*** IBM CONFIDENTIAL ***

AN APL EMULATOR

A. HASSITT

L. E. LYON

IBM PALO ALTO SCIENTIFIC CENTER

JUNE 20, 1972

ABSTRACT

The APL emulator is a microprogrammed implementation of an APL processor. An APL processor provides direct execution of APL programs. This report defines the architecture of the APL processor and it gives the specifications of an integrated APL emulator. The APL emulator is said to be 'integrated' because it is co-resident with the IBM 370 emulator and it can be used under standard operating systems such as OS or CP/CMS.

This APL emulator is part of a system called APLM. APLM provides the user with all of the facilities of APL/360. APLM is fully operational. The APLM software is written in IBM 370 code and in APL. The APL emulator is written in IBM 370 model 145 microcode and will run only on a model 145.

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

APL/360 is an interactive time sharing system which provides interpretive execution of the APL language. Interpretive execution offers many advantages in producing a powerful, safe, and elegant programming language, but 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 statements using the machine language of existing computers. The only way of getting direct execution of APL is to construct a processor specifically for that purpose. Fortunately the reloadable control store, which is a feature of some models of the IBM 370, allows us to construct this APL processor by the use of microprogramming. This manual defines the architecture of an APL processor and it describes an emulator for that processor. The APL emulator is co-resident with the IBM 370 emulator and the manual describes the interaction between these emulators. If any processor is to be used effectively then it must be embedded in suitable software. The APL emulator runs under a software system which presents the user with all of the facilities of APL/360. We will use the name APLM to denote the system composed of the emulator and the associated software.

The first part of this manual will explain some of the novel aspects of APLM and will review the overall working of the system. The later parts will give a precise definition of the architecture and of the interface between the APL emulator and the 370 emulator.

2. THE APL MACHINE

The meaning of the expression 'IBM 7090 emulator' is quite obvious; an emulator is a hardware assisted simulator and an IBM 7090 is a machine which is specified by the IBM 7090 theory of operations manual. The meaning of the expression 'APL emulator' does not become apparent until we have described an APL machine. Consider first of all the use of an IBM 7090. It will involve the following steps.

(a) The programmer writes a program using symbolic instructions such as 'CLA I' or 'STO J'.

(b) The assembler allocates memory locations for the program and the variables I,J,... and it translates the program into an internal representation whose octal form is '050000 0 01001', '060100 0 01002', etc.

(c) The loader loads the program into memory, possibly relocates the addresses, and supplies library routines as required.

(d) Finally the loader issues an instruction which causes the first of the user's instructions to gain control of the machine. The machine now executes the instructions specified by the user.

In order to use an APL machine we follow some similar steps:

(a') The programmer writes his programs in the APL language which is described in the 'APL/360, Users Manual', form number GH20-0906. A typical statement is 'J<--K+L'.

(b') A translator converts all statements and functions into internal form. The translation process is as follows. Convert operators and separators into a 2 byte internal form. Convert all names into a two byte internal form; the first name encountered by the translator is translated into 006C (hexadecimal), the next is 0070, the next is 0074, and so on for successive multiples of four. The internal names 0000 through 0068 are reserved for system use. Having translated operators, names, and constants (see 'INTERNAL TEXT OF STATEMENTS' for translation of constants), reverse the order and add an end of statement marker. If '006C', '0070', and '0074' are the internal names of J, K, and L and if

'1021', '7001' and 'A001' are the internal representations of '+', '<--', and 'end of statement', then 'J<--K+L' has the internal representation '0074 1021 0070 7001 006C A001'.

(c') APL is most effective in a time sharing environment, so the 'loader' is typically part of an APL supervisor. The supervisor allocates a workspace to the user and places the internal representation of statements and functions into the workspace. A workspace is simply a contiguous block of memory which holds the programs and data of a single user. The supervisor and the machine will provide input-output, formatting, trigonometric functions, etc., so the automatic 'library routine' loading phase of the load does not occur. The user may load APL library functions at a later stage. The APL supervisor may swap a workspace in and out of main memory, and the user may perform various editing functions but eventually the user will ask for execution and the APL supervisor will bring the workspace into main memory (main memory could be a virtual main memory).

(d') The supervisor issues an instruction which causes the first of the user's statements to be executed. That first statement can, and usually will, invoke other user defined functions and so on.

When a new machine is being designed it is usual to define the machine language and then the symbolic or assembler language. In the case of the APL machine, the symbolic language was defined first and now we define (later in this manual) a machine language; it would of course be possible to define other APL machines with other internal representations. The steps a', b', c', d' described above are used by APLM but of course similar steps are used by APL/360; the essential difference is that step d' is carried out by an interpreter in the case of APL/360 and by a processor in the case of an APL machine. In our case the processor is microprogrammed (just as the IBM 370 model 145 is microprogrammed) but it could be all hardwired.

3. THE WORKSPACE ENVIRNOMENT

The IBM 370 machine works on a particular instruction at a time, but it also makes use of a global environment which is specified by the PSW, the contents of the main storage locations and the 370 registers (fixed point and floating). The APL machine works on a particular expression at a time, but it also makes extensive use of the current environment; the 'current environment' is the active workspace and the 370 registers. The active workspace contains the users programs, the values of all his defined variables, the current status, and the current execution stack. If the current statement is in a function which was called from another function which was called from another function and so on, then the stack contains all the information pertaining to these function calls; the information on the stack can be displayed by the APL commands)SI and)SIV.

The workspace is divided into four parts, namely

Control words
Address table
Stack
Free space

There is a register denoted by 'WORKBASE' which specifies the address of the beginning of the workspace. The control words area contains certain fixed constants as well as current status information. The address table varies in size. The address table entry at location WORKBASE+n is a word which describes the properties of the variable whose internal name is n; thus if WORKBASE contains 123400 and variable J has internal name '006c' then the word at location 12346C is the address table entry for J. 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 whether a variable has a value or not. If a variable has a value the value may be specified by D and V (scalar characters and logical or short integers are specified this way) or the value may be specified by the block of memory beginning at location A-4.

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 J is an array, its address table entry will point to a block of the form:

```
+----------------------------------------+
|  D   :   J  |           V              |
+----------------------------------------+
```

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

4. EXECUTION

'THE APL MACHINE' stated that the supervisor issues an instruction which causes the first of the user's statements to be executed. What actually happens is that the APL system has translated APL statements and functions into an internal form and has stored them in the free space area of the workspace, and it has set one of the control words with the address of the first statement. Execution begins when the APL system puts the address of this workspace in WORKBASE and issues the macro 'APLSCAN'. The 'APLSCAN' macro then causes the APL emulator to select the appropriate control word, find the address of the first statement, and begin execution.

The APL emulator directly executes statements such as the one illustrated in section 2 (a'), namely:

```
0074  1021  0070  006C  A001
```

The emulator obtains the first two bytes (7004) and examines the last two bits; in this case these are 00 which indicates an 'internal name'. The emulator forms WORKBASE+0074 and finds the appropriate S bits (see 'THE WORKSPACE ENVIRONMENT'). Assuming that the S bits show that 0074 is a variable and not a function, the emulator notes this fact and selects the next item. The low order bits of the next item (1021) indicate that it is an operator. The emulator now selects the 0070, finds its S bits, and, assuming that 0070 is a variable, the emulator starts to perform the addition of variable 0074 and variable 0070. The first action is to examine the P bits of 0074 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 '0070'. Next the emulator checks to see if 0070 and 0074 are scalars, vectors, or arrays, and that their sizes conform. It then decides on the type (integer or real) of the result, 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.

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

5. INTEGRATED EMULATION

The IBM 370 systems support a number of emulators. One of the outstanding features of these emulators is that they are integrated with the IBM 370 system; the IBM 1401 emulator on the IBM 370 Model 145, for example, is not only co-resident with the 370 emulator, but it also runs under control of the standard IBM 370 operating system. There are many advantages to integrated emulation; it is possible for one to schedule the jobs with the standard operating system and it is possible to use IBM 370 devices for 'IBM 1401' input/output. As a result, integrated emulation greatly extends the power, efficiency, and usefulness of the emulation process. The APL emulator is also an integrated emulator and, with suitable software support, it will run under any IBM operating system.

The implementation of an integrated APL emulator presents some unusual problems which arise because the APL machine architecture is so radically different from the IBM 370 architecture. The IBM 370 and the IBM 1401 (or the IBM 370 and the IBM 7090) are quite different from each other, but they do share certain basic properties: both machines use instructions, both machines use addresses, both machines recognize a limited number of operand types, both machines assume (with a very limited amount of checking) that the programmer will have decided which operator to use with which data (for example, integer add for integer data and floating add for floating data). The APL machine language has no instructions; rather, it has statements. The APL machine language has no addresses; it has names. The APL machine recognizes scalars, vectors, arrays of any size and shape. The APL machine applies an operator in many different ways according to the types of the operands. Despite the radical difference in architecture and despite many other problems, it has been possible to make an integrated APL emulator. Before considering the emulator, the basic mechanism of emulation on IBM 370 machines which have a reloadable control store should be examined.

Figure 5.1 shows a block diagram of a simple uni-programmed second generation computer. The processing unit is all hardware; the memory contains a single program. In figure 5.2 we show a diagram of a multi-processor. The hardware has one PSW; the memory contains several programs. One program is active at one time and its PSW is in the

```
+-----------+       +----------+          +---------------+
|PROCESSING|<----->|REGISTERS|     + - ->|PROGRAM        |
|UNIT       |       +----------+     |    |AND            |
|           |                        |    |DATA           |
|           |       +----------+     |    |               |
|           |<----->|PSW       | - - +    |               |
|           |       +----------+          |               |
|           |                             |               |
|           |                             |               |
|           |<--------------------------->|               |
|           |                             |               |
|           |                             |               |
|           |                             |               |
|           |                             |               |
+-----------+                             |               |
                                          |               |
                                          |               |
                                          |               |
                                          |               |
                                          |               |
                                          |               |
                                          +---------------+
```

FIGURE 5.1: A SIMPLE UNI-PROCESSOR

```
+-----------+       +----------+          +---------------+
|PROCESSING|<----->|REGISTERS|           |SAVE PSW       |
|UNIT       |       +----------+          |SAVE REGISTERS|
|           |                       + - ->|PROGRAM        |
|           |       +----------+    |     |AND            |
|           |<----->|PSW       | - -+     |DATA           |
|           |       +----------+          |---------------|
|           |                             |SAVE PSW       |
|           |                             |SAVE REGISTERS|
|           |<--------------------------->|PROGRAM        |
|           |                             |AND            |
|           |                             |DATA           |
|           |                             |---------------|
|           |                             |ETC            |
+-----------+                             |               |
                                          |               |
                                          |               |
                                          |               |
                                          |               |
                                          |               |
                                          |               |
                                          +---------------+
```

FIGURE 5.2: A MULTI-PROCESSOR

```
+-----------+          +---------+                 +--------------+
|PROCESSING|<----->|REGISTERS|                 |SAVE PSW      |
|UNIT      |          +---------+                 |SAVE REGISTERS|
|          |                           + - ->|PROGRAM       |
|          |          +---------+      |         |AND           |
|          |<----->|PSW      | - - +         |DATA          |
|          |          +---------+                 |--------------|
|          |                                      |SAVE PSW      |
|          |                                      |SAVE REGISTERS|
|          |<------------------------------>|PROGRAM       |
+-----------+                                     |AND           |
   A       A                                      |DATA          |
   |       |                                      |--------------|
   |       |                                      |ETC           |
   V       |                                      |              |
+-----+    |                                      |              |
|MPSW|    |                                      |              |
+-----+    |                                      |              |
   |       |                                      |              |
           |                                      |              |
   |       |                                      |              |
   V       V                                     +--------------+
+-----------+
|CONTROL   |
|STORE     |
|          |
|          |
+-----------+
```

FIGURE 5.3: A MICROPROGRAMMED MULTI-PROCESSOR

```
                +------------------------+
                |MICROPROGRAMS FOR:      |
                |      I/O UNIT CONTROL  |
+-----+         |      I/O INSTRUCTIONS  |
|MPSW|- - ->|      370 CPU EMULATOR  |
+-----+         |     1401 EMULATOR      |
                |      APL EMULATOR      |
                +------------------------+
```

FIGURE 5.4: CONTENTS OF CONTROL STORE

hardware PSW register. The other programs are dormant. When the active program is terminated then its PSW and register contents are saved in its save area and the PSW and registers of another program are loaded. Figure 5.3 shows a block diagram of an IBM 370 microprogrammed multi-processor. The processing is now controlled by a processing unit and a microprogram residing in a control store. The processing unit of figure 5.3 is much simpler, and performs far more primitive operations, than the processor of figure 5.1. Typically, the processor of figure 5.1 would contain floating point hardware, whereas the processor of figure 5.3 has no floating point hardware. The floating point operations are done by a series of microinstructions under the control of a microprogram. MPSW denotes the microprogram status word. In a system like the one shown in figure 5.3, the processor reflects the basic data formats (typically 8 bit bytes, 32 bit words, 24 bit addresses). On the other hand, the microprogram determines the instruction set of the machine. For further details see 'An Introduction to Microprogramming' (IBM form number GF20-0385).

The use of microprogramming allows the processing unit to support a wide variety of operations. An IBM 370 with the APL and 1401 emulators installed has a control store whose contents are shown in figure 5.4. The 'I/O UNIT CONTROL' microprogram performs the detailed control of certain I/O units, replacing the separate control units used in System 360. The 'I/O INSTRUCTIONS' microprogram decodes the 370 I/O instructions and commands, initiates I/O operations and posts the status of these operations. The '370 CPU EMULATOR' microcode emulates the IBM 370 non-I/O instructions. The MPSW in the diagram indicates that the 370 CPU microcode is in control, so the processor is currently executing the 370 instruction which is pointed to by the contents of the PSW.

It is obvious that the contents of the MPSW determine whether the machine is executing I/O control, an I/O instruction, a 370 instruction, a 1401 instruction, or an APL statement; the important question is how does the MPSW get set? Suppose the MPSW points to part of the 370 CPU microcode. If a control unit function is needed then there is a microcode trap, the MPSW is saved, the control function is done, the MPSW is restored, and the CPU microcode continues. There are a few other situations which cause a trap but in general the 370 microcode will retain control until the end of the current instruction (note that 'instruction' always means a 370 instruction; we will never use 'instruction' to refer to a microinstruction). At that point, if an interrupt is pending then the microcode will switch PSW's, but the 370 microcode still retains control. The 370 microcode reads the next instruction. If that instruction is a 370 CPU instruction the MPSW gets set to point to the 370 CPU microcode. If that instruction is 'APLEC' the MPSW gets set to the beginning of the APL emulator microcode. The APL emulator will retain control until it reaches the end of a program, needs some supervisor function, or detects an interrupt pending (see later sections). The APL microcode is also subject to the normal microcode traps. If the APL emulator detects a request for APL I/O it calls the APL supervisor. The emulator itself does not do any I/O. When the APL emulator decides to relinquish control, it sets the PSW to point to a 370 instruction and sets the MPSW to point to the beginning of the 370 instruction fetch microcode.

6. INTEGRATED EMULATION AND THE OPERATING SYSTEM

A further feature of integrated emulation is that all of the emulators will work under a single operating system. The APL emulator is independent of operating systems; it will work under DOS, OS, CP/CMS or any other system for which APLM software is provided. A single processor will typically have one APLM supervisor working under the overall supervision of OS. However, the emulator will support any number of users under any number of APLM supervisors and in a CP system it will support any number of virtual machines. As we mentioned earlier, the APL emulator is invoked by the 370 instruction 'APLEC XX' where XX is a hexadecimal code which selects various entries into the emulator. The APL emulator achieves its operating system independence by the following means: (a) from the point of view of OS (where OS stands for OS, DOS, CP/CMS, etc.) the APLEC instruction behaves like any other 370 instruction, (b) the emulator responds to interrupts in the standard 370 manner, (c) the emulator honors the write-protect keys on main memory, (d) the emulator is re-entrant; when the emulator gives up control it saves the current status in the workspace belonging to the current user, (e) the emulator uses the dynamic address translation hardware in the standard manner, (f) the APL emulator does no I/O operations. To summarize the situation: the APLEC instruction initiates a very complex action of APL emulation but from the point of view of the operating system, APLEC behaves like any other non-privileged 370 instruction.

7. THE APLM SYSTEM

The APLM system runs in an environment like the one shown in figures 5.3 and 5.4. The system could be written in IBM 370 code or APL code or a mixture of both. We decided that the APLM system should be written in IBM 370 code. The prime reasons for this decision were that we could make use of a large amount of existing code from the APL/360 system and also the development of the APLM system could go forward in parallel with the development of the emulator. The format of the APL/360 and APLM workspaces are quite different so any code which depends on the internal details of the workspace had to be completely re-written. All of the code concerned with handling the terminals, doing I/O, swapping workspace, scheduling users and so on, is virtually unchanged. In other words, the APLM system is the same as the APL/360 system except for minor changes in the supervisor and completely new code for the translator, the editor, the output format routine and the routines which do error recovery after a user or system error. The APL/360 interpreter is, of course, completely eliminated; calls to the interpreter are replaced by calls to the emulator. Some of the interpreter subroutines, such as the subroutine for domino (matrix inversion and least squares fit) are still required. These routines can be modifications of IBM 360 code routines or they can be written in APL. The APL emulator has the ability to call system routines which are written in IBM 370 code or APL machine code. The APL emulator will directly execute such APL routines without copying them into the workspace.

The overall environment of a typical APLM system is like the one shown in figure 5.3. Referring to that figure, the first partition in memory will typically contain an OS operating system. One of the other partitions will contain the APLM system routines and slots for several workspaces. The machine will function like a normal IBM 370. OS will time-share the CPU between several partitions. When the APLM partition is in control, then it will spend part of its time in normal IBM 370 mode controlling terminals, swapping workspaces, etc. There will come a time when APLM is in control and one particular workspace is ready for execution. APLM will bring the workspace into main memory, will set WORKBASE to point to the workspace and then will give the APLSCAN instruction. APLSCAN is a special IBM 370 instruction which alters the contents of MPSW (see figure

5.4) so that it now points to the APL emulator. The APL emulator does some checks and then, if all is well, it starts directly executing the APL code in the workspace.

Anyone who writes some IBM 370 code tends to assume that this code is executed in contiguous time steps. We know that in practice the code can be interrupted after every instruction (and in a few cases in the middle of an instruction) and that quite frequently the supervisor will suspend one task and go off to do another task. The APLM system can assume (with some limitations discussed below) that the APLSCAN instruction is executed in a single contiguous time step. In practice, the APL emulator does test for interrupts at frequent intervals and when they occur it saves the current status in the current workspace and hands control to the IBM 370 emulator and the OS supervisor. The OS supervisor will switch tasks in the normal way and carry out its normal function. Eventually OS will resume execution of the APLM task. When the APL emulator gave up control, it saved the location of the APLSCAN instruction and set the PSW to point to an APLRESM instruction. When OS resumes APLM execution it executes the APLRESM instruction which gives control back to the APL emulator which resumes work on the interrupted workspace. The one complicating aspect of this situation is that APL/360, and hence APLM, is unlike a normal user program. APLM does sometimes take control from the supervisor and it can, therefore, get control during an APL emulator interrupt. At this stage, the APLM supervisor may choose to do some I/O, however, the only actions it can take which affect the current workspaces are to set the quantum end flag or the attention flag or reset the resume PSW. The former actions will cause the APL emulator (when it gets control) to terminate the APLSCAN at the next convenient point. The reset PSW action is abnormal and should only be used if there seems to be a system error.

The microcode in the APL emulator does all 'store into memory' instructions using the protect key provided by the most recent APLEC instruction. The APLM system ensures that this protect key allows a store into the current workspace only. In the unlikely event of the system or the emulator making an error, then it could harm the current workspace but it cannot harm anything outside of that workspace.

We have so far discussed the typical situation of one APLM system under OS. The APL emulator is operating system independent. It could support several APLM systems under OS or any other operating system. There are difficulties in doing this, but the difficulties lie in the system, not in the emulator. The APL emulator can run under CP or CP/CMS and it has successfully run over twenty virtual machines, each with their own APLM system, all apparently active at the same time.

8. DEFINITION OF THE APL PROCESSOR

In this section we begin the task of providing a precise specification of the APL processor. We have a microprogrammed implementation of this processor, but we believe that the architecture described here would be well suited to a hardware or software implementation.

In order to define the processor, it would be very convenient if we could refer to an up to date and complete formal definition of the language. Since this definition is not available we assume that certain APL concepts are 'well defined' and that the reader will know the meaning of these concepts The current sources of information on APL are the 'APL/360 User's Manual' (IBM form number GH20-0906), the APL program product, and 'A Formal Definition of APL' (Philadelphia Scientific Center report number 320-3008 by R.H. Lathwell and J.E. Mezei). The processor also supports some new features of APL which are available in the experimental version of APL/360 produced by the Philadelphia Scientific Center. We assume that the following concepts are well defined:

1) Functions, statements, variables.
2) The APL character set and the external (e.g. the typewritten) form of the language.
3) The value of a variable.
4) The workspace.
5) The status of a workspace.
6) The method of displaying the current status of the workspace, and the value (or 'no value') of any variable.
7) For any given workspace which contains any given functions and variables, then the effects of 'executing' any given statement are known.

The external effects of the execution are that an error message may or may not result and the workspace status and the values of the variables will usually change. An APL system will contain three major parts:

The 'Translator' which translates statements and functions into internal form and puts them in the workspace.

The 'Processor' which operates on a workspace and which usually changes its contents.

The 'Display' which translates values and status into external form.

It is one of the virtues of APL that the user can never gain direct access of the internal representation. The user must phrase his input in terms of the external language, and he must interrogate the contents of the workspace by way of the display programs. An APL time sharing system will of course contain other major parts such as the scheduler, the swapping program, and so on, but these parts do not enter into this discussion.

The following sections of the manual will define the architecture of an APL processor. They describe the internal representations of the workspace, functions, statements, variables, etc. These definitions should enable the system programmer to write suitable Translator and Display programs. The definition of the actual APL processor is simply that:

If APL statements and functions are transformed by the Translator into the form described here,

and the APL processor is invoked,

and the Display program transforms the internal form of the resultant workspace into a suitable external form,

then the displayed results will conform with the 'well defined' results of APL.

It might be thought that this is just a round about way of saying that the APL processor is simply a microprogrammed version of APL/360, but this is not so. APL/360 uses certain specific methods to compute the results of APL execution, but there are many other methods which can be used. The processor can use any method it chooses as long as it produces the correct results; see the sections on 'AP VECTORS' and 'SYNONYMS' for some non-obvious methods of producing results. There are many different internal representations corresponding to any external representation. The processor can produce the result in any form as long as the displayed result is correct. To give one example, the vector 1 2 3 ... 100 will usually have an internal representation which takes 24 bytes of memory but it may use a representation which takes 416 bytes or 816 bytes or indeed any number of bytes.

9. 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 machine (that is the model 145 operating under 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 the job and of the workspace. The APL machine 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 machine 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 9.1. The system areas are used only by the APL system and are not discussed in this document (with minor exceptions: see 'DEBUGGING AIDS'). 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 user's variable values, 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 push down list used by the APL emulator. The control words contain status information, constants, save areas, and so on.

Free space is used from both the bottom and top as shown. When there is insufficient space left the emulator 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

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

FIGURE 9.1: WORKSPACE FORMAT

additional space, however, the boundary may be dynamically moved. Note especially that the stack grows from high core addresses to low core 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 core address.

10. 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 '370 REGISTERS AND 'GETV''). A map of the control words is given in figure 10.2; figure 10.1 gives a listing of the codes used in the map. Below is an alphabetic list of the control words and their definitions. The microcode 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.

BLANK	Address table entry for a 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 370 functions.
CHKWRD	Word used on entry to the emulator to check that a workspace is properly pointed to. This word contains X'3D8941BB'.
E	Address table entry for 2.718...

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
WRDS	-	Actual number of storage words
R11	-	Displacement from GPRB (not used by the emulator)
R03	-	Displacement from GPR3

FIGURE 10.1: CODES USED IN FIGURE 10.2

RELO	USED	CBYT	WRDS	R11	R03	CONTROL WORDS
A	E	V	1	2F8	-A8	TIDYS
A	B	V	3	2FC	-A4	FUZZCTL
A	E	V	1	308	-98	SEED
-	-	-	1	30C	-94	UNUSED
S	E	U	1	310	-90	CALL370F
S	B	V	1	314	-8C	QEND
S	E	U	1	318	-88	SCANRTN
S	E	U	1	31C	-84	SERVRTN
A	E	V	1	320	-80	INTRTN
X	E	V	9	324	-7C	SAVELS
X	E	V	9	348	-58	SAVELSB
X	E	V	9	36C	-34	SAVTDY
D	E	Z	1	390	-10	FREES
D	E	Z	1	394	-0C	FREET
A	E	V	1	398	-08	CHKWRD
D	S	Z	1	39C	-04	FRSTRELO
X	E	V	2	3A0	00	TMPSAV
-	-	-	5	3A8	+08	UNUSED
A	B	A	1	3BC	+1C	XARGO
A	E	A	1	3C0	+20	BLANK
A	E	A	1	3C4	+24	ZEROVAR
A	E	A	1	3C8	+28	ONE
$	E	A	1	3CC	+2C	REAL1
$	E	A	1	3D0	+30	PI
$	E	A	1	3D4	+34	E
$	E	A	1	3D8	+38	MIN
$	E	A	1	3DC	+3C	MAX
-	-	-	1	3E0	+40	UNUSED
$	E	A	1	3E4	+44	NULNUMVC
$	E	A	1	3E8	+48	NULCHRVC
A	B	A	1	3EC	+4C	INDEX
A	B	A	1	3F0	+50	FILL
R	E	A	2	3F4	+54	TMPNAM
D	B	A	1	3FC	+5C	FUNCTION
R	B	A	1	400	+60	NEXTINST
R	E	A	1	404	+64	TSADR
R	E	A	1	408	+68	BNDATS

FIGURE 10.2: CONTROL WORD MAP

FILL — Fill character to be used by APL coded system routines. Set by the emulator to 0 if the right argument is numeric or to blank if character.

FREES — Displacement of the start of free space plus 4 (ie, GPR3 plus FREES is the address of the word after the dummy block at the bottom of free space).

FREET — Displacement of the first word after the top of free space.

FRSTRELO — The displacement of the first control word to require relocation if the workspace is moved.

FUNCTION — The internal name of the current APL function.

FUZZCTL — Three words for fuzz control. The last two have zero at the fuzz bit; otherwise they are complementary. The first of these has zeros to the left of the fuzz bit and ones to the right. The first of the three words has bytes RS LM LO LF where ...

- LF — number of half bytes to the right of the half byte containing the fuzz bit
- LO — least exponent such that the (normalized) number may be unequal to zero
- Lm — Mask for testing the first half byte in cases where the exponent is LO (this is the half byte with the fuzz bit from the last of the three words)
- RS — reserved for the system (currently this happens to be the number of fuzz bits)

INTRTN — This contains an APLRESM macro. The emulator points the 370 instruction location counter at this when taking an interrupt.

INDEX	If X indexes an operator this contains the ceiling of X less the workspace origin.
MAX	Address table entry for the largest possible real number (X'7FF...').
MIN	Address table entry for the smallest possible real number (X'FF...').
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 by the system relocate routine.
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 '370 REGISTERS AND 'GETV''). Bytes 1-3 contain the address of the system quantum end routine.
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. The area format is given in the 'DEBUGGING AIDS' section.
SAVELSB	Backup area for SAVELS.
SAVTDY	Save area for the TIDY microcode routine.
SCANRTN	Location of the 370 instruction following the last APLSCAN.
SEED	Random number generator's seed value.

SERVRTN	Location of the 370 instruction following the last APLxxxx where xxxx specifies some service function (TIDY, FIND, etc).
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.
TIDYS	Garbage collection count. This word is incremented by one every time TIDY is invoked. Overflow is not tested for so negative values will follow the largest positive value and will eventually turn into positive values again.
TSADR	Address table entry for byte 0 of the next available word on the stack.
UNUSED	Currently unused.
XARGO	Extra argument (ie, 'global') for APL coded system functions.
ZEROVAR	Address table entry for 0 (logical).

11. 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 11.1 to 11.4.

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 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.

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 11.5. 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,

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

SSSS	Syntax (see figure 11.2)
PPPP	Primary descriptor (see figures 11.3 and 11.4)
A..A	Absolute (virtual) address of the named block
V..V	Value
DDDD	Type descriptor (0=logical, 1=integer, 4=character)
MUUU	Sign and unused

FIGURE 11.1: ADDRESS TABLE ENTRY FORMS

SSSS=0	Unused name
SSSS=2	Variable, non-shared
SSSS=3	Function, dyadic
SSSS=9	Function, niladic
SSSS=B	Function, monadic
SSSS=C	Variable, shared
SSSS=F	Group

FIGURE 11.2: 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 11.3: POSSIBLE ADDRESS TABLE PRIMARY DESCRIPTOR BITS

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	0=Not in use	1=In use

FIGURE 11.4: ADDRESS TABLE P-BIT ASSIGNMENTS

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.

```
                  +---------+
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 11.5: UNUSED NAME CHAIN EXAMPLE

12. THE STACK

12.1 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 extend the stack area
         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 to 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 microcode 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.

12.2 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 the 'STATEMENT SCAN AND SYNTAX ANALYSIS' section.

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 UUU1 UUUU UUUU NNNN NNNN NNNN NN00

where the U bits are unused, the N bits give the name 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 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.

13. FREE SPACE

Free space is divided into blocks of words. The first and last blocks of free space each consist of exactly one word containing the integer five. The reason for these two dummy blocks will be discussed later. First let us look at the basic items which may occur in free space (figure 13.1).

There is only one unallocated block. Whenever space must be found for an object, the required amount will be removed from the top or bottom (alternately) of this block. The first word of the block is pointed to by FREEU (see '370 REGISTERS AND 'GETV''). The rightmost bit of FREEU is 1/0 for the next space to be removed from the top/bottom. The bit before this bit may have any value, but it will not be preserved by the emulator.

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 specificly 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 directly plays with free space). The first and last dummy blocks in free space aid in this merging procedure; by having an odd space management control word (the first and last words of any block contain its space management control word) they look like active blocks and thus freeing the first or last real block does not have to be a special case for the merging routine to look out for. We thus see why the dummy words are used and why they contain an odd number. Now why is it five? When the garbage collector scans free space these blocks look like active blocks, but with zero bytes for the interior 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 13.1: 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 13.2: GENERAL FREE SPACE BLOCK

block. Since this cannot occur for a true free space block the routine detects the end of the scan.

With the exception of the two dummy blocks all of the above may be summarized by figure 13.2.

14. 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 14.1. 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. U...U denotes an undefined number of undefined words. This is 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-8 plus the contents of d-4. 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 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 14.2 to 14.4. 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 '04' for AP vectors (see 'AP VECTORS'). D1 uses bit 0 to flag these escape cases. D1 has bit 4 always off. However, when the microcode 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 micro-routine, etc. The descriptor bits of most interest are bits 123 and 567 of D1; these are well

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 14.1: 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 14.2: 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 14.3: 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 14.4: 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 14.5: EXAMPLES OF VARIABLES IN FREE SPACE

described by figures 14.2 and 14.3. Particularly useful is bit 1, the 'pseudo scalar' bit. If this bit is on the variable is null or has more that 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 14.5. Characters are shown in EBCDIC but the APL system may use a different code.

15. 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	04D1:NAME	FIRST	DELTA	NUM ELM	0000:0015

Thus the above examples would become:

```
00000015 04D1xxxx 00000001 00000001 00000003 00000015
00000015 04D1yyyy 0000000A 00000003 00000006 00000015
00000015 04D1zzzz 00000011 FFFFFFF2 00000004 00000015
```

The APL emulator does not examine all vectors to see if they can be represented as AP vectors. But the iota operator 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).

AP vectors permit many stunts such as allowing 'iota one million' to exist in a small workspace and such as being able to sum reduce it in very little time. Their real importance, however, is in subscripting. Programs frequently use subscripts of the form 'A+BxC' where C is an iota vector. AP vectors allow very efficient processing of these subscripts. They also, in conjunction with subscript lists, allow the subscripting microcode 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 and let IN be iota N. Then the emulator will evaluate

```
(TEXT=' ')/IN
```

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

16. 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 16.1 shows what happens for this assignment. T is a temporary name 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 then the synonym chain becomes A, B, D, C so that the links items become:

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 (currently this means that the space management control word is less than 64; see 'VARIABLES IN FREE SPACE') and one of the following is done:

B DF E	where DF is a dyadic function
MF B	where MF is a monadic 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

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 16.1: ADDRESS TABLE AND FREE SPACE ITEMS BEFORE AND AFTER A<--B (THE SPACE MANAGEMENT CONTROL WORDS HAVE BEEN OMITTED FOR SIMPLICITY)

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.

17. 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 (see 17.4) which have non-standard form.

17.1 Operators

Operator codes are shown in figures 17.1 and 17.2. 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 and unequal
- R=1 for left and 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. 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 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

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

↑
|
+--*SCALAR OPS*

MIXED OPS------→

	1	5	9	*D*
110	ρ	↑	ι	⊥
111	∘			
112	⌶		⍎	
113	$\boxed{S}$			$\boxed{M}$
115			⌽	⍋
118	⍉	↓	∊	⊤
11*A*	⌹		⍕	$\underline{*}$
11*B*	$\boxed{R}$			$\boxed{Q}$
11*D*			,	⍒
131			⊖	
155		/		
159	.			
15*D*		\		
171		⌿		
179		⍀		
*C*00	⎕	⍞		

FIGURE 17.1: 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	ρ	⍉	⍳	∊	↑	↓	⊥	⊤	*MIXED OPS*
0	1	∘	.	⌽	,	/	\	⍋	⍒	
1	0	⌶	⌹	⍎	⍕				*̲	
1	1	$\boxed{S}$	$\boxed{R}$					$\boxed{M}$	$\boxed{Q}$	

FIGURE 17.2: OPERATORS ARRANGED BY FUNCTIONAL GROUP

4001	)	5005	[	7001	←
4005	]	500*D*	⁅	8005	⁆
5001	(	6001	;	*A*0*X*1	*END*

FIGURE 17.3: SEPARATORS

exit to the APL supervisor (except for $\underline{*}$). The emulator does not define the properties of these operators. The encode and decode operators overstruck with o are the new operators format and execute. The operators denoted by a letter within a box are in the development stage. Both their external symbol and their definition are subject to change. The emulator implements them by invoking the 370 'box' functions (see 'FUNCTIONS IMPLEMENTED IN APL AND IBM 370 CODE'). The $\underline{*}$ is used to denote an opcode that cannot be entered into the workspace via the APL system. This opcode can only be patched in and it is used only by microprogrammers to test microcode routines before allowing a permanent usage of the routines.

17.2 Separators

The codes for the various separators are shown in figure 17.3. 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 a scalar, vector or the ravel or an array. (Like 8005 it cannot be typed by the user.)

17.3 Special Operators

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

0NNN NNNN NNNN 0011	go to N in a permanent function
1NNN NNNN NNNN 0011	go to N in a temporary function
UUUU UUUU UUUU 0111	make an 'escape' emulator exit
UUUU UUUU UUU0 1011	perform an indirect operation
VVVV VVVV VVVV 1111	secondary decode

The purpose of the 'escape' operation is not defined by the emulator. In fact the system 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 is used by some APL system routines. 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. 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 IBM 370 function corresponding to '11BD'. The 370 function will find the VV...11 in the low half of GPR9 and it can use it to select one of many sub-operations.

IBM

18. 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 COMM 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 (currently this is 1 for a locked function, 2 for a function generated by quad and 4 for a function generated by execute)
- 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 Ith local variable
- 2 marker for the end of the locals list
- EZ marker for the end of statement 0

Note that since the two low 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 SK is the internal text of statement K (see 'INTERNAL TEXT OF STATEMENTS'). If the statement is a comment then SK is absent. EK marks the end of statement K. It contains the trace bit for statement K and the stop bit for statement K+1. X is an 'immediate go to 0'. Further details of EK and X are given in the 'OPERATORS AND SEPARATORS' section. The TAIL contains the byte offsets of EZ, E1, E2, ... EM as half word items. COMM contains system information such as label names, the comments, and so on. The emulator is not concerned with the details of COMM. NB is the number of bytes in the HEAD, BODY, TAIL and COMM. Figure 18.1 provides an example of a translated function.

THE APL FUNCTION ...

```
Z <-- A F B;C;D;E

Z <-- A+B
⍝ COMMENT ON THIS LINE
C <-- D*E
Z;C;A
```

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

```
00000061  00540074  00040046  00000040
006C0070  0078007C  00800084  0002A021
00781021  00707001  006CA001  A0010084
10310080  7001007C  A0310070  6001007C
6001006C  A0010003  A001001A  00260028
00340040  00010002  048B0000  00000054
00000061
```

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

C	0000	0061			
DN	0054	0074			
M	0004				
T	0046				
S	0000				
K	0040				
Z=Z	006C				
L=A	0070				
R=B	0078				
L1=C	007C				
L2=D	0080				
L3=E	0084				
TAIL	001A	0026	0028	0034	0040
COMM	0001	0002	048B	0000	
NB	0000	0054			

FIGURE 18.1: INTERNAL FUNCTION TEXT EXAMPLE

19. INTERNAL TEXT OF STATEMENTS

19.1 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 should not occur in the body of a function. A comment statement should be replaced by a null statement; a null statement consists of an end of statement marker. The system may store the text of the comment in the COMM region of the function (see 'INTERNAL TEXT OF FUNCTIONS'). (The current system gets a block in free space and an internal name for each comment. These names are stored in COMM.)

Labels:

Labels should not occur in the body of a function. The system may store labels in the COMM region of the function. Also see 19.2.

Names:

An external name is represented by an internal name. An internal name is a 16 bit number ending with two zero bits. 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 CCUU VV...
     or    DDDD DDDD UUUL 1010 CCCC CCUU 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 19.2). 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. The 16 bit form is used for logical, character and short integer scalars. In the latter case M is the sign bit. 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 is used for vectors. It could also be used for arrays although the current system does not do this. In this case VV... must begin on a full word boundary (hence the two forms shown) and VV... must be of the form shown 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 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
```

19.2 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 user may, of course, use labels in any legal manner.

20. 370 REGISTERS AND 'GETV'

Much of the information specifying the current status of the workspace is maintained in the 370 registers. General register assignments are delineated in figures 20.1 (GPR's) and 20.3 (FPR's). Register usage may vary a little during some of the microcode routines but the figures represent the normal state of affairs.

20.1 'GETV'

Consider the execution of a statement such as 'Z<-L+R' where Z, L, and R are variables. The SCAN microcode 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 SCAN microcode now calls the microcode which does dyadic operations. The dyadic operations microcode does a GETV on L (that is, it calls the GETV microcode with GPR1 as input) and a GETV on R. GETV is used in all monadic and dyadic operations, in assignment and in subscripting. The results of GETV effect the operation of a large part of the emulator and a significant part of the system. GPR's 0-2 and 6-8 are devoted to the left and right arguments and are passed to GETV to fetch the first value and to change the variable stack word into the appropriate register setup. Figure 20.2 shows this setup.

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, but for character variables only the rightmost byte of the value register is 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

```
     +-------------+-------------+-------------+-------------+
     |      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  |PRESERVED 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 20.1: NORMAL GPR ASSIGNMENTS

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

FIGURE 20.2.1: GETV REGISTERS - INPUT

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

FIGURE 20.2.2: GETV REGISTERS - OUTPUT

```
     +-----------+-----------+-----------+-----------+
     |     0     |     1     |     2     |     3     |
 +---+-----------------------------------------------+
 | 0 |LEFT VALUE IF IT IS REAL                       |
 +---+                                               |
 | 1 |                                               |
 +---+-----------------------------------------------+
 | 2 |SWITCHES   |UNALLOCATED BLOCK ADDRESS (FREEU)  |
 +---+-----------------------------------------------+
 | 3 |   0    4  |UNUSED=00  |NEXT AVAILABLE NAME    |
 +---+-----------------------------------------------+
 | 4 |RESULT VALUE IF REAL, LINKAGE AND MISC         |
 +---+                                               |
 | 5 |                                               |
 +---+-----------------------------------------------+
 | 6 |RIGHT VALUE IF IT IS REAL                      |
 +---+                                               |
 | 7 |                                               |
 +---+-----------------------------------------------+
```

FIGURE 20.3: NORMAL FPR ASSIGNMENTS

```
BIT 0    RESERVED FOR THE SYSTEM
    1    ATTN - STOP AT STATEMENT END
    2    Y SWITCH*
    3    Z SWITCH* (CURRENTLY UNUSED)
    4    QUANTUM END DESIRED
    5    DOING SERVICE FUNCTION
    6    INDEX ORIGIN
    7    SET TO 1 BY TIDY (OTHERWISE UNUSED)

         * USUALLY 0; MAY BE TEMPORARILY
           SET TO 1 BY THE MICROCODE
           (EG, SEE THE GETN ROUTINE)
```

FIGURE 20.4: SWITCH BIT ASSIGNMENTS

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 microcode will not attempt to release the name of the variable after it is used in the operation. Thus one can count on the NAME being good only if the variable is not an immediate. 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; other processing microcode will set it up if a logical vector is being used.

The GETV function is available to the APL system through the APLGETV macro.

20.2 Other Comments

Two bytes (GPRD.0 and GPRF.2) are marked as being masks in figure 20.1. 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 370 function return code byte (see 'FUNCTIONS IMPLEMENTED IN APL AND IBM 370 CODE').

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 an explicit value was not specified.

Normally the first byte of QEND contains the SWITCHES byte. When the APL microcode has control, however, they are maintained in the first byte of FPR2. The individual switch assignments are given in figure 20.4. FPR2 is also described in 'FREE SPACE' (FREEU) and in 'THE ADDRESS TABLE'

(NEXT AVAILABLE NAME).

21. 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 370 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 RR instruction with opcode 0B. 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 21.1 summarizes the APL macros and figure 21.2 gives the BAL definitions. The remainder of this section discusses each macro in the order given in figure 21.1. 'Exceptions' may be real program exceptions (Specification, Data) or an APL error return signaled by a condition code of 1 and an error code in GPR5 (all others).

APLFIND

A block of free space of the indicated number of bytes 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. Note that the number of bytes must include the 12 necessary for the DN and two control words.

Exceptions: Specification
Workspace Full
Address Table/Stack Full

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
TIDY	none	none	perform garbage collection
SCAN	none	none	scan/execute a workspace
GETV	see text	see text	get a stacked variable
GETN	see text	see text	get a variable number
RTN	none	not applic	normal 370 function return
SRTN	none	not applic	special 370 function return
RESM	none	not applic	resume interrupted workspace
DIAG	see text	see text	diagnostic function

FIGURE 21.1: SUMMARY OF THE VARIOUS APLXXXX MACROS

```
         MACRO
&L       APLEC &CODE
&L       DC    Y(X'0B00'+&CODE)
         MEND

         MACRO
&L       APLFIND
&L       APLEC X'63'
         MEND

         MACRO
&L       APLFREE
&L       APLEC X'83'
         MEND

         MACRO
&L       APLFRIF
&L       APLEC X'A3'
         MEND

         MACRO
&L       APLNAME
&L       APLEC X'23'
         MEND

         MACRO
&L       APLUNAM
&L       APLEC X'43'
         MEND

         MACRO
&L       APLTIDY
&L       APLEC X'03'
         MEND

         MACRO
&L       APLSCAN
&L       APLEC X'00'
         MEND
```

FIGURE 21.2.1: BAL DEFINITIONS FOR APL MACROS

```
         MACRO
&L       APLGETV &VAR
         LCLA  &VARC
&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   ANOP
&L       LA    5,&VARC
         APLEC X'D3'
         MEND

         MACRO
&L       APLRTN
&L       APLEC X'01'
         MEND

         MACRO
&L       APLSRTN
&L       APLEC X'02'
         MEND

         MACRO
&L       APLRESM
&L       APLEC X'02'
         MEND

         MACRO
&L       APLDIAG
&L       APLEC X'E3'
         MEND
```

FIGURE 21.2.2: BAL DEFINITIONS FOR APL MACROS

```
         MACRO
&L       APLGETN &VAR,&ENTRY,&TYPE
         LCLA  &VARC,&ENTRYC,&TYPEC,&ARG
&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)
&L       LA    5,&ARG
         APLEC X'C3'
         MEND
```

FIGURE 21.2.3: BAL DEFINITIONS FOR APL MACROS

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

APLFRIF

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

Exceptions: Specification

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
Address Table/Stack Full

APLUNAM

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

Exceptions: Specification

APLTIDY

A garbage collection will be done and all relevant pointers (FREEU, various address table and stack entries, etc) corrected. The GPR variable addresses (GPR2, 8 and C) will also be maintained if accurate, but they will be scratched if not. For example the 370 dominoe function may do an APLTIDY. If it is the dyadic case both the left and right variable GPR's will be maintained, but if it is the monadic case the left variable GPR's are unpredictable.

Exceptions: Specification
Data (see 'DEBUGGING AIDS')

APLSCAN

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

Exceptions: Specification
Workspace Full
Stop Vector Request
Value Error
:
:
Etc. -- See the emulator routine 'ABEN' for a complete list of error exits and their codes.

APLGETV LEFT
RIGHT

This gets a variable from the stack and sets it up for processing (see '370 REGISTERS AND 'GETV''). For the left (right) variable GPR1 (7) must contain the stack word; the macro will setup GPR0-2 (6-8).

Exceptions: Specification
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.

Exceptions: Specification
Domain Error
Range Error

APLRTN

This returns control from a normal 370 function to the APL emulator.

Exceptions: Specification

APLSRTN

This returns control from a special 370 function to the APL emulator.

Exceptions: Specification

APLRESM

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

Exceptions: Specification

APLDIAG

This macro is for use only by microprogrammers. It is a debugging and diagnostic aid. It is documented only in the source listings for the APLDIAG decode point in the SERV microcode routine.

22. 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 ...
    P ut it on the stack as a stack immediate.
Otherwise ...
    Get space in free space.
    Copy the constant.
    Put its S-bits, P-bits and name on the stack.

BITS=11: H is an escape case.
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 22.2) and let SN denote the syntax bits of the next-to-top item. If DTAB<ST;SN> (see figure 22.1) 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 22.3.

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.

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 microcode checks 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 '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 even if the value is arithmetic.

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

FIGURE 22.1: DTAB<ST;SN> - THE SYNTAX DECISION TABLE

0	null
1	operator
2	variable
3	function of two arguments
4	right parenthesis or right subscript bracket
5	left parenthesis or left bracket
6	semi-colon
7	assignment
8	right indexed-operator bracket
9	function of no agruments
A	end of statement
B	function of one argument
C	quad, quote-quad or shared variable
F	illegal (group)

FIGURE 22.2: 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 AN is a function argument. Do a function call.
6	Go to the subscript microcode.
7	Go to the assignment microcode.
8	If the top stack item is a '(' then erase it and the corresponding ')' and pull the stack up. The alternative is that the item is a left subscripting bracket in which case merely continue the scan.
9	Change syntax type 8 to type 4 and continue the scan.
10	Do the end of statement processing.
11	Call the APL supervisor shared variable 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 left bracket as a left 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 22.3: TABLE OF ACTIONS SPECIFIED BY DTAB

23. 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. The state indication can be displayed by use of the APL command)SI.

23.1 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 SCAN microcode has read the P then 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 microcode 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 = 0010  1111  uuuu  uuuu  wwww  wwww  wwww  ww00
     where ww ... ww00 = internal name of variable Vn

C  = 0000  1010  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 workspace relocation and garbage collection.

The function call microcode, sets the address table entries of V1, V2, V3, V4, ... to no value and then it gives V2 the value P and it gives V3 the value Q. If P is a large vector or array the 'giving' V2 the value P 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.

23.2 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 a function. 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 + e X

where X is a character string with value 'A<--B+C' and e is the execute operator, 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 then 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.

23.3 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 less than 1 or greater than N
--> 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.

23.4 Function Return

The contents of the stack registers can be ignored, so using the example of 23.1 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 section 23.1 that C begins with a zero bit; however, C may now begin with zero or 1. After C has been put on the stack then 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 then the APL system changes the first bit of C to 1, and on detecting this case the emulator takes an 'ERASE' type exit. If the first bit of C is zero then function return continues.

The emulator now goes through the stack and does the following:

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 plant Vn at that address

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 SCAN microcode routine. If the function has no result then it checks that the top of the stack is null and that the next instruction is an end of statement.

23.5 Return From a Temporary Function

The return from a quad input or an execute temporary function requires special action. The APL system traps this case by use of the 'stop' bit in the end of statement marker. Let us now consider temporary functions resulting from single statements. The operator '-->' with no argument produces a syntax error which the APL system also traps.

There are two other cases, namely

(1) --> n

(2) successful execution of the statement with no branch

n stands for a scalar integer. Statements of the form '--> expression' are either in error or reduce to case (1).

Consider case (1). This case has arisen because the user had typed '--> n'; at the beginning of the execution of this statement, the stack was either empty or the head of the stack was a STOP WORD (see 23.6). After executing the statement, the emulator frees the temporary function. At this point the stack registers are irrelevant and the stack in memory has the form

K ....

The two low order bits of K may be 01, 10, or 11. If the bits are 10 then this is the end of stack marker and the emulator takes a normal end of execution exit. If the bits are 01 then the bits were originally 11, corresponding to a stopped function, but the function has been erased; in this case the emulator takes an 'EMPTY' exit. If the bits are 11 then this requires the restart of a stopped function. K contains the name of the stopped function. The emulator puts this name into the control word 'FUNCTION' and now does a normal 'go to' in the context of that function.

In case (2), that is successful execution of the statement with no branch, then the emulator takes a normal exit from the emulator and does not free the space used by the function.

23.6 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 TSADDR, together with the contents of the stack. The status can be displayed by use of the APL commands)SI and)SIV. In this section we describe how this status is 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 SCAN microprogram may use the stack for intermediate working. (2) The function call microcode 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 status are

	Stack	Comment	Status
TSADDR -->			
	STOP WORD		H<3> *
	CALL BLOCK	T2 calls H	
	SCAN BLOCK	T<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 SCAN microprogram will push into 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 section 23.1)

```
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.

If the system is going to analyze the stack then it must start at the top of the stack which is (contents of TSADR)+4. 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 of 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. The system can analyze the situation in the following way: 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 STOP WORDs, beginning of CALL BLOCK words, 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 or to find the name of the calling function or to find the names and old values of all local variables.

IBM

24. 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 24.1 gives the console listing for the example, figure 24.2 delineates several key items and figure 24.3 gives a dump of the workspace.

In figure 24.2 we see that the workspace was loaded and the GO function executed. A 'domain error' occurred and then we see 'DUMP NO 00000001'. Normally the APL system would not produce a dump, but this was run using a version of the system especially coded to provide system and microcode debugging information. The remainder of the console listing is as would occur with the standard APL system. It shows the status indicators, the shadowed variables, function definitions and current variable values.

Figure 24.2 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 270A8 to 270E4 in the sequence GPR4, ..., GPRF, GPR0, ..., GPR3 (see 'DEBUGGING AIDS'). Thus GPR3 is found to be 273A0. Since 'A' has internal name 0070 its address table entry is at 273A0+70 or 27410.

The beginning of free space was calculated as follows: FREES (at 27390) is C70. This is a displacement so we add GPR3 (273A0) to give 28010. Since FREES points to the first real block in free space, the dummy block is the preceeding word (at 2700C).

The stack in the workspace dump (figure 24.3) shows a temporary function calling GO and GO calling F which then calls G. 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 06/06/72


       GO
DOMAIN ERROR
DUMP NO 00000001
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 3ρP←⍳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 24.1.1: EXAMPLE WORKSPACE CONSOLE LISTING

```
      )VARS
A          B          R          U

      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 24.1.2: EXAMPLE WORKSPACE CONSOLE LISTING

EXTERNAL NAME	INTERNAL NAME	ENTRY ADDRESS	VALUE OR ADDRESS
A	0070	27410	36794
B	0078	27418	367A8
F	0074	27414	28034
G	0088	27428	2809C
GO	0094	27434	280E8
P	007C	2741C	NO VALUE
Q	0080	27420	NO VALUE
R	0084	27424	36754
U	0090	27430	IMEDIATE 'X'
X	008C	2742C	NO VALUE
Z	006C	2740C	NO VALUE
Z	006C	2740C	NO VALUE
A	0070	27410	36794
F	0074	27414	28034
B	0078	27418	367A8
P	007C	2741C	NO VALUE
Q	0080	27420	NO VALUE
R	0084	27424	36754
G	0088	27428	2809C
X	008C	2742C	NO VALUE
U	0090	27430	IMEDIATE 'X'
GO	0094	27434	280E8

ITEM	ADDRESS	VALUE	
GPR3	270E0	273A0	(SEE TEXT)
GPRB	270C0	27000	(SEE TEXT)
TSADR	27404	27F4C	
NEXTINST	27400	280BC	
FUNCTION	273FC	0088	
BNDATS	27408	27C08	
FREE SPACE - BEGIN	27390	2800C	(CALCULATED)
FREE SPACE - END	27394	367FC	(CALCULATED)

FIGURE 24.2: EXAMPLE WORKSPACE ITEMS

```
GPR 0 = 00000000 000140F1 00000037 00000000 000164DC 00000008 FFFD9002 06010002
GPR 8 = 20036758 10211021 0001404C 000141F4 00013090 00027000 000135B0 5001376A

FPR 0 = 0000000000000000
FPR 2 = 830282040400004A4
FPR 4 = 20021F4800000000
FPR 6 = 0000000000000000

027000 = 81818181 81818182 0001674E F00273A0 000164DC 00000008 FFFD9002 06010002
027020 = 20036758 10211021 100214BC 00027000 00016390 00036C00 00036C10 A00164D8
027040 = 00000000 00000000 00028184 04000498 00000000 00000000 00000000 00000000
027060 = FF0400FE 70016766 00000000 00000000 00036D12 00036D10 0000116E 0000E672
027080 = 00001174 00027000 0001A71A 00036C00 00036E18 4001AB7A 00000002 0002816E
0270A0 = 00028172 FFFFA001 21023625 00000008 00000002 06010002 20036758 10211021
0270C0 = 100214BC 00027000 F003677C 00026300 07040088 00710100 00000061 06040090
0270E0 = 26040061 F00273A0 00013090 00000000 0650584D 5E565900 00000000 00000000
027100 = 00000000 00000000 81870E81 870E8883 00CFE701 F161F1F5 00000100 00000C00
027120 = 0000FC00 0000F800 0000F794 00001158 0000F7F8 00000010 00000001 07000000
027140 = 000D0000 00780000 4D58564A 52578D4E 5B5B585B 9293888D 81870E81 870E8883
027160 = 92935900 00000000 00020000 00000000 00000000 00000000 00000000 00000000
027180 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0271A0 TO 027240 SUPPRESSED LINE(S) SAME AS ABOVE .....
027240 = 00000000 00000000 80000001 84000000 00000000 00000000 00000000 00000000
027260 = 00000000 60000005 00000000 00000000 00000000 00000000 00000000 00000000
027280 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0272A0 TO 0272C0 SUPPRESSED LINE(S) SAME AS ABOVE .....
0272C0 = 00000000 07000000 00000000 00000000 00000000 00000000 00000000 00000000
0272E0 = 0B100B11 50030B00 F002017E 50000B00 F002017E 0005EA06 00000004 0A803502
027300 = 000003FF FFFFF800 00000000 00000000 0001DF28 830192B4 F002017E F001ABAC
027320 = 0B020000 F001EDCE 4000046C 00000000 F00273A0 F002731C 0402626F F002780C
027340 = 8C023935 B7232340 00000008 10211021 07040088 F00280BC F0027430 F0027430
027360 = F00247CE F7022388 B7054461 DF020340 F001A4F6 70000B03 B7232340 F002736C
027380 = F0027348 700D0B00 00000000 00000000 00000C70 0000F460 3D8941BB 00000044
0273A0 = 00000002 00000002 00000000 00000000 00000000 00000000 00000000 00000000
0273C0 = 2F04008D 2F000000 2F000001 2B0192D0 2B0192E4 2B0192F8 2B01930C 2B019320
```

FIGURE 24.3.1: EXAMPLE WORKSPACE DUMP

```
0273E0 = 2F000000 2B028014 2B028024 27000000 27000000 0FFE0070 0FFE0090 2F040088
027400 = F00280BC 2B027F4C 2B027C08 27000054 2B036794 3B028034 2B0367A8 27000054
027420 = 27000042 2B036754 BB02809C 27000042 2F040061 9B0280E8 27000000 27000000
027440 = 27000000 27000000 27000000 27000000 27000000 27000000 27000000 27000000
027460 TO 027800 SUPPRESSED LINE(S) SAME AS ABOVE .....
027800 = 27000000 27000000 27000000 2F040063 2F04004A 2F04004F 2F04004B 2F040059
027820 = 2F04005A 2F04005B 2F040050 2F040061 2F04005E 2B0280D8 99028158 040004A8
027840 = 2B0281A0 0400049C 00000000 00000000 00000000 00000000 00000000 00000000
027860 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
027880 TO 027C00 SUPPRESSED LINE(S) SAME AS ABOVE .....
027C00 = 00000000 00000000 00000000 FF000000 FF000000 FF000000 FF000000 FF000000
027C20 = FF000000 FF000000 FF000000 FF000000 FF000000 FF000000 FF000000 FF000000
027C40 TO 027F40 SUPPRESSED LINE(S) SAME AS ABOVE .....
027F40 = FF000000 FF000000 FF000000 FF000000 0FFE0030 0FFE0002 2B0281F4 0F020080
027F60 = 27000000 0F020090 27000042 0F020001 27000000 0FD1008C 0FD10074 0F000042
027F80 = 60016001 2E010003 60016001 62016001 40054005 07010084 0FFE0040 0FFE0002
027FA0 = 2B0281E0 0F020084 27000000 0FFE0080 2B028188 0F02007C 27000000 0F020078
027FC0 = 27000000 0F020070 27000000 0F00006C 0F000094 0F000054 07010084 0F000028
027FE0 = 0F000002 27000018 0F000001 27000018 0F000001 27000018 0F000001 0F000498
028000 = 0F000018 07000000 08000002 00000005 0000000D 00500044 00000000 0000000D
028020 = 0000000D 00540048 00000000 0000000D 00000065 00540074 00030050 00000040
028040 = 006C0070 0078007C 00800084 0002A001 007011D9 70010080 A0010078 10210070
028060 = 70010084 A0014005 60010106 00036001 04060061 00885005 00707001 006CA001
028080 = 0003A001 001A0024 0030004A 00000000 00000058 00000065 00000039 00540088
0280A0 = 0001002A 00000030 008C0001 00900080 0002A001 01060002 10210090 7001008C
0280C0 = A0010003 A0010016 00240000 0000002C 00000039 0000000D 00540494 01505800
0280E0 = 0000000D 0000006D 00540094 0002005A 00000028 00010001 00010002 A0010106
028100 = 00091109 7001007C 1101510A 00160000 00000003 00000003 00000002 70010084
028120 = A0010106 00031109 10251591 11114001 01060003 11095001 00740084 A0010003
028140 = A0010014 00380054 00000000 00000060 0000006D 0000002D 00540498 0001001E
028160 = 00000028 00010001 00010002 A0010094 A0010003 A0010014 00180000 00000020
028180 = 0000002D 00000015 08D10C10 00000001 00000001 00000009 00000015 0000003D
0281A0 = 007104A0 00000001 00000002 00000003 00000004 00000005 00000006 00000007
0281C0 = 00000008 00000009 00000003 00000003 00000002 00000009 0000003D 00000011
0281E0 = 01F10C00 007104A0 FFFF0070 00000011 00000011 01D10BB8 007104A0 0070FFFF
```

FIGURE 24.3.2: EXAMPLE WORKSPACE DUMP

```
028200 = 00000011 0000E54A 00000000 00000000 00000000 00000000 00000000 00000000
028220 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
028240 TO 036740 SUPPRESSED LINE(S) SAME AS ABOVE .....
036740 = 00000000 00000000 00000000 0000E54A 0000003D 00710084 00000002 00000004
036760 = 00000006 00000006 00000009 0000000C 0000000A 0000000E 00000012 00000003
036780 = 00000003 00000002 00000009 0000003D 00000011 01F10070 007104A0 0C000BB8
0367A0 = 00000011 0000003D 00710078 00000001 00000002 00000003 00000002 00000004
0367C0 = 00000006 00000003 00000006 00000009 00000003 00000003 00000002 00000009
0367E0 = 0000003D 00000014 08D104A0 00000001 00000001 00000003 00000014 00000005
036800 = 0000046C 00000470 00000474 00000478 0000047C 00000480 00000484 00000488
036820 = 0000048C 00000490 00000494 80000000 00000000 00000000 00000000 00000000
036840 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
036860 TO 036C00 SUPPRESSED LINE(S) SAME AS ABOVE .....
036C00 = 00000000 00000000 00000000 00000000 00016390 00036C00 00036C10 A00164D8
036C20 = 0001674E 00036C10 00036C20 60016742 0001674E 0001674E F00273A0 000164DC
036C40 = 00000008 4001A84A 00000000 00001158 0000E6A0 0001E309 00000094 00000039
036C60 = 00036D1C 0003682C 00000000 00000000 00000000 0000FD13 00000001 00000000
036C80 = 00000001 00000000 00000000 00000000 0001A71A 700199BC A002086E 0000FD10
036CA0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
036CC0 = 00000000 00000000 00000000 00000001 00001158 0000116E 00000000 00000000
036CE0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
036D00 = 00001000 00000000 00008D8D 8D8D8D8D 50589293 4E929357 92930D5A 0D5B9293
036D20 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
036D40 TO 036E00 SUPPRESSED LINE(S) SAME AS ABOVE .....
036E00 = 00000000 00000000 00000000 00000000 00000000 00000000 0001A71A 5001B002
036E20 = 0001A71A 4001D46A 0001D79C 0000E6A0 0001E309 00000094 0000010A 00000000
036E40 = 00000002 00000000 00000010 0000010A 00036D17 00000000 00000212 00036E18
036E60 = 00FCFCFC FCFCFCFC FCFCFC01 84000212 00036C37 00000000 00000005 00000000
036E80 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
036EA0 TO 037000 SUPPRESSED LINE(S) SAME AS ABOVE .....
```

FIGURE 24.3.3: EXAMPLE WORKSPACE DUMP

25. FUNCTIONS IMPLEMENTED IN APL OR IBM/370 CODE

Most of the APL operations are done by the microcode but some of them are done by APL functions or IBM/370 code. There are several reasons for using non-microcoded functions. There is only a small amount of control store and the amount of microcode has to be strictly limited. For some operations, the 370 code is almost as fast as a microcode implementation would be. Some operations are used relatively infrequently and they do not justify the use of microcode. Some operations, such as input/output, require extensive interaction with the operating system; putting them in microcode would not improve performance and it would make the emulator system dependent. Some operations, such as domino (matrix divide and least squares fit) are obviously at a higher level than operations such as plus and minus; it is natural to put these operations in APL or 370 code.

25.1 The Calling Mechanism

The non-microcode 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 APL or 370 function. The transfer vector entries are shown in figure 25.1. Suppose the APL emulator is in control and it decides to call the dyadic Ibeam 370 function. The emulator puts the arguments of the Ibeam into the general purpose registers using the process specified in the section '370 REGISTERS AND 'GETV''. It sets GPR4 equal to C. It sets the 370 instruction location counter equal to C + hexadecimal 84 (according to the table the dyadic Ibeam entry is 84) and it exits to the 370 emulator. Location C + 84 contains a branch to the 370 function which does the dyadic Ibeam and it can use GPR4 as a base register. The 370 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.

25.2 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 or both 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 370 function using the calling mechanism described in the previous paragraph:

power, log, real residue, binomial, circular

We also call the '370 function' for real divide and some cases of real multiply but in these cases the transfer vector entries reduce to:

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

00 unused	04 unused	08 unused	0C unused
10 unused	14 floor	18 roll	1C factorial
20 unused	24 multiply	28 unused	2C binomial
30 power	34 deal	38 circle	3C unused
40 unused	44 unused	48 unused	4C unused
50 unused	54 unused	58 I roll	5C unused
60 unused	64 residue	68 unused	6C unused
70 logarithm	74 unused	78 divide	7C unused
80 m Ibeam	84 d Ibeam	88 xten stack	8C xten name
90 share-in	94 share-out	98 decode	9C encode
A0 take	A4 drop	A8 grade up	AC grade down
B0 d iota	B4 member	B8 unused	BC rotate
C0 unused	C4 unused	C8 m dominoe	CC d dominoe
D0 execute	D4 unused	D8 scan	DC reduce
E0 inner prod	E4 outer prod	E8 m format	EC d format
F0 m box	F4 d box	F8 share-post	FC unused

FIGURE 25.1: 370 (OR APL) FUNCTION TRANSFER VECTOR (m=monadic, d=dyadic; 00 through 7C are scalar functions; 88, 8C and F8 are special functions which must return with APLSRTN)

In these two cases the calling mechanism may seem somewhat elaborate but it ensures a clean interface between the APL and 370 emulators. In all of these dyadic scalar cases we are calling a 370 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, the left and right operands are real, they are in FPR4 and FPR6, and the result goes in FPR4. The 370 functions must save and restore any registers they use (other than FPR4, GPR4, GPR5 and GPRA). If the 370 functions detect an error (for example, negative input to be logarithm routine) then they should go directly to the appropriate error exit in the APL system. The 370 functions may look at the descriptor bits (see the section on 'GETV') to determine the properties of the arguments.

The reduction and inner and outer product routines go through a similar sequence and they call the same 370 functions in the same way. 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' routine is required to do several things 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

There is also a 370 function called 'I roll' which sets GPR5 equal to a random choice from iota N where N is the integer in GPR6.

25.3 Complete 370 Functions

In a case such as 'L d R' where d stands for dyadic Ibeam, then the emulator goes through 'GETV' for L and R and then calls the 370 dyadic Ibeam 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 370 function the registers are as specified by 'GETV'. The 370 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 can be used by system programmers but they must take care that it is syntactically correct. In the first three cases, the APL emulator will free the space used by the arguments, if necessary, and return to the SCAN microcode. The fourth case is discussed below. Monadic 370 functions are treated in the same way except that the 'left' argument will be missing and an immediate zero will have been substituted.

25.4 APL Functions

Any of the complete functions (but not the scalar functions of 25.2) may be written in either 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, gets the internal representation of the APL function and produces a CSECT. The easiest way of getting the CSECT is to punch the internal representation on cards in the form

DC X'hexadecimal internal representation of APL function'

and assemble it using the OS or DOS assembler. The CSECT is loaded as part of the APL system. When the APL emulator detects the dyadic domino operator, then it calls the appropriate 370 function. That 370 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. This description should not be taken to imply that the domino in APLM is in APL code; one early version of ALPM did have an APL domino, but it may have been changed to 370 code. This would not require any change to the APL emulator.

25.5 Microcode/370/APL Functions

In the case of functions like domino or shared input/output, the emulator has no interest in what the functions do or how they do it. As long as they do not destroy the integrity of the workspace or the registers, then any action or inaction is allowed. In the case of a function like encode, then the operation is regarded as part of the APL processor and the microcode, 370, APL functions should complement each other. In these cases the microprogrammer will have specified what cases the microcode will do and what additional information it will provide to the non-microcode functions.

25.6 APLRTN

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, and it interprets that byte as follows:

uuuu uu00	return from scalar dyadic operation (see 25.2)
uuuu uu01	return from scalar monadic operation (see 25.2)
uuuu uu10	return from complete function (see 25.3 and 25.4)
uuuu 0011	return from shared output or execute (see 25.7 and 25.8)
uuuu 0111	return from shared input or execute (see 25.7 and 25.8)

In the cases which we have described so far then the emulator will have set GPRF byte 2 before it exits to be 370 emulator so the 370 functions do not have to be aware of this byte, but some other cases will change this byte.

25.7 Shared Input and Output

The emulator does not initiate any input or output, but it does call the system whenever I/O is required. If 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/trace' 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 microcode SCAN routine reads the S then it calls the 370 'share-in' or 'share-out' function. At this stage the stack is:

```
1)    S      <- ...
2)    S      xx ...
3)    S<...> xx ...
4)    S<...> <- ...
```

where xx 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 memory 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. For case four, proceed as follows: 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 now 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' 370 routine. At this stage GPRF will contain the name of the subscripted variable, that is, the name which was formerly in GPR1. The system should communicate whatever information is necessary to the shared memory processor, and then it should APLSRTN.

25.8 Execute

Suppose the APL emulator encounters e̲X where e̲ stands for the execute operator. The emulator will do a 'GETV' and call the 370 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 in section 25.4. '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 the 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 eX in the context ...+eX... does produce a result whereas eX in the context eX,... does not cause printing. In this case, proceed as in '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.

26. ERROR RECOVERY

The use of the emulator may cause various errors to be detected. These errors can be divided into several types:

1) Errors in the user's program or data.
2) 'System error' return from APLSCAN.
3) Specification, data or addressing exceptions.
4) Errors other than type 1 and 3 on return from APL macros other than APLSCAN.

The first type of error will cause a 'syntax error', 'value error', etc. to be signalled on return from APLSCAN. This type of error is discussed 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 analysing 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 name of some temporary variables. The name and space of these 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... 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 arguements during the function call process. The arguements will have received permanent names by the time the function is entered and the control words will have been reset to 27... TMPNAM0 may 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.

27. 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.

27.1 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 microcode does one of two things: (a) If the APL emulator is not installed on this machine then give an operation execption. (b) If APL is installed then activate it. When APL is activated it first checks the contents of CHKWRD; if it is incorrect then APL gives a specification exception, otherwise APL proceeds with emulation of the workspace. The test of CHKWRD safeguards against APL activation as a result of a wild branch in some 370 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 core and retrying the APLEC instruction. Page faults are disusssed in greater detail below.

Termination of the APLEC instruction is always accomplished by branching to APLM.XITNRM for a normal exit with condition code zero or by branching to APLM.XITERR for an error exit with condition code one. The APLM micro-routine then retrieves the address following the APLEC from SCANRTN or SERVRTN and sets it up as the 370 instruction location. A return to I-cycles then passes control back to the 370 emulator.

27.2 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 microcode 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 micro-address with that of the micro-instruction in APLM 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 faulting micro-address are saved in SAVELS and the 370 instruction location register is set to point at the resume APLEC in INTRTN. The faulting memory address is then loaded into an appropriate register and a branch made to the instruction in APLM that reads CHKWRD. This causes a re-faulting that APL will allow 370 to process since it occurs on the 'read CHKWRD' micro-instruction.

When the paging software has the required page available and thinks it is re-executing the faulting 370 instruction it will actually be executing the resume APLEC and the APL emulator will continue with the workspace execution.

27.3 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 (I1 latch set) the APL emulator will checkpoint itself in SAVELS exactly as above, set the 370 instruction location counter to point at INTRTN and do a return to I-cycles from APLM. 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 supervisor. 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 I1 the APL emulator polls the quantum end switch. If 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 370 APL system's quantum end routine) and do a return to I-cycles from APLM. As above, the workspace can be resumed later but to do so the 370 APL system must explicitly execute an APLRESM.

27.4 370 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 (BAL 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 'FUNCTIONS IMPLEMENTED IN APL AND IBM 370 CODE'; here we merely complete our description of the interface between the 370 and APL emulators.

All breakouts to 370 functions are processed through the S370 microcode routine. There are no microcode linkages or working storage to be saved. This routine merely sets the 370 instruction location counter to point to the appropriate entry in the 370 function transfer vector using CALL370F and branches to the common exit portion of APLM. When the 370 function is complete it will issue an APLRTN. The 370 emulator will decode the APLEC and branch to APLM; APLM will then recognize the 'return' APLEC code and branch to the S370 routine which will send control back to the appropriate place.

Some of the emulator features which are coded as 370 functions, such as address table extension, require saving of microcode linkage and work registers. These checkpoint themselves in SAVELS as in the page fault case prior to going to S370. The corresponding 370 functions terminate with APLSRTN rather than APLRTN. This special return does not trickle back through S370; rather, the checkpointed information is recovered and control passed back to the invoking micro-routine via APLM and INTR.

IBM

27.5 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 is decoded into the APL emulator and after awhile a return to I-cycles is made with the 370 instruction location counter pointing at the point following the APLEC. At other times the return is accompanied by an instruction location counter pointing to some vastly different address (INTRTN, c(QEND), or some point in the 370 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 other 370 instructions is that it may be decoded at location xyz, but cause a page fault as if it had 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 370 function for domino; in fact most of the 370 functions were written by APL system programmers. But the 370 functions are logically viewed as mere extensions of the microcode. When the APLSCAN is complete, control will return to the next instruction.

28. DEBUGGING AIDS

There are several debugging aids available. These are discussed below in 28.1 and 28.2. An example is then discussed in 28.3. Figure 28.1 summarizes much of the debugging aids information. These aids have been of great value during the early development stages but are of much less importance now since bugs occur quite infrequently.

28.1 DBUG Microcode Routine

The DBUG microcode routine may be either active or inactive. When active it monitors all entries to and exits from the APL emulator (except the TIDY data exception: see 28.2) and provides useful debugging information. It represents considerable entry/exit overhead and should be inactive, or not even assembled, in an ideal situation. If the DBUG routine is included in a microcode coreload, then after a normal IMPL DBUG is inactive. Let XX denote 'the DBUG module' of the coreload (the value of XX may be found by consulting the DBUG routine listing for the coreload). To activate the DBUG 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   APLM.CHECK.0X
0000XX81   at   APLM.SETNSI.00
0100XX80   at   PAGE.REFALT
```

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

The DBUG routine uses a 'DBUG BOX' stored in the workspace. The DBUG box is at the location GPRB+(the contents of control store location XX08). The format of the DBUG box is given in figure 28.2.

When DBUG is active and any call is made to the APL emulator, the DBUG box is updated and the APLEC in byte 0-1 of word 0 of the DBUG box is executed. Similarly any exit from the APL emulator will be filtered through the APLEC in bytes 2-3 of this word. This provides two convenient address stop locations for emulator/system tracing.

FIGURE 28.1: DEBUGGING SUMMARY SHEET

WORKSPACE DISPLACEMENTS

R11	R03	CONTENTS
0A8	---	SAVE REGS 4 TO 3
2E0	---	DEBUG BOX
2F8	-A8	TIDYS
2FC	-A4	CPUTFUZX
300	-A0	SKIP1
304	-9C	CPUTFUZM
308	-98	SEED
30C	-94	UNUSED
310	-90	CALL370F
314	-8C	QEND
318	-88	SCANRTN
31C	-84	SERVRTN
320	-80	INTRTN
324	-7C	SAVELS
348	-58	SAVELSB
36C	-34	SAVTDY
390	-10	FREES
394	-0C	FREET
398	-08	CHKWRD
39C	-04	FRSTRELO
3A0	00	TMPSAV0
3A4	+04	TMPSAV1
3A8	+08	UNUSED
3BC	+1C	XARG0
3C0	+20	BLANK
3C4	+24	ZEROVAR
3C8	+28	ONE
3CC	+2C	REAL1
3D0	+30	PI
3D4	+34	E
3D8	+38	MIN
3DC	+3C	MAX
3E0	+40	UNUSED
3E4	+44	NULNUMVC
3E8	+48	NULCHRVC
3EC	+4C	INDEX
3F0	+50	FILL
3F4	+54	TMPNAM
3FC	+5C	FUNCTION
400	+60	NEXTINST
404	+64	TSADR
408	+68	BNDATS
40C	+6C	BEGIN USER NAMES

SAVELS FORMAT

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

DEBUG BOX

```
0B    10    0B    11
*C    *E    0B    *A
10    11    12    13
prior *C*E0B*A
prior 10111213
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
10     = key
11-3   = location
PFADR  = fault address in bytes 1-2
         if *E is 05
ECNUM  = total emulator call number
         if *E is not 05
```

APLEC CODES

00	SCAN	03	TIDY	63	FIND	C3	GETN
01	RTN	23	NAME	83	FREE	D3	GETV.
02	RESM	43	UNAM	A3	FRIF	E3	DIAG

SPECIAL MICRO ADDRESSES

XX00	total emulator call number
XX04	emulator call count down number
XX08	DEBUG BOX R11 dispalcement
XX0C	return to I-cycles
XX10	one instruction stop-loop
XX80-XX8C	DBUG 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, say 'BOO' and repeat this step.

4. ADDRESS COMPARE = CTR WORD ADR
   Push CONTROL ADDRESS SET.
   Push START several times.

5. Repeat step 1 and push START.

```
WORD 0        +---------------+
WORD 1        |0B  10  0B  11|
WORD 2        |*C  *E  0B  *A|
WORD 3        |I0  I1  I2  I3|
WORD 4        |prior *C*E0B*A|
WORD 5        |prior I0I1I2I3|
              |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 28.2: DBUG BOX FORMAT

In figure 28.2 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 DBUG box are ignored) XX00 is incremented and XX04 is decremented. The microcode instruction DBUG.STOP.0 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 microcode 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 micro-step, etc.

The DBUG package also 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 micro-loop). The second is useful in conjunction with alter/display and the microcode address trap console feature.

28.2 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 microcode. In this section we have described the DBUG box which is in the system's part of the workspace. We further note that the system usually precedes each APL macro with

```
LM   4,3,X'A8'(11)
```

and follows the macro with a corresponding STM. These things are true for the present system but they are not part of the emulator or system specifications.

During garbage collection the TIDY microcode routine checks each active block in free space for a correct back pointer (address table or stack entry). On errors it gives a data exception with the PSW pointing at the word after the DN word. On these dumps the collection will be partly done; it will have proceded to the error block so the free space format may look strange. This code could be removed if the emulator and system could be judged perfect, but it costs relatively little in terms of overhead.

On exits for page faults, to take interrupts, or any other exit that passes through the INTR microcode routine, 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. This is a permanent emulator feature.

On APL error exits, i.e., exits which pass through the ABEN microcode, local storage is saved in SAVELSB in the same format as above; however, the ABEN routine will have altered a few of the registers (see the microcode listing for details). This is temporary code which should be removed eventually.

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

Finally there is the APLDIAG instruction. This is used mainly in testing areas where there may be a machine malfunction or a misunderstanding of the microcode instruction specifications. Consult the source listings for comments at the APLDIAG decode point in the SERV routine.

28.3 An Example

Figure 28.3 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 (DBUG BOX: THIS EXIT). The faulting address was 5400XX (DBUG 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

0B100B11 = PSEUDO CALLS		DBUG BOX
40050B00 = 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 28.3: EXAMPLE DUMP INFORMATION

FUNN.131. That instruction 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 370 function (DBUG BOX: LAST ENTRY) having an APLRTN at address 016340-2 (DBUG BOX: LAST LOC). There is a history of ASGN calling GETV (LINKAGE REGISTERS: GPRA) and of SERV calling FREE (LINKAGE REGISTERS: GPRD).

29. CONCLUSIONS

The architecture of the APL processor and the design of the 370 emulator/APL emulator interface was completed in October 1970. The implementation of both microcode and software 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. Almost all of the microcode debugging was done using simulation. An initial and partial version of the emulator and the APLM system was operational in November 1971. A complete emulator and APLM system was running successfully and reliably in April 1972. It is reasonable to expect that any system will contain some errors; one of the problems with errors in microcode is that they may be confused with hardware errors and they may cause the operating system to fail disasterously. The APL emulator has been used for several months on a model 145 which supports general scientific computing and systems programming under a CP operating system (the system is similar to CP/67 but operates on a model 145). During this time the emulator has never caused the system to fail and has never caused an error which looked like a hardware problem.

The execution of APL programs is done partly by the microcode and partly by the software. At the current time the status is as follows ...

(a) Completely in microcode:

statement scan and syntax analysis
management of free space and garbage collection
function call and return
plus, minus, negative, signum
magnitude, maximum, minimum
all logical operations
all comparisons
size, reshape, catenate, laminate, ravel
index generator, compress, expand
reverse, transpose
goto, assignment, subscripting
integer cases of times, residue, floor, ceiling

(b) Analysis and operand fetch/store in microcode; operation on one element or one pair of elements is done in IBM 370 code:

power, exponential, logarithm, binomial
circular, real residue, factorial
real multiply, pi, divide, reciprocal
real floor, real ceiling

(c) Done by the APLM system; may be in APL or IBM 370 code:

grade up, grade down, roll, deal, domino
scan, format, translation part of execute, Ibeam
input/output

(d) The remaining items are:

index of	currently in APL; will be microcoded
membership	same
encode, decode	currently in APL; will be partially microcoded
take, drop	in microcode if the left argument has one element; otherwise in APL but will be in IBM 370 code
rotate	in APL if the right argument is an array; otherwise in microcode
inner product	in APL if either argument is an array; otherwise in microcode
outer product	same
reduction	scalar and non-AP vector in microcode; rest in APL

The microcode for the APL emulator occupies 20K bytes of control memory and can co-reside with an IBM 370 emulator which will fit in the remaining 44K bytes. We could conveniently have put the items in class (d) and the items in rows three and four of class (b) in microcode but will 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 class (c); the grade operation might seem like a good candidate for microcode but a superficial examination indicates that on the model 145 the microcode would not be appreciably faster.

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 D. L. McNabb and J. R. Walters ('MPL/145 A Language and Compiler for System/370 Model 145 Microprogramming', Palo Alto Scientific Center report number ZZ20-6410). 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. M. J. Beniston was responsible for the design and implimentation of all the APLM software for use under CMS; this work, to be reported elsewhere, includes the software for the translator, the editor, shared variables, error recovery, format, the 370 functions, etc. J. W. Lageschulte used the APL/360 and CMS/APLM systems to develop the stand alone and OS versions of APLM. H. R. Penafiel wrote an early version of the translator and part of an interpreter in IBM 370 code which was used in checking out the software. R. J. Creasy provided advice and encouragement throughout the project, wrote a number of the APL system functions and solved one of our major problems by pointing out that the problem could not occur.

30. ADDITIONAL REFERENCES

A. Hassitt, J. W. Lageschulte and L. E. Lyon, "Implementation of a High Level Language Machine", accepted for publication in the CACM (this is an unclassified version of IBM Palo Alto Scientific Center report number ZZ20-6402)

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.

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

This provides an APL description of the methods used by the microcode subscripting routines.

31. MICRO-ROUTINE NAMES

Other portions of this document refer to specific microcode routines by name (i.e. SCAN or GETV). These are the names used for the routines during most of the development stages. When the routines were actually merged with the 370 routines it became necessary to rename the routines. The corespondence follows:

DEVELOPMENT NAME	FINAL NAME
ABEN	PLAA
APLM	PLAB
APVX	PLAC
ASGN	PLAD
BASE	PLBW
CHIX	PLAE
CMEX	PLAF
COMA	PLAG
COPY	PLAH
DBUG	PLDB
DIVI	PLBX
DMIX	PLAI
DYAD	PLAJ
DYOP	PLAK
DYOV	PLAL
DYSC	PLAM
DYVC	PLAN
EPSI	PLBY
FINS	PLAO
FRNS	PLAP
FUNN	PLAQ
GETN	PLAR
GETV	PLAS
GOGO	PLAT
INDA	PLAU
INDB	PLAV
INDC	PLAW
INDD	PLAX
INDE	PLAY
INTR	PLAZ
MMIX	PLBA
MNAD	PLBB
MULT	PLBC
PAGE	PLBD

PROD	PLBE
REAL	PLBF
REDU	PLBG
ROTA	PLBH
RSHP	PLBI
SCAN	PLBJ
SCN2	PLBK
SERV	PLBL
SKIP	PLBM
STOR	PLBN
SYNN	PLBO
S370	PLBP
TIDY	PLBQ
TKDP	PLBR
TRAN	PLBS
TRED	PLBT
UNFN	PLBU
XTEN	PLBV