produce a 'domain' error unless C=1.

	1	5	9	D
100			<	≤
101	∨	∧	⍲	⍱
102	+	×	⌈	!
103	*	?	○	
108			≥	>
109		→		~
10*A*	−	|	⌊	
10*B*	⍟		÷	
180		=		
188		≠		
189		→		

↑
|
+---*SCALAR OPS*

MIXED OPS------→
(FAST CODES
OMITTED --
ADD X'0800')

	1	5	9	D
110	⍴	↑	⍳	⊥
111	∘			
112	⌶		⍎	
115			⌽	⍋
118	⍉	↓	∊	⊤
11*A*	⌹		▼	
11*B*				[illegible]
11*D*			,	⍒
135			⊖	
155		/		
159	.			
15*D*		\		
175		⌿		
17*D*		⍀		
C00	⎕	⍞		

Note: Row headings give the first three digits.
Column headings give the last digit.
For example, 108D is the code for >.

FIGURE 17: OPERATORS ARRANGED BY HEXADECIMAL CODE

F	*G*	0	1	2	3	4	5	6	7	← *BITS I H D*
0	0			<	≥	=	≠	≤	>	*SCALAR OPS*
0	1	∨		⍱		∧	→	⍲	~	
1	0	+	−	⌈	⌊	×	|	!		
1	1	*	⍟	○	÷	?				
0	0	⍴	⍉	⍳	∊	↑	↓	⊥	⊤	*MIXED OPS*
0	1	∘	.	⌽	,	/	\	⍋	⍒	
1	0	⌶	⌹	⍎	⍕					
1	1								⎕	

NOTE: LOOKING ONLY AT BITS F G AND I H D WE HAVE THE FOLLOWING EQUIVALENCES: ⌽:⊖ /:⌿ \:⍀ NORMAL MIXED OP:FAST MIXED OP

FIGURE 18: OPERATORS ARRANGED BY FUNCTIONAL GROUP

4001	)	6001	;
4005	]	7001	*FAST* ←
5001	(	7101	*NORMAL* ←
5005	[	8005	→
500*D*	⍅	*A0X*1	*END*

FIGURE 19: SEPARATORS

For the mixed operators, C=1 flags the 'fast' case. Suppose V is a variable, X is a mixed operator and the emulator is evaluating the expression X V. Furthermore, suppose that the value of X V is the same as the value of V. For example X is ravel and V is a vector, or X is transpose and V is a scalar, vector or one element array. Then:

if C=1 then the result of X V is V

if C=0 then the result of X V is a copy of V

(In the later case the 'copy' may actually be a synonym.) These results do have the same value but they may have different side effects. In the statement 'A←,V' either code may be used. But, if V is a vector, then in the statement '(V←6)+,V' only the normal code will produce the same result as APL\360 (the fast code produces the scalar 12).

If the emulator detects two contiguous operators and if either has R=1 then it checks for reduction, scan or inner or outer product. When the emulator is actually performing an operation it usually holds a copy of the operator in the left half of GPR9. However, it may change certain bits to indicate special conditions. For example, in scalar operations E is usually set to 1 if real arithmetic is needed. Also, Z is set to 1 if an operator is explicitly indexed.

The operators with M=1 and F=1 cause an exit to the APL system. The emulator does not define the properties of these operators. The operator denoted by the boxed star (X'11BD') cannot be entered into the system by an APL user. It is generated by the emulator during the processing of a secondary decode special operator (see below). The emulator implements it by invoking the system's 'box' functions (see 'APL ASSIST AND THE APL SYSTEM').

THE GOTO PRIMITIVE FUNCTION

There are two internal forms for the GOTO arrow, namely, 1095 and 1895. The 1095 form should be used if the arrow is the first item in any statement of a defined function. The 1895 form must be used if the arrow is not the first item in a statement or if it is a statement for immediate execution. Both 1095 and 1895 produce a 'rank' error or a 'domain' error if appropriate.

The 1095 form is significantly faster than the 1895 form. It assumes that the next item to be scanned is an end of statement marker and it checks this item for the trace bit. If the trace is on then the emulator behaves as if the 1895 form had been used (see below). If the trace is off then it either continues execution on the next statement (if the argument is null) or it branches to the statement number specified by the argument.

The 1895 form does not immediately do the branch (or no-branch) operation. Instead, it produces a result which is an escape form of stack immediate. If the argument is null then the result is X'2E-3UUUU' where '-' is B'1UUUU' and 'U' is undefined. If the argument is not null then the result is X'2EZ3NNNN' where 'Z' is B'0UUUU' and where 'NNNN' specifies the branch target. If the result of a GOTO is used as the argument of a primitive or defined function, as for example in 2+→3, then the GETV process (see '370 REGISTERS AND EXTERNAL FUNCTIONS') will cause a 'value' error. If a GOTO is used in a context such as ⍎'⍎''→2+3''' then the result of the GOTO (for example, X'2E030005') will be on the stack when the end of statement is reached. The emulator will then make a 'print' error return to the system (see 'APL SYSTEM/APL EMULATOR INTERFACE') which may do the actual branch.

To summarize the situation: The emulator will process the usual situation where the GOTO is the first item in a statement of a defined function. It analyses unusual usages of GOTO and either gives a 'value' error or presents the GOTO as the result of the statement.

SEPARATORS

The codes for the various separators are shown in figure 19. The two assignment arrows, 7001 and 7101, have the same effect in most circumstances, but they have different side effects if there are multiple assignments in one statement. The emulator will produce the same result as APL\360 if either of the following rules is used:

(a) Always use 7101.

(b) Let f be the first name in the APL statement

--- f x ...

If x is a left arrow and --- does not contain left parenthesis or bracket, use code 7001. In all other cases use 7101.

Rule (b) is more complicated than rule (a) but it leads to more efficient execution. Let Y be the item being assigned to something. The 7101 arrow leaves the value of Y on the stack; the 7001 arrow may leave the value of Y on the stack or it may leave the name of the variable into which the assignment is made. In statements like

A ← B + C

the two arrows produce the same result. In statements like

(A←B) + A←C

they produce different results.

The 8005 separator is the bracket used when an operator is indexed. (It is generated automatically by the APL system and cannot be entered into the workspace by overstriking the bracket with a minus.) The X in the end of statement marker is: 0 for no stop or trace, 1 for trace (this statement), 2 for stop (before the next statement), and 3 for both stop and trace. APL functions which are part of the system may use the separator 500D. This separator works like 5005 except that it allows an array to be indexed like a vector. In other words, a 500D type subscript on a scalar, vector or an array has the same effect as a 5005 type subscript on the ravel of a scalar, vector or array. (Like 8005 it cannot be typed by the user.)

SPECIAL OPERATORS

A special operator has a 16 bit code ending in 11. The defined codes are:

0NNN NNNN NNNN 0011	GOTO N, corresponds to 1095 form
1NNN NNNN NNNN 0011	GOTO N, corresponds to 1895 form
UUUU UUUU UUUU 0111	make an 'escape' emulator exit
UUUU UUUU UUU0 1011	perform an indirect operation
UUUU UUUU UU01 1011	skip over some function bytes
VVVV VVVV VVVV 1111	secondary decode

The purpose of the 'escape' operation is not defined by the emulator. As an example of its use, APL/CMS uses hexadecimal XX07 to flag an illegal character, where XX gives a representation of the character, and it uses NNNN TTF7 to flag an assignment to a stop or trace vector of a function. In this case NNNN is the internal name of the function and TT is the internal representation of 'S' or 'T'. The indirect operation may be used by some APL system routines (such as the one for the 'scan' operator). If i is the 'indirect operation' operator and N is a name, then Ni causes the emulator to get the low order eight bits of the address table entry for N and to use these eight bits as the low bits of a scalar operator. Thus if N is an integer address table immediate with the value five, then the emulator adds 18 to produce the scalar operator 1805 which is the code for '='.

The skip operator is used to skip over a portion of the internal text. It might be used, for example, to include comments in the text. The form in which it is used is:

S N B1 B2 ... BN

where S is the skip operator, N is a halfword even integer and the Bi are the bytes to be skipped over.

The secondary decode operation causes the emulator to put the word:

0001 0001 1011 1101 VVVV VVVV VVVV 1111

on the stack. This will subsequently be treated like a '11BD' operation and it will eventually call the external function corresponding to '11BD' ('m box' or 'd box', see 'APL ASSIST AND THE APL SYSTEM'). The external function will find the VV...11 in the low half of GPR9 and it can use it to select one of many sub-operations.

INTERNAL TEXT OF FUNCTIONS

A function has the same internal form as a character vector, however, the syntax bits in the address table will distinguish between a variable and a function of 0, 1 or 2 arguments. The internal form of a function is:

C DN HEAD BODY TAIL SYS NB C

C is the usual space management control word (see 'FREE SPACE'). D is the descriptor of a character vector (0054) and N is the internal name of the function. HEAD contains the half word items:

M T S K Z L R L1 L2 ... LN 2 EZ

where we have ...

M	highest statement number
T	byte offset of TAIL from DN
S	system information, not used by the emulator
K	40 + 8 times the the number of locals (decimal)
Z	name of the result or the number 1
L	name of the left argument or the number 1
R	name of the right argument of the number 1
Li	name of the i-th local variable
2	marker for the end of the locals list
EZ	marker for the end of statement 0

Note that since the low two bits of a name are zero we can use both 1 and 2 to indicate a non-name. The BODY has the form:

S1 E1 S2 E2 ... SM EM X EX

where Si is the internal text of statement i (see 'INTERNAL TEXT OF STATEMENTS'). Ei marks the end of statement i. It contains the trace bit for statement i and the stop bit for statement i+1. X is an 'immediate go to 0'. Further details of Ei (see 'SEPARATORS') and X (see 'SPECIAL OPERATORS') are given in the 'OPERATORS AND SEPARATORS' section. The TAIL contains the byte offsets of EZ, E1, E2,

THE APL FUNCTION ...

```
 ∇   Z ← A F B;C;D;E
[1]  Z ← A+B
[2]  ⍝ COMMENT
[3] LAB: C ← D*E
[4]  Z;C;A
[5]  →A/LAB
 ∇
```

WITH A TRACE VECTOR OF 3
AND A STOP VECTOR OF 1 4
HAS INTERNAL FORM ...

00000079	005400C4	0005005E	00000040
00BC00C0	00C800CC	00D000D4	0002A021
00C81021	00C07001	00BCA001	001B0008
8D4C5856	564E575D	A00100D4	103100D0
700100CC	A03100C0	600100CC	600100BC
A0010116	00031555	00C01095	A0010003
A001001A	00260034	0040004C	00580001
000300D8	0000006C	00000079	

WHERE WE HAVE UNDERLINED ALL END OF
STATEMENT MARKERS INCLUDING EZ AND EX.

FIGURE 20.1: INTERNAL FUNCTION TEXT EXAMPLE

THE CORRESPONDENCE BETWEEN OUR DEFINED SYMBOLS, THE APL VARIABLES AND THE ACTUAL INTERNAL FORM IS ...

DEF	APL	INTERNAL					
C		0000	0079				
DN		0054	00C4				
M		0005					
T		005E					
S		0000					
K		0040					
Z	Z	00BC					
L	A	00C0					
R	B	00C8					
L1	C	00CC					
L2	D	00D0					
L3	E	00D4					
TAIL		001A	0026	0034	0040	004C	0058
SYS		0001	0003	00D8			
NB		0000	006C				

FIGURE 20.2: INTERNAL FUNCTION TEXT EXAMPLE

... EM as half word items. SYS contains system information such as label names. The emulator is not concerned with the details of SYS. NB is the number of bytes in the HEAD, BODY, TAIL and SYS. Figure 20 provides an example of a translated function.

INTERNAL TEXT OF STATEMENTS

TRANSLATION OF ITEMS

The external form of a statement may contain comments, labels, names, constants, operators and separators. See 'OPERATORS AND SEPARATORS' for the various 16 bit codes into which these items are translated. The remaining items are translated as follows:

Comments:

Comments may occur within any statement. See the skip operator in 'SPECIAL OPERATORS'.

Labels:

Labels should not occur in the body of a function. The system may store labels in the SYS region of the function. Also see 'USE OF LABELS'.

Names:

An external name is represented by an internal name. An internal name is a 16 bit number ending with two zero bits. In a given workspace, an external name has the same internal name irrespective of whether the name is the name of a local variable, a shared variable, a global variable or a function.

Constants:

A constant may be scalar, 16 bit or general. A constant is translated into a descriptor followed by the internal representation of the constant, according to the following bit formats:

```
Scalar:    0000 DDDD UUUU 0010 VV...
16 Bit:    MUUU DDDD UUUL 0110 VV...
General:   DDDD DDDD UUUL 1010 CCCC CCCC CCCC CC00 VV...
     or    DDDD DDDD UUUL 1010 CCCC CCCC CCCC CC10
                               UUUU UUUU UUUU UUUU VV...
```

where U stands for unused and D..D is the descriptor bits described in 'VARIABLES IN FREE SPACE'. L is used to flag label constants (see 'USE OF LABELS'). The first type of representation is used for integer scalars and real scalars. Integers are in IBM/370 32-bit integer format and reals are in IBM/370 64-bit floating point format (VV...). The 16 bit form is used for logical, character and short integer scalars. In the latter case M is the sign bit and VV... is 16 bits long. As examples of this representation (in hexadecimal):

```
0006 0001      logical   1
0106 0040      integer  64
8106 FFC0      integer -64
0406 0099      character with internal code of 99
```

The general form can be used for vectors or arrays. In this case VV... must begin on a full word boundary (hence the two forms shown). VV... must be of the form described in 'VARIABLES IN FREE SPACE'. This implies that it must be padded out to a full word and should end with an element count. CCCC CCCC CCCC CC00 is equal to four times N+2 where N is the number of words in VV... As an example, the three element vector 64 -64 1024 has the internal representation (assumming that it does not begin on a full word boundary):

```
510A 001A 0000 0000 0040 FFFF FFC0 0000 0400 0000 0003
```

USE OF LABELS

The emulator does not recognize the use of labels. If the program contains →ALPHA and ALPHA is a label attached to statement 64 then ALPHA has the internal representation 0116 0040. This is the internal representation of the short integer 64 with the L bit on. The emulator ignores the L bit. The system may use the L bit when converting from internal to external form. The APL user may, of course, use labels in any legal manner.

EXAMPLE WORKSPACE

In this section we provide a workspace which has intentionally been setup to produce an error, thus supplying an example with information on the stack, shadowed variables and so on. Figure 21 gives the console listing for the example, figure 22 delineates several key items and figure 23 gives a dump of the workspace.

In figure 21 we see that the workspace was loaded, the GO function executed and a 'domain' error produced. Figure 23 shows the contents of the workspace at this point. The console listing in figure 21 then goes on to show the stack and the items in the workspace.

Figure 22 gives the symbols in sequence by both external name and by internal name as well as several other items. We note here that on the dump the displayed GPR's are those active when the system provided the dump; the GPR's of interest to us are stored in locations 560A8 to 560E4 in the sequence GPR4, ..., GPRF, GPR0, ..., GPR3. Thus GPR3 is found to be 563A0. Since 'A' has internal name 00D0 its address table entry is at 563A0+D0 or 56470.

The beginning of free space was calculated as follows: TSADR points at the next available stack word. From here one scans up the stack through increasing core addresses until coming to the 'beginning of stack' marker, 08000002. This is immediately followed by the free space bottom dummy block, 00000005.

The stack in the workspace dump (figure 23) shows a temporary function calling GO and GO calling F which then calls G. These function call blocks will be described later in the 'FUNCTION INVOCATION' section. The reader may wish to review this section at that point. The dump is worth studying in detail to find such things as a shadowed AP vector (P in GO) and a synonym chain linking an array (A) and a vector (shadowed Q).

```
      )LOAD EXWKSP
SAVED  13.59.36 02/27/73

      GO
DOMAIN ERROR
G[1]  X←U+2
       ∧

      )SIV
G[1] *  Q        U        X
F[3]    R        Q        P        B        A        Z
GO[2]

      ∇GO[⎕]∇
    ∇ GO
[1]   R←3 3⍴P←⍳9
[2]   R F(⍳3)∘.×⍳3
    ∇

      ∇F[⎕]∇
    ∇ Z←A F B;P;Q;R
[1]   Q←,A
[2]   R←A+B
[3]   Z←A[G'X';3;]
    ∇

      ∇G[⎕]∇
    ∇ X←G U;Q
[1]   X←U+2
    ∇
```

FIGURE 21.1: EXAMPLE WORKSPACE CONSOLE LISTING

```
          ⎕NL 2
U
A
B
R

          A
 1  2  3
 4  5  6
 7  8  9

          B
 1  2  3
 2  4  6
 3  6  9

          R
  2    4    6
  6    9   12
 10   14   18

          U
X
```

FIGURE 21.2: EXAMPLE WORKSPACE CONSOLE LISTING

EXTERNAL NAME	INTERNAL NAME	ENTRY ADDRESS	VALUE OR ADDRESS
A	00D0	56470	57600
B	00D8	56478	575C0
F	00D4	56474	574A0
G	00C0	56460	57464
GO	00E4	56484	57508
P	00DC	5647C	NO VALUE
Q	00C8	56468	NO VALUE
R	00E0	56480	57614
U	00C4	56464	IMMEDIATE 'X'
X	00BC	5645C	NO VALUE
Z	00CC	5646C	NO VALUE
X	00BC	5645C	NO VALUE
G	00C0	56460	57464
U	00C4	56464	IMMEDIATE 'X'
Q	00C8	56468	NO VALUE
Z	00CC	5646C	NO VALUE
A	00D0	56470	57600
F	00D4	56474	574A0
B	00D8	56478	575C0
P	00DC	5647C	NO VALUE
R	00E0	56480	57614
GO	00E4	56484	57508

ITEM	ADDRESS	VALUE	
GPR3	560E4	563A0	(SEE TEXT)
GPRB	560C4	56000	(SEE TEXT)
TSADR	56404	5739C	
NEXTINST	56400	57484	
FUNCTION	563FC	00C0	
BNDATS	56408	57058	
FREE SPACE		5747C	(SEE TEXT)

FIGURE 22: EXAMPLE WORKSPACE ITEMS

```
GPR0 = 00000020 0000229B 0002034C 000021C0 000022C2 000022C2 00000020 FFFFFFE9
GPR8 = 000183EC C4D4E2C5 D9D94040 00056000 000113E0 0000CB88 00012620 400113EC

FPR0 = 4000000000001C25
FPR2 = 00057624040008C4
FPR4 = 0000000000000000
FPR6 = 0000000000000000

056000 00000000 5002AC48 00020000 00021766 0003B762 00000008 00000002 2E010002
056020 0007C354 10211021 00000000 00056000 60031FB2 0007CC00 07000000 40032016
056040 40000000 00001C25 00057624 040008C4 00000000 00000000 00000000 00000000
056060 FFD50000 70032022 00000201 00000000 000374A0 000207E0 00056000 60035680
056080 000360B2 5F9A9191 00000000 00000000 00000000 00000000 00000000 00000000
0560A0 00057624 040008C4 0003B762 00000008 00000002 2E010002 0007C354 10211021
0560C0 00000000 00056000 000575E8 00000009 07000000 00000000 00000061 2F0000C4
0560E0 2F040061 000563A0 000207E0 00000000 064E6160 545C5900 00000000 00000000
056100 00000000 00000000 81820E83 880E8886 038597D3 F161F1F5 00000400 00001000
056120 00026C00 000263FC 00000000 000015A8 000263F4 0000000A 00000001 07000004
056140 00000400 0078FF01 0005758C 00000001 0007CE8C 00000000 00000002 00001594
056160 00024E3E 00056000 4C03173A 0007CC00 0007CE8A 400233F4 0007CE88 00000008
056180 00048888 000000E4 00000000 00000000 00000000 00000000 00000000 00000000
0561A0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0561C0 TO 056240 SUPPRESSED LINE(S) SAME AS ABOVE .....
056240 00000000 00000000 80008081 00000000 00000000 00000000 00000000 00000000
056260 00039F48 00000002 00000000 00000000 00000000 00000000 4530B27C 91083001
056280 4710B28C 6020B0A0 9042B0A8 5820B294 07F20700 00039C88 00039F48 00000000
0562A0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0562C0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 A0100000
0562E0 A0110000 00000000 00000000 00000000 00000000 00000000 00000005 00000000
056300 40000000 00001C25 00000000 00000000 00030398 8202DE20 00021766 00031886
056320 A0020000 00000000 00000000 0000003D 4003C178 9003AC5E 00000000 00000000
056340 01000001 00000004 0003BE26 00000000 0003AC0C 00039D66 00000000 00000000
056360 0007CEA2 00000000 0004947A 000014C2 00032F10 00048000 0003AA88 60039D6C
056380 00000000 5C031876 00000002 00000000 000010C0 0002605C 3D8943BE 00000054
0563A0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0563C0 2F04008D 2F000000 2F000001 2B02DE4C 2B02DE60 2B02DE74 2B02DE88 2B02DE9C
0563E0 2F000000 2B02DEB0 2B02DEC0 27000000 27000050 2707C390 2F040061 2F0000C0
056400 00057484 2B05739C 2B057058 DF00000D DF000001 DF000000 DF00000A DF000078
```

FIGURE 23.1: EXAMPLE WORKSPACE DUMP

```
056420 DF000000 D4000000 D4000000 D4000000 D4000000 D4000000 D4000000 D4000000
056440 D4000000 D4000000 D4000000 D4000000 D4000000 D4000000 D4000000 27000000
056460 BB057464 2F040061 27057614 27000000 2B07C390 3B0574A0 2B07C3A4 27000000
056480 2B07C350 9B057508 27000000 27000000 27000000 27000000 27000000 27000000
0564A0 27000000 27000000 27000000 27000000 27000000 27000000 27000000 27000000
0564C0 TO 056840 SUPPRESSED LINE(S) SAME AS ABOVE .....
056840 27000000 27000000 27000000 27000000 27000000 27000000 27000000 0F016100
056860 0F015000 0F015E00 0F015A00 0F016300 0F014A00 0F014F00 0F014B00 0F015900
056880 0F015B00 0F025058 07000000 07000000 07000000 07000000 07000000 07000000
0568A0 07000000 07000000 07000000 07000000 07000000 07000000 07000000 07000000
0568C0 TO 056C40 SUPPRESSED LINE(S) SAME AS ABOVE .....
056C40 07000000 07000000 07000000 07000000 07000000 07000000 07000000 99057578
056C60 040008CC 040008C0 2B0575C0 00000000 00000000 00000000 00000000 00000000
056C80 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
056CA0 TO 057040 SUPPRESSED LINE(S) SAME AS ABOVE .....
057040 00000000 00000000 00000000 00000000 00000000 00000000 00000000 FF000000
057060 FF000000 FF000000 FF000000 FF000000 FF000000 FF000000 FF000000 FF000000
057080 TO 057380 SUPPRESSED LINE(S) SAME AS ABOVE .....
057380 FF000000 FF000000 FF000000 FF000000 2F0000C4 10211021 2E010002 07000000
0573A0 0F000030 2F000002 2B057614 2F0000C8 2F000001 2F0000C4 27000000 2F000001
0573C0 2F000001 2F0000BC 0F0000D4 0F000042 60016001 2E010003 60016001 62016205
0573E0 40054005 07000000 0F000040 2F000002 2B057600 2F0000E0 2F000001 2F0000C8
057400 2B0575A8 2F0000DC 2F000001 2F0000D8 2F000001 2F0000D0 2F000001 2F0000CC
057420 0F0000E4 0F000054 07000000 0F000028 2F000002 27000000 2F000001 27000000
057440 2F000001 27000000 2F000001 0F0008BC 0F000018 07000000 08000002 00000005
057460 00000039 005400C0 0001002A 00000030 00BC0001 00C400C8 0002A001 01060002
057480 102100C4 700100BC A0010003 A0010016 00240000 0000002C 00000039 00000065
0574A0 005400D4 00030050 00000040 00CC00D0 00D800DC 00C800E0 0002A001 00D011D9
0574C0 700100C8 A00100D8 102100D0 700100E0 A0014005 60010106 00036001 04060061
0574E0 00C05005 00D07001 00CCA001 0003A001 001A0024 0030004A 00000000 00000058
057500 00000065 0000006D 005400E4 0002005A 00000028 00010001 00010002 A0010106
057520 00091109 710100DC 1101510A 00160000 00000003 00000003 00000002 700100E0
057540 A0010106 00031109 10251591 11114001 01060003 11095001 00D400E0 A0010003
057560 A0010014 00380054 00000000 00000060 0000006D 0000002D 005408BC 0001001E
057580 00000028 00010001 00010002 A00100E4 A0010003 A0010014 00180000 00000020
0575A0 0000002D 00000015 08D11060 00000001 00000001 00000009 00000015 0000003D
0575C0 007108C8 00000001 00000002 00000003 00000004 00000005 00000006 00000007
```

FIGURE 23.2: EXAMPLE WORKSPACE DUMP

```
0575E0 00000008 00000009 00000003 00000003 00000002 00000009 0000003D 00000011
057600 01F11050 000008C8 FFFF00D0 00000011 00000011 01D11008 000008C8 00D0FFFF
057620 00000011 00024D26 00000000 00000000 00000000 00000000 00000000 00000000
057640 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
057660 TO 07C340 SUPPRESSED LINE(S) SAME AS ABOVE .....
07C340 00000000 00000000 00024D26 0000003D 007100E0 00000002 00000004 00000006
07C360 00000006 00000009 0000000C 0000000A 0000000E 00000012 00000003 00000003
07C380 00000002 00000009 0000003D 00000011 01F100D0 000008C8 10501008 00000011
07C3A0 0000003D 007100D8 00000001 00000002 00000003 00000002 00000004 00000006
07C3C0 00000003 00000006 00000009 00000003 00000003 00000002 00000009 0000003D
07C3E0 00000014 08D108C8 00000001 00000001 00000003 00000014 00000005 04BC08B8
07C400 04E8005C 00200004 00080008 00020014 01400040 00040004 00100010 00100014
07C420 800004BC 800004CC 00010010 00020014 800004D4 800004C4 800004D0 800004D8
07C440 800004DC 00010004 800004C8 800004E0 800004C0 800004E4 00000000 00000000
07C460 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
07C480 TO 07CC00 SUPPRESSED LINE(S) SAME AS ABOVE .....
07CC00 4D5E5659 00C20041 00000000 00000000 00000000 00000000 00000000 00000000
07CC20 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
07CC40 0005758E 00001578 00026E84 00000008 50580000 00000000 00000003 00032ED8
07CC60 00000000 00000000 00000000 00000000 0007CE85 00000003 00000000 00000000
07CC80 00000000 4C03173A 4C031800 00000000 000D0000 00000000 00000000 00000000
07CCA0 00000000 00000000 00000000 00010100 10000000 00000000 4C03173A 5C031876
07CCC0 0000004D 00000004 00000004 000329FC 00000000 00000004 C9D9C5C3 E3D6D9E8
07CCE0 00000000 00000000 00026E8A 01820E83 880E8886 03852009 F161F1F5 00000000
07CD00 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
07CD20 00000000 00000000 00000000 00000000 00020000 00000000 00000001 00056000
07CD40 4C027D52 00031FA2 00020000 00021766 0003B762 00000008 60031FB2 40032010
07CD60 0007CE8C 0007CE8C 00000002 00048888 00000000 00000000 00000000 00000000
07CD80 00004D58 564A5257 8D4E5B5B 585B9293 0007C3C0 00000003 00000000 00000000
07CDA0 00000000 00000000 00000000 00000000 00000000 00000000 105B1050 58108200
07CDC0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
07CDE0 TO 07CE80 SUPPRESSED LINE(S) SAME AS ABOVE .....
07CE80 00000000 8D8D8D8D 8D8D5058 5E56598D 58576054 5C59848D 81890E83 820E8885
07CEA0 92930000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
07CEC0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
07CEE0 TO 07D000 SUPPRESSED LINE(S) SAME AS ABOVE .....
```

FIGURE 23.3: EXAMPLE WORKSPACE DUMP

SECTION III: APL EXECUTION

APL SYSTEM/APL EMULATOR INTERFACE

The most important function of the emulator is to execute APL statements. The emulator also provides service functions which can be used by the software to assist the translator, the external functions and the error recovery procedure. The execution of APL statements and the service functions are initiated by IBM/370 assembler language macro instructions. All such macros rely on a single instruction which has been added to the 370 instruction set. The APL Emulator Call (APLEC) is an SI instruction with opcode A0. It is similar to SVC in that the immediate byte gives a call code and certain registers may be used for arguments and results. It is dissimilar in that, additionally, GPR3 must properly address a workspace or a specification exception will occur. We pointed out earlier that the APL emulator works in an environment consisting of a workspace and the 370 registers. This environment is assumed throughout this report. Thus when we say, for example, that APLSCAN will cause scanning and execution of the workspace we are assuming that GPR3 addresses a workspace as described earlier, that GPR1, GPR9, GPR7 and GPRE are properly set up as stack registers (see 'THE STACK') and so on. Figure 24 summarizes the APL macros, figure 25 gives their assembly language definitions and figure 26 details their register usage. This section discusses each macro in the order given in figure 24. 'Exceptions' are a real program exceptions like 'specification'. 'Errors' are APL error returns signaled by a condition code of 1 and an error code in GPR5. The APL errors may be true user errors such as 'syntax' or pseudo errors such as a request for printing a value left on the the stack at the end of execution of a statement. For an exception designated as a 'Trap', control will not return to the next sequential instruction. The exception will be detected in an external routine which will branch directly to the error handling routine. Figure 27 lists the APL error return codes.

XXXX	ARGUMENT	RESULT	FUNCTION
FIND	R5=bytes	R4=DN addr	find a free space block
FREE	R5=name	none	free an item
FRIF	R5=name	none	free an item if temporary
NAME	none	R4=name	provide an unused name
UNAM	R5=name	none	release an obsolete name
SCAN	none	none	scan/execute a workspace
GETV	see text	see text	get a stacked variable
GETN	see text	see text	get a number from a variable
RTN	none	see text	normal 370 function return
SRTN	none	see text	special 370 function return
RESM	none	none	resume interrupted workspace
DIAG	see text	see text	diagnostic function
CSL	see text	see text	control store load

FIGURE 24: SUMMARY OF THE VARIOUS APLXXXX MACROS

```
         MACRO
&L       APLEC &CODE
&L       DC    AL1(X'A0',&CODE,0,0)
         MEND

         MACRO
&L       APLFIND &MODE=0
&L       APLEC X'63'+8*&MODE
         MEND

         MACRO
&L       APLFREE &MODE=0
&L       APLEC X'83'+8*&MODE
         MEND

         MACRO
&L       APLFRIF &MODE=0
&L       APLEC X'A3'+8*&MODE
         MEND

         MACRO
&L       APLNAME &MODE=0
&L       APLEC X'23'+8*&MODE
         MEND

         MACRO
&L       APLUNAM &MODE=0
&L       APLEC X'43'+8*&MODE
         MEND

         MACRO
&L       APLSCAN &MODE=0
&L       APLEC X'00'+8*&MODE
         MEND
```

FIGURE 25.1: DEFINITIONS FOR APL MACROS

```
         MACRO
&L       APLGETV &VAR,&MODE=0
         LCLA  &VARC
&L       DS    0H
&VARC    SETA  X'02'
         AIF   ('&VAR' EQ 'LEFT').VAROK
&VARC    SETA  X'68'
         AIF   ('&VAR' EQ 'RIGHT').VAROK
         MNOTE 'BAD VARIABLE SPECIFICATION - RIGHT ASSUMED'
.VAROK   AIF   ('&MODE' EQ '0').MODE0
         MVI   GPR5+3,&VARC
         AGO   .APLEC
.MODE0   LA    5,&VARC
.APLEC   APLEC X'D3'+8*&MODE
         MEND

         MACRO
&L       APLRTN &MODE=0
&L       APLEC X'01'+8*&MODE
         MEND

         MACRO
&L       APLSRTN &MODE=0
&L       APLEC X'02'+8*&MODE
         MEND

         MACRO
&L       APLRESM &MODE=0
&L       APLEC X'02'+8*&MODE
         MEND

         MACRO
&L       APLDIAG &MODE=0
&L       APLEC X'E3'+8*&MODE
         MEND

         MACRO
&L       APLCSL
.* MODE MUST BE ZERO
&L       APLEC X'F3'
         MEND
```

FIGURE 25.2: DEFINITIONS FOR APL MACROS

```
         MACRO
&L       APLGETN &VAR,&ENTRY,&TYPE,&MODE=0
         LCLA  &VARC,&ENTRYC,&TYPEC,&ARG
&L       DS    0H
&VARC    SETA  X'02'
         AIF   ('&VAR' EQ 'LEFT').VAROK
&VARC    SETA  X'68'
         AIF   ('&VAR' EQ 'RIGHT').VAROK
         MNOTE 'BAD VARIABLE SPECIFICATION - RIGHT ASSUMED'
.VAROK   AIF   ('&ENTRY' EQ 'FETCH').ENTRYOK
&ENTRYC  SETA  1
         AIF   ('&ENTRY' EQ 'INIT').ENTRYOK
&ENTRYC  SETA  2
         AIF   ('&ENTRY' EQ 'CVT').ENTRYOK
         MNOTE 'BAD ENTRY SPECIFICATION - CVT ASSUMMED'
.ENTRYOK AIF   ('&TYPE' EQ 'LOG').TYPEOK
&TYPEC   SETA  1
         AIF   ('&TYPE' EQ 'INT').TYPEOK
&TYPEC   SETA  3
         AIF   ('&TYPE' EQ 'REAL').TYPEOK
&TYPEC   SETA  2
         AIF   ('&TYPE' EQ 'ASIS').TYPEOK
         MNOTE 'BAD TYPE SPECIFICATION - ASIS ASSUMMED'
.TYPEOK  ANOP
&ARG     SETA  &VARC+256*(&ENTRYC+4*&TYPEC)
         AIF   ('&MODE' EQ '0').MODE0
         MVC   GPR5+2(2),=AL2(&ARG)
         AGO   .APLEC
.MODE0   LA    5,&ARG
.APLEC   APLEC X'C3'+8*&MODE
         MEND
```

FIGURE 25.3: DEFINITIONS FOR APL MACROS

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
FIND	P	P̲	A̲	P̲	R	R	P	P̲	A̲	P̲	D	P	A̲	P	P	P
FREE	P	P	P	P̲	P	D	P	P	P	P	P	P	P	D	P	P
FRIF	P	P	P	P̲	P	D	P	P	P	P	P	P	P	D	P	P
NAME	P	P	P	P̲	R	P	P	P	P	P	D	P	P	P	P	P
UNAM	P	P	P	P̲	P	P	P	P	P	P	P	P	P	D	P	P
GETV	U	U̲	U	P̲	P	D'	U	U̲	U	P	D	P	P	P	P	P
GETN	U̲	U̲	U̲	P̲	I	D	U̲	U̲	U̲	P	P	P	P	P	P	P
CSL	P	P	P	P̲	U̲	P	P	P	P	P	P	P	P	P	P	P

P   Preserved
I   Preserved except for the INIT entry
D   Destroyed
D' Destroyed by the macro, then preserved by APLEC
A   Addresses preserved - macro may cause a garbage collection, if the register is doing it's normal addressing function the new value will be posted, otherwise the register may be destroyed
U   Updated or preserved - for GETV and GETN the macro references left or right, corresponding registers will be updated, other registers will be preserved
R   Result
_   Only underlined registers must contain the correct values expected by the microcode (if the microcode is to maintain them correctly)

Note: This figure assumes a normal return; an error exit always destroys registers 4 and 5.

FIGURE 26: GPR TREATMENT BY THE VARIOUS APLXXX SERVICE MACROS

DEC	HEX	REASON
08	08	DOMAIN ERROR
12	0C	ESCAPE OPERATOR
16	10	INDEX ERROR
20	14	LENGTH ERROR
24	18	NONCE ERROR
28	1C	PRINT EXIT
32	20	RANK ERROR
36	24	EOS STOP BIT ON
40	28	SYNTAX ERROR
44	2C	SYSTEM ERROR
48	30	ERASE EXIT
52	34	VALUE ERROR
56	38	WORKSPACE FULL
60	3C	RANGE ERROR
64	40	EMPTY EXIT
68	44	IMPLICIT ERROR: QUADCT
72	48	IMPLICIT ERROR: QUADIO

FIGURE 27: APL ERROR RETURN CODES

All the APL macros except APLCSL have a mode which defaults to zero. If the mode is zero, the IBM/370 registers should contain the information specified in the appropriate section of this manual. If a mode of one is used then only GPR3 need be as described. The FPR2 and GPR4, GPR5, ..., GPR2 values should be in SAVEREGS (memory locations GPR3-X'300' through GPR3-X'2BC') and the emulator will load these. On the corresponding exit (if any) to the next sequential instruction FPR2 and GPR4, GPR5, ..., GPR3 will be stored into SAVEREGS.

APLFIND

A block of free space of the indicated number of bytes (rightmost two bits will be forced to 00, must include the 12 necessary for the DN and two control words) will be found. Its space management control words and the N portion of its DN word will be completed. It will be classified as a temporary variable with an addressed value and its address table entry will be completed. The address of byte 0 of its DN word will be returned in GPR4 (byte 0 of GPR4 will be zero). The control word value (block length less 3) will be returned in GPR5. In the unallocated block this macro changes only the control words of the unallocated and result blocks and the DN word (D scratched, N fixed) of the result. Thus the system may build a result in the unallocated block, set/reset the rightmost bit of FREEU (see 'FREE SPACE') and then do the APLFIND.

Exceptions:	Specification Protection Operation
Errors:	Workspace Full Address Table/Stack Full Trap

APLFREE

This releases the free space associated with the named item unless it is an immediate and, in the case of temporaries, releases the name as well.

Exceptions: Specification
Protection
Operation

APLFRIF

This performs an APLFREE if the named item is a temporary. If it is a permanent then nothing is done.

Exceptions: Specification
Protection
Operation

APLNAME

The next available name will be removed from the unused list and returned in the right half word of GPR4; the left half word is unpredictable. The address table will be unchanged.

Exceptions: Specification
Protection
Operation

Errors: Address Table/Stack Full Trap

APLUNAM

The specified name will be restored to the list of unused names and the address table so marked.

Exceptions: Specification
Protection
Operation

APLSCAN

Scanning and execution of the workspace will commence at the address specified by the control word 'NEXTINST'.

Exceptions: Specification
Protection
Operation

Errors: Workspace Full
Address Table/Stack Full Trap
Syntax Error
Value Error
:
:
Etc. -- see figure 27

APLGETV LEFT
RIGHT

This gets a variable from the stack and sets it up for processing (see 'GETV'). For the left (right) variable GPR1 (7) must contain the stack word; the macro will setup GPR0-2 (6-8).

Exceptions: Specification
Protection
Operation

Errors: Syntax Error
Value

```
APLGETN         LEFT ,INIT ,LOG
                RIGHT FETCH INT
                      CVT   REAL
                            ASIS
```

This gets a number from a variable which has been set up by the emulator or APLGETV. The first call should be with 'INIT'; this will return the element count in GPR4 as well as the first number. Subsequent calls should be with 'FETCH' for each additional element. Cyclic fetching will be done automatically if the element count is one. If the user does his own initialization and fetching 'CVT' may be used for conversion only. In any case one requests the type of output desired: logical, integer, real, or 'ASIS', i.e., no conversion. GPR4 will be altered only by the 'INIT' option. (Note: The 'INIT' entry converts the first value which was fetched by APLGETV and sets up the next value address. Additional 'FETCH's maintain this next address. Thus, for example, to fetch real values from a (non-AP) integer vector in the reverse order, skip the 'INIT' call and point to the last element. Then the program loop can make a 'FETCH' call followed by a decrementing of the address by 8.

Exceptions: Specification
Protection
Operation

Errors: Domain Error
Range Error

APLRTN

This returns control from a normal external function to the APL emulator. See 'APL ASSIST AND THE APL SYSTEM'.

Exceptions: Specification
Protection
Operation

APLSRTN

This returns control from a special external function to the APL emulator. See 'APL ASSIST AND THE APL SYSTEM'.

Exceptions: Specification
Protection
Operation

APLRESM

This returns control from an interrupt or quantum end condition to the APL emulator.

Exceptions: Specification
Protection
Operation

APLDIAG

This macro was used during the development stage but it is not supported by the distributed version of the APL Assist.

APLCSL

The APL emulator comes in two parts. A skeletal part exists with the machine's 370 emulator. This part is essentially a routine for loading control store from main store. The second and main part is microassembled separately and is loaded into control store by the APL system using APLCSL. There are two ways to use this macro:

> If GPR4 is zero when the APLCSL is executed then no loading is attempted. However, the condition code is set to 0 or 1 to indicate whether the APL emulator is or is not already loaded.
>
> If GPR4 is non-zero: If the APL emulator is loaded then the condition code is simply set to 0. If the APL emulator is not loaded then the skeletal part of the emulator attempts to load the remainder of the emulator. GPR4 must point to an address in main memory which contains an encoded form of the APL emulator. GPR4 is updated as the loading progresses. If memory locations having the wrong format are encountered, then a data exception will occur.

Even with the APL Assist feature installed, the other APL macros will cause operation exceptions until control store is correctly loaded with this macro.

Exceptions: Specification
Protection
Operation
Data

370 EMULATOR/APL EMULATOR INTERFACE

An earlier section described the APL macros which provide the interface between the APL system and the APL emulator. This section is concerned with the interface between the two emulators.

APLEC ENTRY AND TERMINATION

The only way for the APL emulator to gain control of the CPU is for the 370 emulator to process the APLEC instruction. When the 370 emulator encounters an APLEC instruction, then the emulator does one of two things: (a) if the APL Assist is not installed on the machine then give an operation exception or (b) if the APL Assist is installed then activate it. When APL is activated it first checks the contents of CHKWRD; if this is incorrect then APL gives a 'specification' exception, otherwise APL proceeds with its requested task. The test of CHKWRD safeguards against APL activation as a result of a wild branch in some non-APL program. If the emulator is working with a virtual memory then the test of CHKWRD accomplishes another vital function: It insures that the control words page of the workspace is in real memory. If the page is not in real memory when the APLEC instruction is encountered, then a page fault results and the 370 supervisor takes the normal page fault action of swapping the page into memory and retrying the APLEC instruction. Page faults are discusssed in greater detail below.

On termination of the APLEC instruction, the APL Assist sets the condition code to zero for a normal exit or to one for an error exit. It then retrieves the address following the APLEC from SCANRTN or SERVRTN (see 'THE CONTROL WORDS') and sets it up as the 370 instruction location. Control then passes back to the 370 emulator.

PAGE FAULTS

The APL emulator must reference many memory locations during a single APLEC execution. It cannot anticipate possible page faults and force all pages to memory prior to real execution. Rather, page faults must merely cause execution to be suspended in a particular workspace until the required page is available. Meanwhile execution may continue in another workspace. The following paragraphs detail this process using 'page fault' to mean a real translation exception; mere refreshing of the associative registers is handled by a trap which is transparent to the APL emulator.

When the 370 page fault routine is activated, it tests for a 1401 emulation opcode and, if doing 1401, it branches to a different set of instructions. This routine has been altered to also test for an APLEC opcode and, if doing APL, it branches to the APL page fault routine.

The APL page fault routine compares the faulting control store address with that of the microinstruction which reads CHKWRD. If a match occurs it merely returns to the 370 page fault routine for normal 370 page fault processing. If there is a mismatch, then the APL emulator was actively working in a workspace and must be checkpointed. Local storage and the control store address of the faulting microinstruction are saved in SAVELS (this is possible since all the control words, including CHKWRD, reside within a single page) and the 370 instruction location register is set to point at the APLRESM in INTRTN. The faulting memory address is then loaded into an appropriate register and a branch is made to the instruction that reads CHKWRD. This causes a re-faulting that APL will allow 370 to process since it occurs on the 'read CHKWRD' microinstruction.

When the paging software has the required page available and attempts to re-execute the faulting 370 instruction it will actually execute the APLRESM. The execution of the APLRESM gives control to the APL emulator which will then continue with the workspace execution.

INTERRUPTS AND QUANTUM ENDS

As well as making many memory references, execution of an APLEC may require considerable time, at least in comparison to the execution times for most other 370 instructions. Thus the APL emulator must be able to pause periodically. This is done in a manner similar to the above page fault processing. If the hardware requests a pause for an interrupt, the APL emulator will checkpoint itself in SAVELS exactly as above, set the 370 instruction location counter to point at INTRTN and revert to IBM/370 mode. The workspace will be resumed later just as in the page fault case.

Such an interrupt might be caused by a time-out initiated by the 370 APL system. If so it will set on the 'quantum end desired' switch and cause resumption of the workspace by the APL emulator. As well as polling for interrupts, the APL emulator polls the quantum end switch. If this switch is on, the emulator will checkpoint itself in SAVELS, set the 370 instruction location register to the contents of QEND (i.e., it will point to the APL system's quantum end routine) and revert to IBM/370 mode. As above, the workspace can be resumed later, but to do so the APL system must explicitly execute an APLRESM.

USING EXTERNAL FUNCTIONS

The APL emulator is intended to co-reside with the 370 emulator and must therefore limit the amount of control store it uses. To meet this end it has been necessary to put some of the slower and less frequently used opcodes, such as domino, and some of the cases where we wish to share the 370 emulator's microcode, such as floating divide, in external code (assembly language or APL). Some of the less frequently used emulator features, such as extending the name table, have also been put in external code. The specific linkage conventions, etc., are discussed in 'APL ASSIST AND THE APL SYSTEM'; here we merely complete our description of the interface between the 370 and APL emulators.

All breakouts to external functions are processed through the 'call external function' emulator routine.

There are no emulator linkages or working storage values to be saved. This routine merely sets the 370 instruction location counter to point to the appropriate entry in the external function transfer vector using CALL370F and branches to the common exit portion of emulator. When the external function is complete it will issue an APLRTN. The 370 emulator will decode the APLEC and branch to the APL emulator which will then recognize the 'return' APLEC code and branch back to the 'call external function' routine. This routine then goes back to the appropriate place.

Some of the emulator features which are coded as external functions, such as address table extension, require the saving of emulator linkages and work registers. These checkpoint themselves in SAVELS, as in the page fault case, before doing the above steps. The corresponding external functions terminate with APLSRTN rather than APLRTN. This special return does not trickle back through the 'call external function' routine; rather, the checkpointed information is recovered and control passed directly back to the invoking emulator routine.

SUMMARY VIEWPOINTS

There are two major ways to look at the APLEC instruction. Each is given a paragraph description below. The first viewpoint is that seen from the 370 emulator. The second is that seen, or at least rationalized, by the APL system programmer.

The APLEC instruction is a slow conditional branch as far as the 370 emulator is concerned. It sees two cases: Sometimes APLEC causes control to pass to the APL emulator and after awhile control reverts to the 370 emulator with the 370 instruction location counter pointing at the instruction following the APLEC. At other times the return is accompanied by an instruction location counter pointing to some vastly different address such as INTRTN, c(QEND), or some point in the external function transfer vector. In both cases the time spent in the APL emulator is considerably longer than is spent executing a 370 'BC' instruction. The only way in which APLEC is different from all other 370 instructions is that it may be decoded at location xyz, but cause a page fault as if it had been decoded at location abc.

When the APL system programmer writes APLSCAN, he uses it like a normal 370 instruction whose execution will always be followed by the execution of the next sequential instruction. He may know that the APL emulator can temporarily breakout at a different point such as the external function for domino, but the external functions are logically viewed as mere extensions of the emulator. When the APLSCAN is complete, control will return to the next instruction.

STATEMENT SCAN AND SYNTAX ANALYSIS

At the beginning of the execution of an APL statement the stack contains

```
U   N   U   E   prior
```

where U denotes undefined, N denotes null, E denotes empty and 'prior' denotes whatever was on the stack before the current function was entered. The SCAN routine changes the stack to

```
U   N   E   E   prior
```

and then does the following:

LOOP: Get the next half word from the APL statement.
Increase NEXTINST by two.
Let H denote the half word just read.
Branch on the two low order bits of H.

BITS=00: H is a name.
Get its address table entry.
Put it on the stack.

BITS=01: H is an operator or separator.
Put it on the stack.

BITS=10: H begins a literal.
If it is a 16 bit literal, then ...
    Put it on the stack as a stack immediate.
Otherwise ...
    Get space in free space.
    Copy the constant and put its
    S-bits, P-bits and name on the stack.

BITS=11: H is a special operator.
These cases cause an immediate action. No further scanning is done. See 'OPERATORS AND SEPARATORS' for a description of the escape cases.

Having put the item on the stack (and thus erasing the undefined item at the top of the stack), let ST denote the syntax bits of the top item on the stack (syntactical types are shown in figure 29) and let SN denote the syntax bits of the next-to-top item. If DTAB[ST;SN] (see figure 28) is zero then push the contents of the stack as described in 'THE STACK' and go to LOOP. Otherwise do the action specified in figure 30.

End of statement processing (action 10) includes checking to see if printing is required and checking for stop, trace, attention and quantum end. If there is a temporary on the stack and no print or trace is requested and the last action was an assignment, then free the name and space used by the variable (unless it is a stack immediate).

The system uses syntax type F for group names. The emulator should never encounter these names, but if they do occur due to a user error then the emulator gives a 'syntax' error.

SN \ ST	1	2	3	4	5	6	7	8	9	A	B	C/D	F
0	1	0	1	0	1	0	1	1	5a	10	1	11	1
1	3	2	4	0	4	4	4	4	5a	4	4	11	1
2	0	1	0	1	8	0	0	9	1	10	5b	1	1
3	1	5c	1	0	1	1	1	1	5a	1	1	11	1
4	1	0	1	0	14	14	1	1	5a	1	1	11	1
5	12	6	1	0	1	1	1	1	5a	1	1	11	1
6	1	0	1	0	14	14	1	1	5a	10	1	11	1
7	1	7	1	13	1	1	1	1	1	1	1	11	1

FIGURE 28: DTAB[ST;SN] - THE SYNTAX DECISION TABLE

0	null
1	operator
2	variable
3	function of two arguments
4	) or]
5	(or [
6	;
7	←
8	right indexed-operator bracket
9	function of no agruments
A	end of statement
B	function of one argument
C	⎕ or ⍞ or shared variable
D	system variable
F	illegal (group)

FIGURE 29: SYNTACTICAL TYPES

0	Continue the scan.
1	Give a 'syntax' error.
2	Do a dyadic operation. The stack is left operand, operator, right operand.
3	Check for reduction and, if so, do it. Check for inner or outer product and, if so, encode the three operators into a single word (for example, +.x is encoded as the . operator with + and x in the low half of the word). If neither reduction nor product then do action number 4.
4	The stack is operator, operator, operand. Subtract two from NEXTINST and ignore the top stack word (the first operator). Do a monadic operation.
5a	The stack is function, ... Change it to undefined, function, undefined, ... Do action 5c.
5b	The stack is function, argument, ... Change it to undefined, function, argument, ... Do action 5c.
5c	The stack is A1 F A2, U F A1, or U F U where U is undefined, F is a function and Ai is a function argument. Do a function call.
6	Go to the subscript emulator routine.
7	Go to the assignment emulator routine.
8	If the top stack item is a left subscript bracket then simply continue the scan. Otherwise the top stack item is a '('. Erase it. The two items now on the top of the stack should be a value and ')'. If this is the case then erase the ')' and continue the scan. Otherwise give a 'syntax' error.
9	Change syntax type 8 to type 4 and continue the scan.
10	Do the end of statement processing.
11	Go to the shared variable external routine.
12	The top stack item is an indexed operator. Remove the index and brackets from the stack. Encode the index in 9 bits and store it in the stack word for the operator. Then continue the scan.
13	Mark the right bracket as a right bracket with an assignment arrow and continue the scan.
14	Put an empty subscript marker (6201 or 6205) on the stack and continue the scan.

FIGURE 30: TABLE OF ACTIONS SPECIFIED BY DTAB

Some of the dynamic properties of APL can give rise to some unusual problems, in particular a change of the syntax type of a variable may produce an error. The emulator insists on the following rule: if a name has a syntax type other than 2 then it must have a descriptor of type character. As an example, functions (see 'INTERNAL TEXT OF FUNCTIONS') are of type character. The GETV process (see 'GETV') includes a check of the syntax of all character items and it gives a 'syntax' error if the syntax type is not 2. Consider the execution of the statement 'Z←F+A', where F is a niladic function and A is a variable. The emulator puts entries for 'null', 'variable A', '+', and 'function F' on the stack and then it calls F. If the function F executes correctly and it has a result then the emulator will attempt to add A to the result of F. The addition will cause the emulator to do a GETV of A. If A is no longer a variable then a 'syntax' error results. The syntax of A could have changed because A was made into a shared variable, or because the user stopped the execution of F and changed A into a function or a group name. The address table entry for a shared variable does not point directly to the value of the variable. The method of storing shared variables is not defined by the emulator, but the block which the address table points to must be of type character.

FUNCTION INVOCATION

This section describes how function call and return affect the contents of the stack and it shows how the state indication can be found. An APL system displays the state indication when the command)SI is used.

FUNCTION CALL

Suppose the emulator is executing the statement

B ← (P F Q) + R

where P, Q, R are variables and F is a function of two arguments with the header information

∇ *V1*← *V2* *F* *V3*;*V4*;*V5*;*V6*

At the point where the emulator has read P, the stack will be

P F Q) + R null prior

where 'prior' denotes whatever was on the stack at the beginning of execution of this statement. P F Q and) are actually in the stack registers (see 'THE STACK') and '+' is the last item to be put into the memory stack. When we say that 'P' is on the stack, we of course refer to a full word item which contains the syntax bits and internal name of P according to the format described in 'THE STACK'. The emulator uses the header information of F, and it changes the stack contents to

U null E E K L A6 W6 ... A1 W1 C I) + R null prior

The top four items are in the stack registers and K is the last item in the memory stack. U is undefined, E is empty and the items) + R null prior are unchanged.

K = 0000 1111 uuuu uuuu kkkk kkkk kkkk kk00
where u = undefined and kk ... kk00 = decimal 40 + 8 times the number of local variables. (In this case there are three local variables and 40 + 24 is 64 decimal or 40 hexadecimal so the low half of K is 0040.)

L = 0000 1111 uuuu uuuu 0000 0000 0000 0010
(This is a special case of Wn and it marks the end of the W1 A1 W2 ... sequence.)

An = address table entry for variable Vn

Wn = 0000 1111 uuuu uuuu wwww wwww wwww ww00
where ww ... ww00 = internal name of variable Vn

C = 0000 1111 uuuu uuuu cccc cccc cccc cc00
where cc ... cc00 = internal name of function which contains the statement which calls F

I = 0000 1111 uuuu uuuu iiii iiii iiii iiii
where ii ... ii = offset of next byte of calling statement, which in this example is the offset of the (

The extension to functions with a greater or less number of local varibles should be obvious. For functions with no result then the A1 and W1 items still appear but A1 shows 27... (hexadecimal) and W1 shows 0FUU0001. Similarly with A2-W2 for monadic and niladic functions and with A3-W3 for niladic functions.

If Vn is the name of an item which is in free space then An contains the address of that item. Let x denote the address of the word An; let y denote the address in the low 24 bits of An. Before function call, the half word at location y+2 contains the internal name of Vn. During function call, we change this half word to x minus GPR3. This change of the contents of y is necessary for correct operation of garbage collection.

The emulator does the function call as follows:

1. Make a copy of P and Q and give the copies the names TMPNAM0 and TMPNAM1.

2. Check that sufficient space is available in the stack.

3. Stack the items I and C described above.

4. For n=1,2,... stack the items Wn and An described above. Set the address table entry for Vn to 'no value'. If Vn had syntax class other than D (system variable) then 'no value' is X'27UUUUUU'. If Vn did have syntax class D then 'no value' is X'D4UUUUUU'. If Vn is ⎕*CT* or ⎕*IO* then set the apporpriate 'implicit' error bit in SWITCHES (see figure 35).

5. Stack the items L and K described above.

6. Change the names of the items in TMPNAM0 and TMPNAM1 to V2 and V3.

7. Set stack registers R4, R3 and R2 to E, E and null.

Steps 1 and 6 make sure that V2 is given the value P. If P is a large vector or array then this means that P and V2 are made into synonyms. The emulator does, of course, process correctly those complicated cases in which P or Q or both P and Q have the same name as V1 or any other local variable.

The statement:

    U ← X G X

where G has the header

    ∇ X ← A G B

illustrates one of the more complex cases.

TEMPORARY FUNCTIONS

In APL\360 the user can type a single statement which receives immediate execution. The emulator requires that such single statements should be converted (by the translator part of the APL system) into functions. If the user types the statement

    A ← B + C

then the translator supplies a head and a tail and the emulator actually sees an internal representation of a niladic function having a temporary name. We will refer to this construct as a temporary function.

There are two other occasions when temporary functions are used. A statement such as

    P ← Q + ⍎X

where X is a character string with value 'A←B+C', requires that the character string X should be treated as an APL expression. This is implemented as follows: When the emulator sees the execute operator it calls the APL system. The system builds a temporary function, t, like the one above and returns the name of t to the emulator. The emulator now behaves as though the statement had been written

    P ← Q + t

and it calls t using the mechanism described in the previous section. Quad input is also implemented in this way.

The use of temporary functions is a simple but powerful way of unifying several different concepts in APL; for example multiple nested execute operations are easily handled in this way.

EXIT FROM PERMANENT FUNCTIONS

Consider a function F which contains N statements. The program will exit from F if any of the following statements occur:

- →
- → integer where the integer is >N or <1
- → expression where the expression reduces to an integer >N or <1
- Execution of statement N with no branch

The first case causes the emulator to signal a 'syntax' error; the system should trap the error return and do the appropriate action. The third case is similar to the second case. The fourth case is also similar to the second case because the translator always includes a fictitious '→0' after statement N. As cases two, three, and four are being executed the stack contains

→ V E E K L ...

where V is a constant or a variable, E denotes empty, and K L ... denotes the sequence described in 'FUNCTION CALL'. The first four items are in the stack registers and the K L ... are in memory. Assuming that V is a scalar (or a vector) and assuming that V (or the first element of V) is an integer less than one or greater than N, then the emulator frees the space used by V, if necessary, and then it does a function return.

FUNCTION RETURN

The contents of the stack registers can be ignored, so using the example in 'FUNCTION CALL' the stack is

K L A6 W6 ... A1 W1 C I) + R null prior

The emulator uses the value of K as an offset on the current stack address and picks up C and I. We said in 'FUNCTION CALL' that C begins with a zero bit; however, C may now begin with zero or one. After C was put on the stack, the user may have suspended execution and then erased the function named by C. Obviously it would be dangerous to return to a non-existent function, so when the erasure occurs, the APL system changes the first bit of C to one, and on detecting this case the emulator takes an 'ERASE' type error exit. If the first bit of C is zero then function return continues.

The emulator now goes through the stack and does:

a) Get Wn and hence get the name of Vn

b) If Vn has a value in free space then release this space

c) Get An and store it in the address table entry for Vn

d) If An points to an address in free space then store the name of Vn at that address plus two

There are two variations on this theme. Before doing steps a) through d), save the current value of V1, if any, because this is the result. Also, if Wn is an odd number then ignore subsequent steps because this is an empty slot corresponding to a no argument or no result.

The emulator now checks that the function has a result, and it gives the result the temporary name t, it sets the stack (and stack registers) to

U t U) + R null prior

restores NEXTINST (from I) and FUNCTION (from L) and returns to the part of the emulator which does statement scan and syntax analysis. If the function has no result then it puts X'27000050' on the stack (this is the stack entry for the control word NOVALUE).

RETURN FROM A TEMPORARY FUNCTION

There are three kinds of temporary function. These arise from: (a) immediate execution, (b) quad-input, or (c) the use of the execute primitive function. Let us consider case (a). If the statement used in immediate execution produces an explicit result then the emulator will return to the system with a 'PRINT EXIT' return code (see figure 27). If the statement does not produce a result then the normal return (condition code 0) will result.

There is one aspect of the 'PRINT EXIT' which requires special note. The 'PRINT EXIT' specifies that one or more items are on the stack and that printing may be required. However, if the top item on the stack is the result of a GOTO (see 'THE GOTO PRIMITIVE FUNCTION') then the system must take special action. In this case the next item on the stack should be a null (otherwise it is a 'value' error). The top of the stack in memory will be a word with the two low order bits:

10	BEGIN STACK marker
11	STOP WORD marker
01	originally 11, but the function has been erased

(The 'STATUS INDICATION' section discusses the 'BEGIN STACK' and 'STOP WORD' markers.) The system should free the temporary function and should handle the GOTO according to the rules of APL.

The emulator provides no special facilities for handling quad input or the execute function. As mentioned previously, the system can implement this case by building a temporary function. The system must turn on the trace bit in the temporary function so that it regains control at the end of the statement. The system should then remove the call of the temporary function from the stack, and then proceed as in the normal return from an external function. If the execute has no result then X'27000050' (the stack entry for the NOVALUE control word) should be put into GPR9. Note that the emulator will not branch on a GOTO within quad input or within the execute primitive. In these cases it will leave the result of the GOTO (see 'THE GOTO PRIMITIVE FUNCTION') on the stack for the system to handle.

STATUS INDICATION

The execution of an APL program can be terminated in several ways. Typical examples are (a) the program completes successfully or (b) the emulator detects an error or an exceptional condition such as 'WORKSPACE FULL' or (c) the emulator detects a stop bit at the beginning of a statement or (d) the user gives an attention. In all of these cases the current status is determined by the control words FUNCTION, NEXTINST, and TSADR, together with the contents of the stack. An APL system will display the status when the commands)SI and)SIV are used. The following sections describe how this status can be determined.

In this section the word stack will refer to the stack in memory; the contents of the stack registers are irrelevant. Items are placed on the stack in one of three ways: (1) The statement scan part of the emulator may use the stack for intermediate working. (2) The function call part of the emulator saves certain information which is described in a previous section. (3) If an error or stop is encountered then the APL system puts a stop word on the stack. Let us denote these three types of stack information as 'SCAN BLOCK', 'CALL BLOCK', and 'STOP WORD'. At the beginning of execution in a clear workspace, the stack contains just one word which is the 'BEGIN STACK' word.

Suppose the user types in a statement which the system embeds in a temporary function T1. Suppose T1 calls function F, statement 8 of F calls G and G has an error at statement 5. Suppose the user now types in another statement, which the system embeds in a temporary function T2. Suppose T2 calls function H and H has an error at statement 3. The stack contents and the APL status are

	<u>Stack</u>	<u>Comment</u>	<u>Status</u>
TSADR -->			
	STOP WORD		H[3] *
	CALL BLOCK	T2 calls H	
	SCAN BLOCK	T2[1]	
	STOP WORD		G[5] *
	CALL BLOCK	F calls G	F[8]
	SCAN BLOCK	F[8]	
	CALL BLOCK	T1 calls F	
	SCAN BLOCK	T1[1]	
	BEGIN STACK		

A STOP WORD has the form

STOP = 0000 1III IIII IIII NNNN NNNN NNNN NNP1

where NN...NN00 gives the internal name of the function in which the statement occurred and III...IIII gives the statement number. P is usually one but it gets set to zero if the function NN...NN00 is erased or edited in a way which damages the stack. If H has the internal name 007C then a stop at statement 3 would give the STOP WORD 0803007F hexadecimal.

A SCAN BLOCK can contain any item which the emulator will push into the stack (see 'THE STACK'). All of these items are single words of the form

SSSS ....

where SSSS can be 0000 through 0111. If SSSS is 0000 then the next four bits are always 0111 so that this case (which is the null item) has the form

NULL = 0000 0111 ....

The CALL BLOCK is described in a previous section, but notice that it always begins with a word of the form (item K of 'FUNCTION CALL')

CALL = 0000 1111 00

Finally the BEGIN STACK word has the form

BEGIN = 0000 1uuu u... uu10

where u stands for undefined; in practice the BEGIN STACK word is 08000002 hexadecimal.

The state of the stack on return from an APLSCAN entry to the emulator is as follows. If the emulator has just done a 'successful completion' exit then the top of the stack will be a STOP WORD or the BEGIN STACK word. If the emulator has just encountered a stop bit in a 'begin statement' then the top of the stack will be a CALL BLOCK. The system will then place a STOP WORD on the stack. If the emulator has just encountered an error then the top of the stack may be (a) part of a SCAN BLOCK or (b) the beginning of a CALL BLOCK or (c) a STOP WORD or (d) the BEGIN STACK word.

In order to analyze the contents of the stack, the system must start at the top of the stack. The address of this word is four more than the contents of TSADR. If the top of stack word begins with 00001 then it is a CALL WORD or STOP WORD or BEGIN STACK word. Otherwise it is part of a SCAN BLOCK. If the top of the stack is part of a SCAN BLOCK then it will erase this word (by increasing TSADR by 4) and repeat the analysis. When this analysis is complete then the top of the stack word has the form

```
0000 1... ...        ..xn
```

where xn=00 indicates a CALL BLOCK, 10 indicates the BEGIN STACK word, and 01 or 11 indicates a STOP WORD. If the top of the stack is a CALL BLOCK, then the system will add a STOP WORD to the stack; it will form this word from the contents of FUNCTION and NEXTINST.

To summarize the situation: Starting at the top of the stack it is possible to distinguish among STOP WORDs, words which begin CALL BLOCKs, the BEGIN STACK word and words which belong to SCAN BLOCKs. Having recognized a STOP WORD it is possible to determine the statement number and function name. Having recognized a beginning of a CALL BLOCK it is possible to skip over that BLOCK and to find the name of the calling function as well as the names and old values of all local variables.

APL ASSIST AND THE APL SYSTEM

In this section we describe the way in which the emulator and the APL system co-operate to provide all the APL language features which the APL programmer will see. As we have mentioned previously, the design of the emulator required a balance between the use of control store and the performance improvement to be gained by including particular functions in the emulator. We tried to achieve this balance by excluding certain functions from the emulator and by providing a facility for the emulator to call upon software implementations of these functions. The software may be in IBM/370 code or, with a few exceptions, it may be in APL code. Functions are excluded from the emulator if they use I/O (for example ⎕), if the emulator version would not be significantly faster than the software version (for example ⌹), or if the function is both complex and infrequently used (for example rotate with a non-scalar left argument).

EXTERNAL FUNCTIONS

The division between functions internal to the emulator and functions which the APL system should supply is as follows:

(a) Operations done completely by the emulator:

- statement scan and syntax analysis
- finding and freeing of temporary space
- function call and return
- plus, minus, negative, signum
- magnitude, maximum, minimum
- all logical operations and comparisons
- size, reshape, catenate, laminate, ravel
- index generator, index of, compress, expand
- membership, reverse
- goto, assignment, subscripting
- integer cases of times, residue, floor, ceiling
- integer cases of divide with no remainder
- raising to power 2

(b) Analysis and operand fetch/store in the emulator; operation on one element or one pair of elements is done in an external function (see 'SCALAR FUNCTIONS' later in this section):

logarithm, binomial, circular
real residue, factorial
roll, real multiply, real floor, real ceiling
divide and power except as noted in (a)

(c) Function not done by the emulator; complete function should be done by an external routine:

grade up, grade down, deal, domino
scan, format, translation part of execute, I-beam
shared and system variables, encode/decode
garbage collection

(d) In the remaining cases the emulator does the function for some operands and exits to external functions in other cases. Let 'true' vector/array denote a non-pseudo scalar vector/array (i.e., one with an element count not equal to 1).

transpose — uses the method described by Hassitt and Lyon <1>. The emulator exits to an external routine which should generate an appropriate subscript list. The emulator takes this list with the right argument of the transpose and generates the final result.

take, drop — requires an external routine if the left argument is a true vector and the right argument is either a pseudo scalar or a true array. All other cases are done by the emulator.

rotate — requires an external routine if the right argument is an array and the left argument has more than one element. All other cases are done by the emulator.

reduction — on scalars and non-AP vectors is done by the emulator. Reduction on AP vectors and arrays is done by an external routine.

products — inner product is done by the emulator if either argument is scalar or if both are vectors, otherwise an external routine must be used. Outer products are split in the same way as inner products, except that the vector-vector case requires an external routine if either argument is character.

The APL emulator occupies 18.5K bytes of control memory and can co-reside with an IBM/370 emulator which will fit in the remaining 47.5K bytes. We could conveniently have put the items in class (d) and the items in rows three and four of class (b) in the emulator but did not do so because of a lack of space. The use of IBM/370 code for items in rows one and two of class (b) is quite satisfactory and we can see no reason for microcoding them. Likewise, there is no reason to alter most of class (c). The grade operation might seem like a good candidate for the emulator but a superficial examination indicates that on the model 145 the emulator would not be appreciably faster. Scan, encode and decode might be faster in microcode, however there are very many special situations to consider and an efficient microcode implementation would require a large amount of control store. Garbage collection was microprogrammed but was removed due to limitations on control store size.

THE CALLING MECHANISM

The external functions are used in several ways but they are all called in the same way. There is a control word named CALL370F which contains an address which we will denote by C. Beginning at Location C, there is a transfer vector with one entry per external function. The transfer vector entries are shown in figure 31. Suppose the APL emulator is in control and it decides to call the dyadic I-beam external function. The emulator puts the arguments of the I-beam into the general purpose registers using the process specified in the section on 'GETV'. It sets GPR4 equal to C. It sets the 370 instruction location counter equal to C plus X'70' (according to the table the dyadic I-beam entry is at X'70') and reverts to IBM/370 mode. Location C+X'70' contains a branch to the 370 function which does the dyadic I-beam and it can use GPR4 as a base

Scalar functions:

00 multiply	04 ...	08 divide	0C ...
10 floor/ceil	14 factorial	18 roll	1C I roll
20 power	24 logarithm	28 circle	2C residue
30 binomial	34 deal		

Complete functions:

40 en/decode	44 grade	48 take*	4C drop*
50 reduce*	54 scan	58 inner prod*	5C outer prod*
60 m I-beam	64 m domino	68 m format	6C m box
70 d I-beam	74 d domino	78 d format	7C d box
80 share-in	84 share-out	88 share-post	8C execute
90 rotate*		98 m tranpose+	9C d tranpose+

Special functions:

38 extend the stack	3C extend the name table
94 garbage collection	

* Some cases are done completely in emulator, other cases exit to the software.

\+ Part of the operation is done in emulator, part is done in software.

m = monadic, d = dyadic

FIGURE 31: TRANSFER VECTOR FOR EXTERNAL FUNCTIONS

register. The external function computes the result, if any, and uses an APLRTN instruction to return control to the APL emulator. The transfer vector, the 370 functions and the APL functions are resident in the APL system; the functions are re-entrant and may be used by any number of users. With the exception of the exits for multiply and divide, before the APL emulator exits to the system for an external function, it stores FPR2 and GPR4-GPR3 in the workspace in the area named SAVEREGS (see 'APL SYSTEM/APL EMULATOR INTERFACE').

370 REGISTERS AND EXTERNAL FUNCTIONS

Most of the external functions will require one or two arguments as input and will produce a single result. The emulator uses certain GPR's and FPR's to communicate the arguments to the external functions and it expects the result in a specific GPR or FPR. General register assignments are delineated in figures 32 (GPR's) and 34 (FPR's). Register usage may vary a little during some of the emulator operations but the figures represent the normal state of affairs.

'GETV'

The information in the registers may be easier to appreciate if we describe the context in which it is generated. Consider the execution of a statement such as 'Z←L+R' where Z, L, and R are variables. The emulator will scan this statement until it has detected the 'L+R'. At this stage the stack registers (see 'THE STACK') will contain:

```
GPR1 = stack word for L
GPR9 = stack word for +
GPR7 = stack word for R
GPRE = null
```

The emulator now uses a process which we will refer to as GETV. The GETV process takes a stack word and gets the properties and initial value of an APL variable. The GETV process is used in all monadic and dyadic operations, in assignment and in subscripting. The results of the GETV

	0	1	2	3
0	LEFT GETV REGISTERS			
1				
2				
3	EMULATOR WORKSPACE BASE REGISTER			
4	ABEN LINKAGE AND MISC			
5	MISC			
6	RIGHT GETV REGISTERS			
7				
8				
9	OPCODE		RESULT NAME	
A	LINKAGE AND MISC			
B	RESERVED FOR THE APL SYSTEM			
C	MISC	RESULT CURRENT ADDRESS		
D	MASK AND MISC	RESULT ELEMENT COUNT AND MISC		
E	NEXT STACK WORD			
F	RESULT BYTE	RESULT DESCB1	MASK AND CODE	INDEX VALUE

FIGURE 32: NORMAL GPR ASSIGNMENTS

	0	1	2	3
0/6	UNDEFINED			
1/7	STACK WORD FOR A VARIABLE OR IMMEDIATE			
2/8	UNDEFINED			

FIGURE 33.1: GETV REGISTERS - INPUT

	0	1	2	3
0/6	VALUE UNLESS IT IS REAL			
1/7	PD DESCRIPTOR		NAME	
2/8	MASK	CURRENT ADDRESS		

NOTE: GETV DOES NOT ACTUALLY INITIALIZE MASK

FIGURE 33.2: GETV REGISTERS - OUTPUT

```
     +-----------+-----------+-----------+-----------+
     |     0     |     1     |     2     |     3     |
+----+-----------------------------------------------+
| 0  |LEFT VALUE IF IT IS REAL                       |
+----+                                               |
|    |                                               |
+----+-----------------------------------------------+
| 2  |SWITCHES   |UNALLOCATED BLOCK ADDRESS (FREEU)  |
+----+-----------------------------------------------+
|    |  0   4    |UNUSED     |NEXT AVAILABLE NAME    |
+----+-----------------------------------------------+
| 4  |RESULT VALUE IF REAL, LINKAGE AND MISC         |
+----+                                               |
|    |                                               |
+----+-----------------------------------------------+
| 6  |RIGHT VALUE IF IT IS REAL                      |
+----+                                               |
|    |                                               |
+----+-----------------------------------------------+
```

FIGURE 34: NORMAL FPR ASSIGNMENTS

BIT 0	RESERVED FOR THE SYSTEM
1	ATTN - STOP AT STATEMENT END
2	QUADCT HAS ILLEGAL VALUE
3	QUADIO HAS ILLEGAL VALUE
4	QUANTUM END DESIRED
5	DOING SERVICE FUNCTION
6	INDEX ORIGIN
7	RESERVED FOR THE EMULATOR*

* CURRENTLY UNUSED

FIGURE 35: SWITCH BIT ASSIGNMENTS

process affect the operation of a large part of the emulator and a significant part of the system.

The input to GETV is a stack word in either GPR1 or GPR7, and the output is as shown in figure 33. The 'value' referred to is the value of the variable, if it is a scalar, or the first element if it is a vector or an array. If the variable is real then the value will be in the corresponding floating point registers (FPR0 or FPR6). Logical values will be setup as full words so that they may be treated like integers. The value of a character variable is in the rightmost byte; the content of the other three bytes is not defined.

The PD DESCRIPTOR is the regular descriptor halfword (see 'VARIABLES IN FREE SPACE') with P-bits 5 and 6 (see 'THE ADDRESS TABLE') or'ed into the first byte (which is why those bits must be 00 in the descriptor). These P-bits identify 'addressed value' or 'immediate value' and 'temporary' or 'permanent' states. A stack immediate is given P-bits 11. The 'permanent' state is set so that the emulator will not attempt to release the name of the variable after it is used in the operation. There is no way to distinguish between stack immediates and address table immediates once they have been through GETV.

GETV will set the current address to point to the beginning of the value portion of the variable block (of the value block in the case of synonyms). In later stages of executing an operator this is usually the address of the element following the element currently given in the registers.

The MASK is not actually setup by GETV; it will be set later if a logical vector is being used.

RELOCATING THE WORKSPACE

The APL system may suspend execution on a workspace and swap the workspace onto secondary storage. At some later time the system may swap the workspace back into the main memory and resume execution. If the new main memory space begins at a different address from the old space then any absolute addresses must be relocated. All address table entries have a bit which shows if relocation is needed (see 'THE ADDRESS TABLE'). Some entries in the stack may need to be relocated (see 'FUNCTION INVOCATION'). There are no absolute addresses in the free space area. GPR3 and FPR2 contain absolute addresses and they must be relocated. GPR2, GPR8 and GPRC will be either unused or will contain absolute addresses; they should be relocated. If the last exit from the emulator was for an interrupt or for a 'special' function (extend the stack or name table, or garbage collection), then the workspace must not be relocated since the registers may be in a non-standard condition.

OTHER COMMENTS ON REGISTER USAGE

Two bytes (GPRD.0 and GPRF.2) are marked as being masks in figure 32. Both refer to a mask for a logical vector result. Some operators will use one byte, others will use the alternative byte. Never will both be in use as masks and frequently neither will be. GPRF.2 also serves as the external function return code byte (see 'APLRTN and APLSRTN').

The result byte (GPRF.0) is used to build up a byte of values prior to storing during some of the cases with logical vector results. The last byte of the result descriptor is usually kept in GPRF.1. The 0-origin index (or its ceiling if real) is kept in GPRF.3 during execution of indexable operators; the default value is given if a value was not explicitly specified.

Normally the first byte of QEND contains the SWITCHES byte. When the APL emulator has control, however, they are maintained in the first byte of FPR2. The individual switch assignments are given in figure 35. FPR2 is also described in 'FREE SPACE' (FREEU) and in 'THE ADDRESS TABLE' (NEXT AVAILABLE NAME).

SCALAR FUNCTIONS

Consider the execution of 'L d R' where L and R are variables and d is a scalar dyadic operator. The emulator does the steps described in the section on 'GETV' then it does the following:

```
check for character arguments

if L and R are both scalar, then
        do operation
        check for 16 bit result, if so, put on stack
        else get space, store result and put descriptor
                and name on stack
        go to DONE

if either L or R are non-scalar, then
        check that size of L conforms with size of R
        get space for result
        go to EXIT if null result
LOOP:   do operation on first elements of L and R
        store result
        go to exit if all elements have been done
        get next two elements and go to loop
EXIT:   put descriptor and name on stack

DONE:   free the space used by L and R if necessary
```

Actually there is another step which is not described above; if the results of integer arithmetic overflow, then we convert all existing results to floating point and continue in floating point mode. If the operation is plus, minus, less than, etc., then the 'do operation' step is done completely in the APL emulator. In the following cases we go to a external function using the calling mechanism described earlier:

power, log, real residue, binomial, circular

We also use the same calling mechanism for real divide and real multiply but in these cases the transfer vector entries reduce to:

```
DDR     4,6          and          MDR     4,6
APLRTN                            APLRTN
```

In these two cases the calling mechanism may seem somewhat elaborate but it ensures a clean interface between the APL and IBM/370 emulators. In all of these dyadic scalar cases we are calling an external function with two 32 bit or 64 bit arguments and we expect a 32 bit or 64 bit result. The APL emulator does all the analysis of the arguments, fetch of the operands one at a time, conversion, if necessary, storing of result and counting the number of operations that must be done. As the 'DDR 4,6' implies, if the left and right operands are real, they are in FPR4 and FPR6, and the result goes in FPR4. If the external function detects an error (for example, negative input to be logarithm routine) then it should go directly to the appropriate error exit in the APL system. The external functions may look at the descriptor bits (see the section on 'GETV') to determine the properties of the arguments.

The monadic operations, namely:

floor, ceiling, factorial and roll

use a similar process. There input is in FPR6 and, in the case of factorial and roll, the result should go in FPR4. The 'floor/ceil' routine does a floor or ceiling with an integer or real result according to the following rules:

Let X=GPR9 byte 1 bit 0
Y=GPR9 byte 1 bit 1
Z=GPRF byte 2 bit 0
A=contents of FPR6

if X=0 then set R equal to the ceiling of A, else set R equal to the floor of A

if Y=1 then put R into FPR4

if Y=0 and R can be expressed as a 32 bit integer, then put it in GPR5, else set Z=1 and put R in FPR4

An external function called 'I roll' is also required. It should set GPR5 equal to a random choice from iota N where N is the integer in GPR6.

COMPLETE 370 FUNCTIONS

In a case such as 'L d R' where d stands for dyadic I-beam, then the emulator goes through 'GETV' for L and R and then calls the 370 dyadic I-beam function immediately. The emulator has checked that L and R have a value and if they were functions or shared variables then it will have got their values, but it does no other checking. On entry to the external function the registers are as specified by 'GETV'. The external function should compute the result and put the stack entry for the result in GPR9. The result can be a stack entry for one of the following:

- null
- a stack immediate
- a temporary or permanent variable
- an APL function

All APL operations which can be used by the ordinary users must have a result. The null result may be required in the case of the execute function. In the first three cases, the APL emulator will free the space used by the arguments, if necessary, and continue with the statement scan and syntax analysis. The fourth case is discussed below. Monadic external functions are treated in the same way except that the 'left' argument will be missing and an immediate zero will have been substituted.

The emulator provides two additional ways of returning from a complete function; these are referred to as 'partial function' in the section on 'APLRTN and APLSRTN'. The first partial function return specifies that the result is the same as the right argument. The emulator will then free the left argument, if necessary, and properly adjust the stack before continuing with the execution. The second partial function return simplifies the external function for the monadic or dyadic tranpose function. The external function builds a subscript list (see reference <1>) and sets up:

GPR1	stack word for right argument
GPR7	stack word for subscript list
GPR9	x'82', right rank, UU, UU
GPRE	unchanged

The emulator then uses the subscripting routines to evaluate the result.

There are some functions, such as rotate, where the emulator does the simpler and more frequently used cases and where it calls an external function for the complex cases. In these cases, the external function may, if it chooses, evaluate the complex cases using an APL function. The APL function may express the complex use of the primitive function in terms of a simple use, where the simple use invokes the emulator's implementation of the function. For example, an APL function to do reduction on a matrix could utilize the emualtor to do reduction along a row or column of the matrix.

APL FUNCTIONS

Any of the complete functions (but not the scalar functions) may be written in either IBM/370 code or in APL or in both; the emulator does not care which is used and the system programmer can make the choice. Suppose the system programmer decides to write dyadic domino in APL. He writes the appropriate APL function, he converts it to hexadecimal, expresses it in the form of assembler DC (define constants) statements, and assembles it using the IBM/370 assembler. The resulting CSECT is loaded as part of the APL system. When the APL emulator detects the dyadic domino operator, then it calls the appropriate external function. That external function should get a temporary name, let us call it 't', in the user's workspace. It should set the address table entry for 't' to:

    3F address of CSECT for APL domino function

and should set GPR9 to

    3F uu internal name of t

and do an APLRTN. After the return the emulator will detect the syntax of '3' so it calls the dyadic APL function whose name is 't'. Notice that only one copy of the domino function exists, but it can be used by any number of users; the arguments for the function, the local variables, and the status are stored in the user's workspace. When the emulator returns from a function which has the immediate bit on (see 'THE ADDRESS TABLE' for immediate bit) then it frees that name.

APLRTN AND APLSRTN

The APLRTN instruction causes the following action. The APL emulator gets control, it checks the CHKWRD (see '370 EMULATOR/APL EMULATOR INTERFACE'), and then it looks at byte 2 of GPRF. It interprets that byte as indicating a return from ...

uuuu uu00	scalar dyadic function
uuuu uu01	scalar monadic function
uuuu 0010	complete function
uu01 0110	partial function (transpose - subscript)
uu11 0110	partial function (result is right)
uuuu 1010	share post
uuuu 1110	processing of system or shared local variable during function exit
uuuu 0011	share out or execute
uuuu 0111	share in or execute

In the cases which we have described so far the emulator will have set GPRF byte 2 before it exits to the 370 emulator so, except for the partial functions, the assist routines do not have to be aware of this byte.

All of the functions discussed thus far are normal functions. When they exit, all information defining the current status is contained in the registers and workspace as previously described. There are also a few special functions such as the routine to extend the stack. These special exits, like exits to service an interrupt, cannot be well planned for by the emulator and it is necessary to save additional status information such as the current values of some of the emulator work registers. These routines must return to the emulator using an APLSRTN which will function in a manner similar to APLRESM. These routines may not allow quantum end nor may they use any APL macro other than APLSRTN.

TREATMENT OF SHARED VARIABLES

The emulator does not initiate any input or output, but it does call the system whenever I/O is required. If the the end of an APL statement is reached, and the stack is not null, and the last operation was not an assign, then the emulator takes a 'print' exit from APLSCAN. Another type of I/O is initiated when the emulator detects a shared variable or the quad symbol. Let S represent the quad or quad prime symbol or a shared variable. When the emulator reads the S then it calls the external 'share-in' or 'share-out' function. At this stage the stack is in one of four possible states:

```
1)   S      ← ...
2)   S      x ...
3)   S[...] x ...
4)   S[...] ← ...
```

where x is any symbol other than '←'. S is in GPR1 (see 'THE STACK'), and GETV has not been done. Case one causes a 'share-out' exit; the other three cases cause a 'share-in' exit. In cases three and four, the system will have to search down the stack until it finds the first closing bracket and then it can distinguish between the two cases.

In case one, the third item on the stack (which is in GPR7) will be a variable. The system should check that the variable has a value and then transmit the appropriate information to the shared variable processor. The system should now move GPR7 into GPR9 and APLRTN. The contents of GPR1 and 7 will not be used. The emulator will have set GPRF byte 2 to 3 before exit.

Now let us consider case two where S is not quad. The system should do the input and store the result in the workspace. The system should form either a stack immediate (if the result is a scalar logical, character or short integer) or a stack entry for a temporary variable. In the latter case the system should have stored the result in the workspace. The stack entry should be put in GPR1 and an APLRTN should be given. The quad input case is similar to the ordinary input case, except that the system should form a temporary function which contains the internal text of the input. The system should put the stack entry for a niladic function in GPR1 and then APLRTN.

After the system has found the closing bracket and has distinguished between cases three and four then it should proceed as follows. For case three, simply proceed as in case two. In case four, the stack entry for the closing bracket was originally 40.... It is now 48... (see action 13 in 'STATEMENT SCAN AND SYNTAX ANALYSIS'). It should be changed to 4C.... Replace GPR1 by the stack entry for a permanent variable which contains the latest value of S, then APLRTN. If there are no errors (possible errors are 'value', 'domain', 'index' and 'workspace full'), the emulator will do the subscripted assign and then it will call the 'share-post' external routine. At this stage NEXTINST-2 will point to the name of the subscripted variable. The system should communicate whatever information is necessary to the shared variable processor, and then it should APLRTN. The emulator will also call the 'share-post' external function if it detects a shared variable during function exit. This case occurs if a local variable has been shared and not subsequently retracted. Note that this case may be distinguished from the subscripted assign use of share-post by inspecting byte 2 of GPRF.

EXECUTE

Suppose the emulator encounters ⍎X. The emulator will do a 'GETV' and call the external execute-operator function. That function should check that X is a legal character string, convert it to internal form, embed it in a temporary function 't' (see the section on 'TEMPORARY FUNCTIONS' in 'FUNCTION INVOCATION'), set GPR9 with the stack entry for 't' and return. The emulator now follows the actions specified above in 'APL FUNCTIONS'. 't' may call other functions, including, of course, a recursive call to the function which called 't'. The internal form of 't' should contain a trace bit at the end of statement one (there is only one statement). When the trace is reached, the emulator takes a normal trace exit. The system should save bit 4 of byte 2 of GPR4; let us denote this bit by 'a'. The stack contains the function call block for the call of 't' (see the 'FUNCTION CALL' part of 'FUNCTION INVOCATION'). The system should remove the call block from the stack and then:

If bit a=0 proceed as in shared input. Set GPRF byte 2 to 7, GPR1 equal to the result of the execute (if any) or null and GPR9, 7... equal to the previous contents of the stack. Give an APLRTN.

If bit a=1 then the execute ended with an assign and the emulator has to ensure that ⍎X in the context ...+⍎X... does produce a result whereas ⍎X in the context ⍎X,... does not cause printing. In this case, proceed with shared output, namely, set GPRF byte 2 to 3. GPR1 and 7 are undefined. GPR9 is the result of the execute. GPRE,... has the previous contents of the stack.

Obviously the procedure outlined in this section is not simple but it requires very little extra 370 code or microcode and it gives a very powerful facility.

ERROR RECOVERY

The use of the emulator may cause various errors to be detected. Some of these are pseudo error returns from the emulator requesting printing, etc. The real errors can be divided into several types:

1) Errors in the user's program or data.
2) APL 'system' error return.
3) Program exceptions such as 'specification'.
4) APL error returns other than type 1 and 2.

The first type of error will cause an error exit from the emulator with a return code showing the type of error. The APL system will presumably wish to recover from this error in the manner described below. The second and third type of error implies that the system or the emulator or the hardware has a bug. It will be necessary to dump the workspace and to trace the cause of the error; see 'DEBUGGING AIDS'. When this type of error is detected, then the workspace may contain unknown errors and execution on this workspace should not be continued. The fourth type of error is almost certainly due to a system program error and the cause should be easy to determine. Errors of type two, three and four should happen very infrequently.

Errors of type one may happen quite frequently and they are a normal part of APL execution. When these errors occur the system should print out an appropriate message and then clean up the stack. To clean up the stack the system should delete all memory-stack entries back to the nearest STOP WORD, BEGIN STACK word or function CALL BLOCK. If a deleted item is the stack entry for a temporary variable then the name and the free space associated with the name (if any) should be freed; there is an APL macro for doing this. If the item which remains at the top of the stack is a CALL BLOCK then the system should add a STOP WORD. The method of analyzing the contents of the stack is given in the section on 'STATUS INDICATION' (see 'FUNCTION INVOCATION'). The data needed for the STOP WORD is found in the workspace in FUNCTION, NEXTINST and in the tail of the current function. Part of the stack information is held in the general purpose registers. During the SCAN process, the registers will have the format described in 'THE STACK'; at other times the format of these registers will vary according to the operation being performed. When an error occurs the format of these registers is unclear. It is possible that these registers will contain the names of some temporary variables. The names and space used by variables must be released. These names cannot be determined from the registers, but they can be determined as follows: Search the address table for entries which belong to temporary variables. If such an entry is found then search all SCAN BLOCKs on the stack (see 'STATUS INDICATION' for a definition of SCAN BLOCK) for a use of the temporary variable. If no use is found then free the name and its associated block in free space (if any). This search requires the system to look at every address table entry however the search is fast and it only occurs when APL execution has terminated due to a user error.

There is another area which the error recovery procedure must check. The control word TMPNAM0 usually has the form 27...; after an error exit, if TMPNAM0 is 29... or 2B... then it should be changed to 27... and the space whose address is in the low 24 bits of TMPNAM0 must be freed. The same remarks apply to TMPNAM1. These control words are used to hold function arguments during the function call process. The arguments will have received permanent names by the time the function is entered and the control words will have been reset to 27.... TMPNAM0 may also be used to hold the result of a function during the function return processing.

If an error occurs in a locked function, a system function, or a temporary function used for quad prime or execute, then the system will need to take special action. These actions are not defined by the emulator.

DEBUGGING AIDS

Several debugging aids are discussed below and an example is then provided. Figure 36 summarizes much of the debugging aids information. These aids were of great value during the early development stages but are of much less importance now that the emulator is complete.

THE DEBUG EMULATOR ROUTINE

The debug routine is a section of microcode which is not distributed with the APL Assist; it was used during the development stages. If debug is part of an emulator coreload, then it may be either active or inactive. When active, it monitors all entries to and exits from the APL emulator and provides useful debugging information. It represents considerable entry/exit overhead and is normally inactive. If the debug routine is included in a emulator coreload, then after a normal IMPL debug is inactive. Let XX denote 'the debug module' of the coreload (the value of XX may be found by consulting the debug routine listing for the coreload). To activate the debug routine the following control words must be patched in as part of the IMPL procedure (i.e. in the patch deck) or later (i.e. using the console alter/display facility):

```
0000XX80  at  PLAM.CHECK.0X
0000XX81  at  PLAM.SAVGPR.01
0100XX80  at  PLPA.REFALT
```

Debug may be deactivated by patching these locations back to their assembled values.

The debug routine uses a 'debug box' stored in the workspace. The debug box is at the location GPR3-X'2D4'. The format of the debug box is given in figure 37.

WORKSPACE DISPLACEMENTS

R11	R03	CONTENTS	R11	R03	CONTENTS
0A0	---	SAVEREGS	3A0	00	TMPSAV
2D4	---	DEBUG BOX	3A8	+08	UNUSED
2F8	-A8	TIDYS	3BC	+1C	XARGO
2FC	-A4	UNUSED	3C0	+20	BLANK
300	-A0	FUZZER0	3C4	+24	ZEROVAR
304	-9C	FUZZER1	3C8	+28	ONE
308	-98	SEED	3CC	+2C	REAL1
30C	-94	UNUSED	3D0	+30	PI
310	-90	CALL370F	3D4	+34	E
314	-8C	QEND	3D8	+38	MIN
318	-88	SCANRTN	3DC	+3C	MAX
31C	-84	SERVRTN	3E0	+40	SYSTEM
320	-80	INTRTN	3E4	+44	NULNUMVC
324	-7C	SAVELS	3E8	+48	NULCHRVC
348	-58	PEMWORK	3EC	+4C	INDEX
390	-10	FREES	3F0	+50	NOVALUE
394	-0C	FREET	3F4	+54	TMPNAM
398	-08	CHKWRD	3FC	+5C	FUNCTION
39C	-04	FRSTRELO	400	+60	NEXTINST
3A0	00	TMPSAV	404	+64	TSADR
			408	+68	BNDATS
			40C	+6C	$CT
			410	+70	$IO

SAVELS FORMAT

324=LS14	334=W	340=LINK
328=LS15	338=V	344=SUTL
32C=LS16	33C=I	
330=LS17		

FIGURE 36.1: DEBUGGING SUMMARY SHEET

DEBUG BOX

SAVE	AX	LINK	
CP	ADSTOP	SLOT	
A0	10	00	00
A0	11	00	00
*C	*E	A0	*A
I0	I1	I2	I3
prior *C*EA0*A			
prior I0I1I2I3			
PFADR or ECNUM			

*E	= 00 if entry 05 if page fault exit 03 if other exit
*A	= APLEC code if *E is 00
*C:H	= ILC and CC if *E is not 00
I0	= PSW key
I1-3	= location
PFADR	= fault address in bytes 1-2 if *E is 05
ECNUM	= total emulator call number if *E is not 05

APLEC CODES (also + X'08')

00	SCAN	23	NAME	83	FREE	D3	GETV
01	RTN	43	UNAM	A3	FRIF	E3	DIAG
02	RESM	63	FIND	C3	GETN	F3	CSL

SPECIAL MICRO ADDRESSES

XX00	total emulator call number
XX04	emulator call count down number
XX08	DEBUG BOX WORD 4 R03 displacement
XX0C	return to I-cycles
XX10	one instruction stop-loop
XX80-XX88	DEBUG transfer vector

ALTER/DISPLAY USING MICRO-ADDRESS TRAP

1. ADR COMP CONTROL = STOP (down)
   ADDRESS COMPARE = CTR WORD ADR TRAP
   RATE = PROCESS
   DIALS = XX10wxyz (wxyz=micro address)
2. When trap occurs press ALTER/DISPLAY, etc.
3. When done you should be back in the loop at XX10. Set RATE to SINGLE CYCLE to check NREG. If it is not XX10 you are temporarily in a 370 trap: set RATE to PROCESS and push START, pause briefly and repeat this step.
4. Set ADDRESS COMPARE to CTR WORD ADR.
   Push CONTROL ADDRESS SET.
   Push START several times.
5. Repeat step 1 and push START.

FIGURE 36.2: DEBUGGING SUMMARY SHEET

```
                  +---------------+
WORD 0            |SAVE   AX  LINK|
WORD 1            |CP ADSTOP  SLOT|
WORD 2            |A0   10   00 00|
WORD 3            |A0   11   00 00|
WORD 4            |*C   *E   A0 *A|
WORD 5            |I0   I1   I2 I3|
WORD 6            |prior *C*E0B*A|
WORD 7            |prior I0I1I2I3|
WORD 8            |PFADR or ECNUM|
                  +---------------+
```

where ...

*E	00 if entry log 05 if page fault exit log 03 if other exit log
*A	APLEC code if *E is 00
*C:high	ILC and CC if *E is not 00
I0	key
I1-3	location
PFADR	fault address in bytes 1-2 (bytes 0 and 3 are junk) if *E is 05
ECNUM	total emulator call number if *E is not 05

FIGURE 37: DEBUG BOX FORMAT

When the debug routine is active and any call is made to the APL emulator, the debug box is updated and the APLEC in word 2 of the debug box is executed. Similarly any exit from the APL emulator will be filtered through the APLEC in word 3. This provides two convenient address stop locations for emulator/system tracing.

In figure 37 ECNUM is the current value of control store location XX00. At IMPL this location is set to zero and location XX04 is set to minus one. On each real emulator call (the pseudo calls out of the debug box are ignored) XX00 is incremented and XX04 is decremented. The emulator instruction STOP.0 in the debug routine will be executed only when the countdown word reaches zero. This mechanism is for use with bugs that occur deeply imbedded in an APL function. With only one APL user on the system the count-up word can be set to zero and the workspace run. When the bug occurs ECNUM determines a setting to key into the count-down word. Using the control store address stop switch it is now possible to rerun the workspace and stop on the entry during which the problem will show up. Then one can microstep, etc.

The debug package contains a 'return to I-cycles' at XX0C and a 'stop and branch to XX10' at XX10. The first of these is useful in recovering from an APL emulator disaster (such as a microloop). The second is useful in conjunction with alter/display and the control store address trap console feature (see figure 36). The debug package also contains the APLDIAG function, which allows dynamic control store modification and provides a skeleton loop for other uses, and code to allow the dynamic use of the control store address trap feature. These are very useful debugging tools, but they are also quite dangerous. Hence they are documented only in the debug (trap modification) and service macro (APLDIAG) emulator routines.

OTHER AIDS

In other sections of this document we have described the APL emulator and the emulator/system interface. We have deliberately avoided any specification of the system. This gives the system programmer greater flexibility in designing the system and allows him to change the system without modifying the emulator. In this section we have described the debug box which is in the system's part of the workspace. We further note that the system usually maintains FPR2 and the GPR's in SAVEREGS (see 'APL SYSTEM/APL EMULATOR INTERFACE'). We also give, in figure 36, the GPRB control word displacements and the following control words which belong to the system:

TIDYS	garbage collection count
SEED	for random numbers
PEMWORK	work area for system
FREES	start of free space
FREET	top of free space
FRSTRELO	first relocatable control word
INDEX	for an indexable operator

On exits for page faults, to take interrupts, or to allow quantum ends, local storage will be saved in SAVELS. The format is: LS14, LS15, LS16, LS17, W, V, I, LS13 (the linkage register), SUTL where SUTL is SPTL with U(3) replacing P.

In addition to the above aids there are the obvious things to look for in workspace dumps: what is NEXTINST and the APL statement to which it points? what is the current APL opcode (GPR9)? the common linkage registers (GPRA, FPR4, GPRD)? the error linkage register (GPR4)?

AN EXAMPLE

Figure 38 lists debugging information found in a workspace dump. We will begin to examine the dump as an illustration of debugging techniques. The actual control store addresses and microinstruction sequence numbers are obviously valid only for the APL emulator as assembled on the date of the dump.

We first note that the emulator last exited for a page fault (DEBUG BOX: THIS EXIT). The faulting address was 540XXX (DEBUG BOX: PFADR). This is outside the address space of the particular machine which was running (DUMP PSW confirms an addressing exception) so the emulator must have developed a bad address.

The microinstruction causing the page fault was at control store location 5BE8 (SAVELS: LINK) which is in the function call portion of the emulator. The microinstruction there is 'RDH LS17 ADJ,W+2' and we can see that W was indeed 0054006C (SAVELS: W).

The bad contents of W look suspiciously like the DN word for a character vector (D=0054). This can be quickly supported: GPR3 is added to 006C to find the address table entry for the variable with internal name 006C. This entry is 9B028034. Syntax code 9 is a niladic function and the internal form of all functions is 'character vector'. Indeed, when we look at location 028034 we find the 0054006C DN word. We now have a good handle on the bug which we suspect to be in the function call mechanism.

Some of the other debugging information which might have proved useful includes: The emulator was last entered by a return from the external function (DEBUG BOX: LAST ENTRY) having an APLRTN at address 016340-4 (DEBUG BOX: LAST LOC). There is a history of the assignment routine calling the 'GETV' processing code (LINKAGE REGISTERS: GPRA) and of the service macro routine calling the FREE entry point in the space management routines (LINKAGE REGISTERS: GPRD).

00000000 = (NOT USED)	DEBUG BOX
00000000 = (NOT USED)	
A0100000 = PSEUDO ENTRY	
A0110000 = PSEUDO EXIT	
4005A000 = THIS EXIT	
F0027322 = THIS LOC	
40000B01 = LAST ENTRY	
F0016340 = LAST LOC	
54540020 = PFADR	
0054006C = LS14	SAVELS
2B028044 = LS15	
0001002C = LS16	
00000003 = LS17	
0054006C = W	
F0027400 = V	
F0028030 = I	
03025BE8 = LINK	
9C013127 = SUTL	
7002359D = GPRA	LINKAGE REGISTERS
00000000 = FRP4	
38021900 = GPRD	
FF050005 40027322	DUMP PSW

FIGURE 38: EXAMPLE DUMP INFORMATION

SECTION IV: CONCLUSION AND REFERENCES

CONCLUSION

The architecture of the APL emulator and the design of the APL emulator-IBM/370 interface was completed in October 1970. The implementation was begun soon afterwards. Some design changes were made as the implementation progressed, but on the whole few changes were necessary. Shared variables were added at a later stage, but since the bulk of the shared variable processing is done by the software, few changes were required in the emulator. An initial and partial version of the emulator and an APL system were operational in November 1971. A complete emulator and APL system were running successfully and reliably in April 1972. This sytem was later developed into the APL/CMS system. The emulator has seen extensive use by APL programmers using the APL/CMS system. The APL emulator supports a major subset of the APL language (see 'APL ASSIST AND THE APL SYSTEM'). APL/CMS fully supports the language described in the APL\360 User's Manual <3> and the APL/CMS User's Manual <4>.

It is obvious that a project of this size and complexity will utilize the programs, techniques and co-operation of a number of people. The emulator was written in a microprogramming language designed by Daniel L. McNabb and John R. Walters, Jr. <2>. The use of this language played a major part in helping us to write and debug the APL emulator and we believe that it provides excellent documentation for the finished product. The debugging of the emulator was greatly simplified by an excellent and reliable assembler and simulator provided by the model 145 group in SDD, Endicott; we are also indebted to W. Decker, R. Dunbar, G. Kinsella and E. Wassel of that group for answering many questions about the workings of the hardware and the assembler and simulator. The authors of this report were responsible for the design of the architecture as well as the writing and debugging of the microcode for the APL emulator. The APLCSL emulator routine was developed by R. Losinger. The synonym technique resulted from some remarks by J. A. Brown.

The implementation and testing of the APL Assist feature would have been impossible without the software support provided by APL/CMS. M. J. Beniston was responsible for the design and implementation of a major part of APL/CMS including the software for the translator, the editor, and error recovery. The shared variable interface was designed by M. Carlitz. The remainder of APL/CMS (utilities, external routines, auxiliary processors, etc.) was written by J. W. Lageschulte, Hassitt, Lyon and N. S. Gussin. R. J. Creasy provided advice and encouragement throughout the project, wrote a number of the early versions of the APL system functions and solved one of our major problems by pointing out that the problem could not occur. We are grateful to W. B. Phelps for his work in detecting errors and delineating problem areas when the system was first put into productive use.

REFERENCES

<1> A. Hassitt and L. E. Lyon, "Efficient Evaluation of Array Subscripts of Arrays", IBM Journal of Research and Development, 16 No. 1, 45-57 (Jan 1972)

<2> Daniel L. McNabb and John R. Walters, Jr., MPL/145 A Language and Compiler for System/370 Model 145 Microprogramming, IBM Palo Alto Scientific Center report number ZZ20-6410 (May 1971)

<3> APL\360-OS and APL\360-DOS User's Manual, IBM form number GH20-0906

<4> APL/CMS User's Manual, IBM form number SC20-1846

<5> APL/CMS Installation Manual, IBM form number SC20-1845

Some other sources of related information are:

L. M. Breed and R. H. Lathwell, "Implementation of APL\360", Symposium on Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds ed., Academic Press, New York (1968)

This paper describes some of the details of the APL\360 workspace format.

A. Hassitt, J. W. Lageschulte and L. E. Lyon, "Implementation of a High Level Language Machine", Communications of the ACM, 16 No. 4 (April 1973)

This describes an earlier and different microcoded version of APL on the model 25. It shares many of the concepts and is somewhat more narative.

R. H. Lathwell and J. E. Mezei, A Formal Description of APL, IBM Philadelphia Scientific Center report number 320-3008 (Nov 1971)

This provides APL descriptions of some APL primitive functions.

An Introduction to Microprogramming, IBM form number GF20-0385

This gives a short introduction to microprogramming and includes some specific examples for an IBM/370 model 145.

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) :

A. Hassitt and L. E. Lyon

2. TITLE :

The APL Assist (RPQ S00256)

3. ORIGINATING DEPARTMENT

Palo Alto Scientific Center - 63G

4. REPORT NUMBER

ZZ20-6428

5a. NUMBER OF PAGES

115

5b. NUMBER OF REFERENCES

9

6a. DATE COMPLETED

10-30-74

6b. DATE OF INITIAL PRINTING

February 1975

6c. DATE OF LAST PRINTING

7. ABSTRACT :

The APL Assist is a hardware feature which enhances the performance of APL systems by providing direct execution of a major subset of the APL language. The feature can be installed on an IBM/370 model 145. The feature implements a new IBM/370 instruction called APLEC. The APL Assist does not modify any other IBM/370 instruction. The assist feature and the APLEC instruction may be used under standard operating systems such as VS and VM/370. APL execution is initiated by loading a general purpose register with the base address of an APL workspace and then issuing the APLEC instruction. This report defines the format of the APL workspace required by the assist feature, it gives the form of the APLEC instruction and it describes the results to be expected from using the instruction.

8. REMARKS :

9. SUBJECT INDEX TERMS

Microprogramming
Machine Language
APL
Performance
IBM System 370/145
07 - Computers
21 - Programming

IBM Cambridge Scientific Center
545 Technology Square
Cambridge, Massachusetts 02139

IBM Houston Scientific Center
6900 Fannin Street
Houston, Texas 77025

IBM Los Angeles Scientific Center
1930 Century Park W.
Los Angeles, California 90067

IBM Palo Alto Scientific Center
2670 Hanover Street
Palo Alto, California 94304

IBM Philadelphia Scientific Center
3401 Market Street
Philadelphia, Pennsylvania 19104