APL\B5500:

THE LANGUAGE AND ITS IMPLEMENTATION

by

Gary A. Kildall

70-09-04

Technical Report No. 70-09-04, Computer Science Group

University of Washington
Seattle, Washington 98105

September 1970

ABSTRACT

APL\B5500 is a multiple-user interpretive system for a conversational programming language implemented on the Burroughs B5500 computer at the University of Washington. The language is patterned after APL\360 which is an implementation of "Iverson Notation." This paper describes the differences between the APL\360 and APL\B5500 languages. In addition, the algorithms and data structures used in the implementation of APL\B5500 are given.

ACKNOWLEDGEMENTS

The APL\B5500 programming system was developed by members of the Computer Science Group at the University of Washington. The system was implemented by Leroy Smith, Sally Swedine, Mary Zosel, and the author, under the guidance of Dr. Hellmut Golde. The author is grateful to Dr. Golde and the co-implementors of APL\B5500 for numerous helpful suggestions regarding the preparation of this manuscript.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

APL\B5500 is a multiple-user interpretive system for a conversational programming language implemented on the Burroughs B5500 computer at the University of Washington. The language is patterned after APL\360[1] which is an implementation of "Iverson Notation"[2]. The APL\B5500 system provides line-by-line evaluation of APL statements as input by a programmer at a remote teletype station. The system provides both an "immediate execution mode" and a "stored program facility." The basic data elements of APL are numeric and character constants. Identifiers, however, can be used to name numeric and character data for later reference. In addition, the data elements are in the form of scalars, vectors, and arrays. A large number of special-purpose operators operate on the data elements allowing concise expression of mathematical or manipulative constructs.

A comprehensive set of commands allows communication with the APL system monitor providing a number of facilities useful in a conversational programming environment.

The conciseness of APL statement expression along with APL monitor functions makes APL\B5500 an excellent interactive programming system.

A full discussion of the capabilities of APL\B5500 are given elsewhere[3]; the purpose here is to provide a reasonably complete discussion of the internal structure of the system. It is useful, however, to provide an introduction to the language as well as to point out major differences between the APL\360 and APL\B5500 languages.

The structure of APL statements is most easily shown with a simple example. Consider the following ALGOL 60 program

```
begin integer n; real t;
read (n);
     begin real array a[1:n] ; integer i;
     for i:=1 step 1 until n do
          begin read (a[i] ); t:=t+a[i]
          end;
     t:=t÷n
     end;
write (t)
end;
```

which calculates the average value from a set of values stored in a dynamically allocated array A.[1] With input data

6, 5, 5, 4, 4, 8, 16,

the ALGOL program produces the output

7.

An APL statement pair which performs the same computation is as follows:

```
X := 5  5  4  4  8  16
(+/X) % RHO X
```

Dynamic storage allocation occurs on the first line where a vector constant is assigned to the variable X. The second line performs the required computation and causes the numeric scalar result to be printed.

The second APL statement contains three operators which require explanation: the reduction operator (/), the divide operator (%), and the RHO operator. The reduction operator applies the operator occurring on its left to the vector to its right by "placing" the operator between each element of the vector. Thus, since

X = 5 5 4 4 8 16

then

(+/X)

is equivalent to

5 + 5 + 4 + 4 + 8 + 16.

In this case, the divide operator divides the scalar to its left by the scalar value occurring on its right. The divide operator, as used here, is said to be "dyadic" since it occurs between two operands (i.e., it operates on two operands, resulting in a single operand). The RHO operator is "monadic" in this example since it operates on only one operand (the one which occurs on its right). The RHO operator is used here to extract the "dimensionality" of the vector X. RHO X results in the scalar value 6 since

---

[1] This ALGOL example includes the use of "read" and "write" procedure calls which produce the obvious effect. Clearly, the computation could be performed without the array A; it is included here in order that the dynamic storage allocation can be compared.

X represents a vector with six elements. Thus, the second APL expression

```
(+/X) % RHO X
```

reduces to a final scalar result through the following steps:

```
(+/X) % RHO X
(+/X) % 6
(+/(5 5 4 4 8 16)) % 6
(42) % 6
7.
```

It should also be noted that there is no hierarchy of operator evaluation in an APL statement. All operators are applied from right to left in the statement; the order of evaluation can be controlled, however, by properly parenthesizing sub-expressions. Hence, the APL statement

```
+/X % RHO X
```

reduces to (approximately) the same result as above through the steps:

```
+/X % RHO X
+/X % 6
+/(5 5 4 4 8 16) % 6
+/(5%6 5%6 4%6 4%6 8%6 16%6)
((5%6)+((5%6)+((4%6)+((4%6)+((8%6)+(16%6))))))
7.
```

Note that in this case the divide operator divides the vector on its left by the scalar on its right, resulting in a vector. The results may actually differ, of course, due to truncation errors.

There are approximately forty-five special-purpose operators in APL\B5500. Most of these operators can be taken as monadic or dyadic depending upon the context in which they are used.

As mentioned previously, APL statements can be grouped together in stored programs or "functions." The APL programmer defines a function by entering "function definition mode." This is accomplished by typing a $ ("del") at the teletype, followed by an identifier which names the function.[1]

---

[1] Although all APL programs are called "functions," it is not necessary that the program return a value in the usual functional sense.

An APL function defined for the purpose of calculating the average of a set of numbers follows:

```
          $AVERAGE
[1]       X:=[]
[2]       (+/X) % RHO X
          $
```

The line numbers enclosed in brackets on the left are typed out automatically by APL\B5500 when the user is operating in function definition mode. In addition to automatic line numbering, an extensive set of commands is provided for displaying defined functions, deleting function lines, and altering previously typed lines.

The above function is invoked by typing

```
AVERAGE
```

at the teletype. The function begins execution at line one and immediately encounters the [] ("quad"). Executing the quad causes the teletype to be opened for input with the prefix

```
[]:
```

indicating that APL requires input data from the terminal station in order to continue execution. The APL programmer may then type input data such as:

```
5  5  4  4  8  16
```

and the result

```
7
```

is printed at the teletype.

As mentioned above, APL\B5500 provides the APL user with a set of monitor communication commands. These monitor commands allow the user to sign onto the APL system, interrogate APL regarding the contents of a work area, maintain separate work areas, and provide APL execution parameters for his programs.

All APL monitor commands are prefixed with a ")" by the user in order to distinguish them from other APL statements. Most of the APL\B5500 commands correspond exactly to APL\360 commands[1]. A more complete discussion of APL monitor commands is found in a later section; the sample terminal session given in Figure 1, however, includes a number of monitor command examples.

```
)SIGNON,
 D. NIXON LOGGED IN WEDNESDAY 10-21-70  10:42
      X:=5 5 4 4 8 16←
      X←
5  5  4  4  8  16

      $AVERAGE[[]]$←

      AVERAGE
[1]   X:=[]
[2]   (+/X)%RHO X


      AVERAGE←
[]:
      5 5 4 4 8 16←
7

      )VARS←
 AVERAGE(F) X        Y
      )FNS←
 AVERAGE
      )DIGITS←
9
      )DIGITS 3←
      1 % 3←
0.333
      )DIGITS 5←
      1 % 3←
0.33333
      )WIDTH←
72
      )LOGGED←
(1) IS  D. NIXON
(2) IS  P. NIXON
      )ORIGIN←
1
      )FUZZ←
1E-11
      )SEED←
59823125
      )CLEAR←
      )VARS←
 NULL.
      )OFF←
      END OF RUN
```

FIGURE 1

SAMPLE APL\B5500 TERMINAL SESSION

HARDWARE CONFIGURATION REQUIRED FOR APL\B5500

APL\B5500 is implemented on a Burroughs B5500 computer system. The machine used in the implementation is a single processor system with 32,768 words of 48-bit central memory. Messages to and from remote teletypes are buffered in a single Burroughs B487 Data Transmission Terminal Unit (DTTU). The B487 DTTU is interfaced with model 33 or model 35 teletypes through line adaptors and Western Electric 103A2 (dial-up) data sets. The equipment required for remote operation of APL\B5500 is a model 33 or model 35 teletype with attached acoustic coupler or data set. The remote teletypes must operate in half-duplex mode. In addition, teletypes may be directly connected to the B487 DTTU through line adaptors.

The virtual memory of the APL system requires access to at least one B475 Disk File Storage Module (9.6 million character capacity).

No line printers, tape drives, or card readers are required for normal APL operation.

A complete description of the B5500 hardware components is given in the B5500 hardware reference manual[4].

SOFTWARE ENVIRONMENT REQUIRED BY APL\B5500

APL\B5500 is designed to operate concurrently with other batch and conversational programs under control of the B5500 multiprogramming Master Control Program (MCP). APL is coded entirely in B5500 Extended Algol, except for a few statements which allow APL to directly communicate with the MCP. For the most part, APL runs under the same conditions as any B5500 user program and thus enjoys the protection and facilities (e.g., dynamic storage allocation, automatic overlay, and disk-file input/output facilities) provided by the MCP.

A primary design objective in the organization of APL was that the resulting system operation interfere as little as possible with normal B5500 user program processing. In light of this objective, APL central memory requirements are approximately 3000 words of resident (non-overlayable) storage with an additional 7000 words of transient (overlayable) storage. Resident and transient requirements can be altered at APL system compilation time with

a corresponding trade-off in system response time.

The current version of APL\B5500 also requires the services of a separate priviledged program called the "remote handler." The remote handler interrogates the B487 DTTU and passes messages between the B487 and APL. APL has been coded in such a way as to allow the remote handler to be removed and its functions taken over by APL with a small amount of re-coding.

In many ways, APL\B5500 can be considered a time-sharing submonitor and language processor under the B5500 MCP since it:

(1) handles its own virtual memory,

(2) handles its own terminal input/output processing,

(3) supervises execution of APL statements and functions,

(4) schedules APL user tasks and APL monitor tasks for execution,

(5) maintains back-up storage for APL work areas, and

(6) provides an APL-oriented command language for user control of APL monitor functions.

After initial connection of a user terminal to the B5500, the terminal is under control of the APL\B5500 system.

A complete description of the B5500 software is given in the Narrative Description of the B5500 Master Control Program[5].

APL\B5500 LANGUAGE DESCRIPTION

The APL\B5500 statement and monitor command syntax is, for the most part, structurally equivalent to APL\360, with a transliteration of the APL\360 character set. The correspondence between the two languages is maintained as much as possible in order that an APL programmer can easily make the transition from one language to the other. In addition to the usual APL\360 operators, the proposed monadic epsilon operator ("execute string") is implemented in APL\B5500, as shown in Table 1. The monadic epsilon operator operates on a vector character string containing an APL statement. The result of the operation is the result of the evaluation of the APL statement contained in the character string. Thus,

```
EPS "2+3"
```

results in the scalar 5. If the APL statement is invalid, an appropriate error

message is printed.

APL\B5500 monitor commands are summarized in Table 2. The command structure is similar to that of APL\360 except for the "SYN," "NOSYN," "STORE," "ABORT," and "Line Edit" commands.

The Line Edit command is particularly useful when only a slight error has been made in a line typed by the user. The form of the Line Edit is:

)"<search string>"<insert string>"<search string>"

or

)"<search string>"<insert string>"

In either case, the last message typed by the APL user is edited according to the Line Edit command and resubmitted for processing by APL. The action of the Line Edit is as follows: the first <search string> is located by APL in the last line typed by the user; when it is found the <insert string> is placed into the line, and all characters up to the occurrence of the second <search string> are deleted. If the first <search string> is not found then no changes are made. If the second <search string> is not found then all characters after the <insert string> are deleted. Finally, if the second <search string> is not specified then no characters are deleted. The null string is found immediately in all cases. Thus, if the user first types:

((+/(X-AVE)*2/⍴ N-1)*.5

he will receive an error message (unbalanced parenthesis: "*2/" should have been typed as "*2)"). The line can be altered by typing:

)"2")"⍴"

and APL will respond with:

((+/(X-AVE)*2)⍴ N-1)*.5.

The statement is then resubmitted for execution. This command is extremely useful when a long expression has been typed which needs simple alteration. A similar command is available in function definition mode allowing alteration of all or part of a function definition.

APL\B5500 also differs from APL\360 in the method of handling global variables when executing functions. Functions which contain errors (syntactic or semantic) are "suspended" at the point where the error occurs. Suspended functions may have operated upon global variables to produce new values for

TABLE 1

APL\360 TO APL\B5500 TRANSLITERATION

APL\360	APL\B5500	MONADIC FORM	DYADIC FORM
+	+	identity	addition
-	-	additive inverse	subtraction
×	&	sign	multiplication
÷	%	mult inverse	division
*	**	exponential	exponentiation
⍟	LOG	natural logarithm	logarithm
⌈	CEIL or MAX	ceiling	maximum
⌊	FLR or MIN	floor	minimum
|	ABS or RESD	absolute value	residue
!	FACT or COMB	factorial	combinatorial
?	RNDM	random number	random deal
~	NOT	negation	
○	CIRCLE	circular	circular
<	LSS		less than
≤	LEQ		less or equal
=	=		equals
≠	NEQ		not equal
≥	GEQ		greater or equal
>	GTR		greater than
∧	AND		and
∨	OR		or
⍲	NAND		nand
⍱	NOR		nor
⍳	IOTA	index generator	indexing
⍴	RHO	dimension vector	restructuring
,	,	ravel	catenation
⍉	TRANS	transpose	
⊥	BASVAL	base-2 value	base value
⊤	REP		representation
∈	EPS	**execute**	membership
↑	TAKE		take
↓	DROP		drop
;	;		heterogeneous output
/	/		compression
\	\	scan	expansion
⌽	PHI	reversal	rotation
⍋	SORTUP	sorting up	
⍒	SORTDN	sorting down	

TABLE 1 (CON'T)

APL\360	APL\B5500	USAGE
return key	←	end of message signal
□	[]	input or display
⍞	["]	character input
→	=: or GO	transfer control
←	:=	assignment
[...;...;...]	[...;...;...;...]	subscripts
¯	#	minus sign
E	@ or E	power of ten
Δ	$	function definition
'string'	"string"	string quotes

TABLE 2

APL\B5500 MONITOR COMMANDS

MONITOR COMMAND	MONITOR FUNCTION
)SAVE <name>	All variables and functions in the active work area are stored in a disk file library. The library is labeled with the user's B5500 <job number> and the identifier specified by <name>
)SAVE <name> LOCK	This command performs the same function as above except that all other APL users are prevented from accessing the library.
)LOAD <name>	The library labeled <job number> and <name> is activated for the user. All library variables and functions are accessible after the LOAD operation.
)LOAD <job number>, <name>	This command allows access to saved libraries of other APL users when the library was originally saved without the lock option. The <job number> corresponds to the user who originally saved the library.
)COPY <name>,<function>	This command adds the function named by <function> to the active work area for the user from the library labeled by <name>.
)COPY <job number>, <name> <function>	This command has the same function as the copy command above except that another APL user's library may be referenced.
)CLEAR <name>	This command removes the referenced library from the disk.
)CLEAR	This command causes all variables and functions in the active work area to be erased.
)ERASE <name>	This command selectively erases variables or functions named by <name> from the active work area.
)FNS	This command provides the user with a list of all defined functions in the active work area.

TABLE 2 (CON'T)

MONITOR COMMAND	MONITOR FUNCTION
)VARS	This command lists all variable and function names in the active work area. Functions are identified by a following "(F)."
)SI	This command lists the names of all suspended functions in the active work area.
)SIV	This command lists the names of local variables in suspended functions as well as the function name.
)ABORT	This command terminates all suspended functions.
)STORE	This command stores variables into the active work area which are global to suspended functions and which have been altered during function execution. If the ABORT command is issued before the STORE in suspended mode, global variables are left in their original state for re-execution at a later time.
)"<search string>" <insert string>" <search string>	This is the Line Edit command. The command is used to alter the most recently typed line. This command is described fully in the text.
)ORIGIN <integer>	The origin (first subscript) of arrays is assumed to be that specified by the integer value <integer>.
)WIDTH <integer>	This command changes the width of the output line to <integer> characters.
)DIGITS <integer>	This command changes the number of digits printed after the decimal point in output to <integer> digits.
)SEED <integer>	This command changes the base of the random number generator.
)SYN	This command causes APL to check each line typed by the user in function definition mode for syntactic correctness.
)NOSYN	This command reverses the action of the SYN command above.

TABLE 2 (CON'T)

MONITOR COMMAND	MONITOR FUNCTION
)LOGGED	This command lists the terminal number and user identification of each active APL user.
)MSG <integer> <message>	The MSG command allows active APL users to communicate. The <integer> is the terminal number of the station which is to receive the character string <message>.

NOTE: If the <integer> in any of the commands ORIGIN, WIDTH, DIGITS, or SEED is omitted then the current value assumed by APL is printed.

the global variables. The altered values are not permanently entered into the active work area until the function has successfully completed or until the user has issued the STORE command while the function is suspended. This feature allows re-execution of the corrected function without re-initialization of the global data.

The APL\B5500 function editor differs somewhat from the APL\360 editor. The APL editor is invoked whenever the APL user types a $ ("del") followed by a function "header" while in calculator mode. The simplest form of a function header is an APL identifier. Hence, if the user types:

```
$F
```

APL will enter function definition mode and (assuming F is a new function) will respond with

```
[1]
```

and await the first line of the function F by opening the teletype for input. As subsequent lines of text are entered, the line counter is incremented by one. Thus the user could enter the three lines:

```
[1]    A
[2]    B
[3]    C
```

with the line numbers to the left supplied automatically by APL. Although APL will "prompt" the user for the fourth line, it is possible to insert lines elsewhere in the function. The user could, for example, insert a line between lines one and two by replying to the prompt with:

```
[4]    [1.1]D
```

overriding the line prompt. APL will then take the increment last used by the APL programmer and prompt with:

```
[1.2].
```

The smallest increment possible is .0001 between lines. The largest line number possible is 9999.9999.

In general, any line prefixed by a "[" while in function definition mode is taken to be an editor command. Table 3 provides a complete listing of APL\B5500 editor commands.

The <line reference> is a basic constituent in almost all editor commands. In the simplest case, the <line reference> is an integer value corresponding

TABLE 3

APL\B5500 FUNCTION EDITOR COMMANDS

APL EDITOR COMMAND	COMMAND FUNCTION
[[]]	This command causes the currently active function to be displayed at the terminal.
[<line reference>[]]	This command causes the line specified by the <line reference> to be displayed at the terminal.
[<line reference>[] <line reference>]	This command causes all lines from the first through the second <line reference> to be displayed at the terminal.
[<line reference>]<statement>	This command inserts the APL statement specified by <statement> in the current function at the line denoted by <line reference>. The current line and the increment are changed in most cases.
[<line reference>]	This command deletes the function line corresponding to <line reference>.
[<line reference>][<line reference>]	This command deletes all lines from the first through the second <line reference>.
[IOTA]	This command causes the current function to be completely renumbered starting at one with an increment of one.
[["]]<line edit>	This command causes the APL editor to alter all lines of the current function according to the rule given in the <line edit>. The <line edit> is the same as the edit described under APL monitor commands.
[<line reference>["]] <line edit>]	This command is similar to the above edit command except that only the line referred to by <line reference> is altered.
[<line reference>["] <line reference>]<line edit>	This command applies the <line edit> from the line specified by the first <line reference> through the line specified by the second <line reference>.

to a line of a function. Thus (referring to the delete command of Table 3), the user could delete the first three lines of the above function by typing (after the APL prompt):

```
      [1.2] [1][2].
```

APL deletes the lines and returns the prompt:

```
      [1.2]
```

opening the terminal for input. Note that the function F now contains:

```
      [3]    C.
```

Another type of <line reference> is an APL statement label. Statements are labeled by placing identifiers separated by colons before the APL statement. Thus, the APL user may continue definition of F by typing (with prompting by APL):

```
      [1.2]  [4]D
      [5]    E: F+G
      [6]    L1:L2:H+I
      [7]
```

where "E," "L1," and "L2" are all statement labels. Although statement labels are used primarily for transfer of control at function execution time, they can be used as <line reference>s when in function definition mode. The line with <line reference> 5 can be deleted by typing either of the following commands:

```
      [5]
      [E].
```

A <line reference> may also involve a numeric offset on either side of the statement label. Line 5 can be displayed by typing:

```
      [L1-1[]]
```

and APL will respond:

```
      [5]    E: F+G.
```

Further, an entire set of lines around statement five may be displayed by typing:

```
      [E-1[]E+1],
```

resulting in the response from APL:

```
      [4]    D
      [5]    E: F+G
      [6]    L1:L2:H+I.
```

APL allows statement labels to be edited as well. The statement at line six can be edited by typing:

```
[L2["]]:L"3".
```

APL searches the line labeled L2 for an occurrence of ":L," inserts a "3" immediately after the occurrence, and deletes characters up to the following ":." Hence, the command:

```
[L2[]]
```

results in an error message (the label L2 no longer exists), but the command:

```
[L3[]]
```

results in the display:

```
[6]    L1:L3:H+I.
```

A last point which should be made is that labels within functions are treated as local variables, but are initialized to their respective line numbers. The line number value of a label may be altered during function execution. Further examples of function definition are given in the discussion of the APL\B5500 function editor implementation.

Appendix A shows a sample APL\B5500 terminal session including examples of APL operators, APL monitor commands, and APL function editor usage.

A formal definition of the syntax of APL\B5500 is included in Appendix B.

APL\B5500 IMPLEMENTATION

The internal data structures and program organization of APL\B5500 are given in the following sections. The time-sharing facilities of APL are explained along with a description of monitor command execution and APL "Machine" organization. APL\B5500 functions are logically divided into component parts as shown in Figure 2:

1. Resource Management. Central memory and central processor resources are allocated by the Resource Management component of APL\B5500.
2. Terminal Input/Output Handler. Terminal message bufferring and dispatching, along with primitive input/output facilities, are provided by the Terminal Input/Output Handler.
3. Virtual Memory Management. The Virtual Memory Management section provides an APL controlled extension of the B5500 central memory resources.

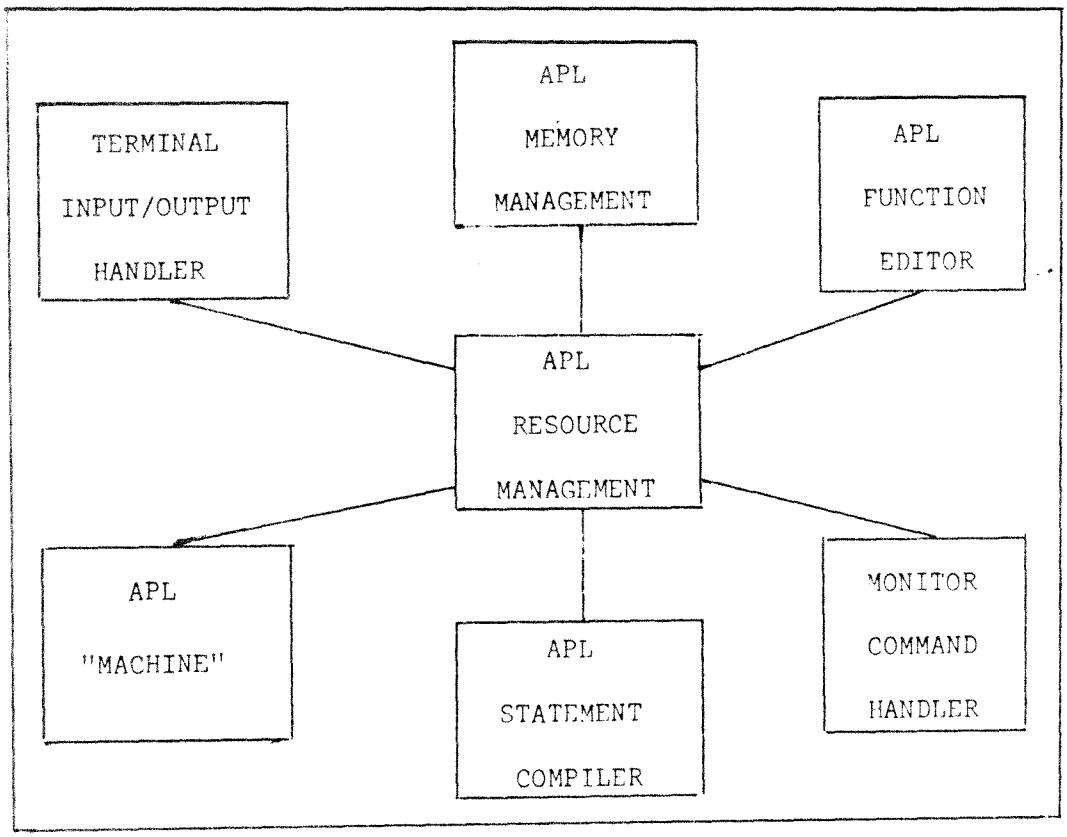


FIGURE 2

APL\B5500 SOFTWARE COMPONENTS

4. APL Function Editor. Terminal messages issued while the user is in function definition mode are processed by the APL Function Editor.

5. Monitor Command Handler. All terminal input messages which are prefixed with a ")" (i.e., APL monitor commands) are processed by the Monitor Command Handler.

6. APL Statement Compiler. The APL Statement Compiler checks the syntax of APL statements submitted for execution by the user. In most cases, "pseudo-code" is generated corresponding to the APL statement.

7. APL "Machine." The APL "Machine" is a software simulation of a computing machine oriented toward execution of APL statements.

APL RESOURCE MANAGEMENT

APL Resource Management is responsible for allocation of work to the other components of the APL system. In addition, the needs of the various terminal users are monitored constantly.

The current "state" of each active APL user is maintained in a table called the User State Table, shown in Figure 3. Each element of the User State Table, called a User State Register, corresponds to exactly one APL user. The field width of each element in the User State Register varies according to the maximum data size.

APL schedules tasks for execution based on a simple two-queue algorithm[6], with tasks which have not required a full time-slice in a (first-in first-out) queue for immediate processing. A production queue lists all tasks which require central processor resources and which have used at least one time-slice. Users without a task in the immediate queue or production queue are considered to be in an "idle" queue.

The "current mode" field of the User State Register indicates the present status of the corresponding user's APL run. The current mode of a particular user can be:

1. Calculator Mode. The user is in an idle state and is not using the APL Function Editor. Further, the user is not executing APL statements. APL is awaiting input from the user's terminal.

2. Execution Mode. The user is in the process of executing an APL statement. The APL statement may or may not have invoked functions.

FIGURE 3A

USER STATE REGISTER

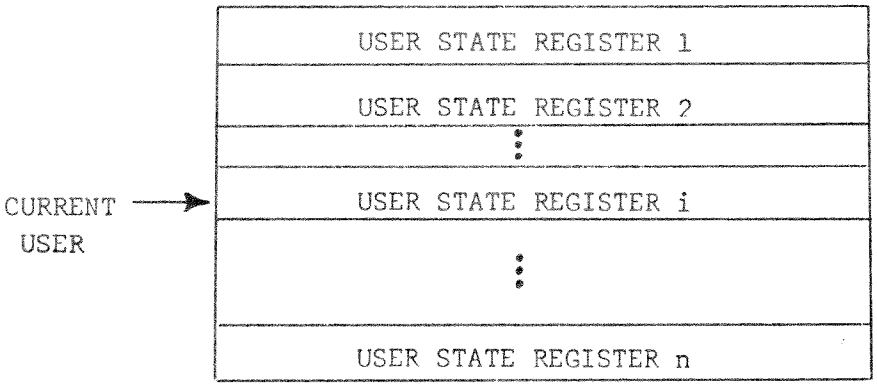


FIGURE 3B

USER STATE TABLE

3. Function Definition Mode. The user is currently defining an APL function. All messages, except those prefixed by "),," are directed to the APL Function Editor.

4. Input Mode. A user in execution mode is changed to input mode when his APL program requestsinput from the terminal (by encountering a "quad" or "quote quad"). The user is restored to execution mode when input is completed.

5. Error Mode. A user is put into error mode when his program encounters an error during execution. Messages are sent to the terminal and corrective action is taken before changing the user's current mode.

Concurrency of APL tasks is thus maintained by retaining the status of each user in order that his task may be started and stopped in various states of execution. The current user is set by the Resource Manager before execution of an APL task is initiated. The current user is indicated by an index to the corresponding User State Register, as shown in Figure 3b. When control is given to another APL component, such as the Function Editor, a small increment of processing is done for the current user with control returning to the Resource Manager almost immediately. Since the parameters required by each component are maintained in the User State Registers, it is possible for one user to define one line of an APL function and immediately after another user can use the function editor to define a line of his APL function.

This notion of concurrency is, of course, fundamental in the operation of any operating system, including time-sharing. Each component of the system must be coded in such a way as to return control to the Resource Management component as soon as possible in order that other system users do not notice any delay. This notion is referred to here as "functional concurrency."

The "user mask" field of the User State Register shown in Figure 3a contains a set of binary "switches" associated with the user's APL run. The bit positions of the user mask include:

1. The Master Mode Bit. The master mode bit is set if the User State Register belongs to the APL system supervisor. The system supervisor's user code is compiled into the APL system, and thus is the only initial valid user. A user operating with the master mode bit set (i.e., the system supervisor) is provided with additional monitor commands which

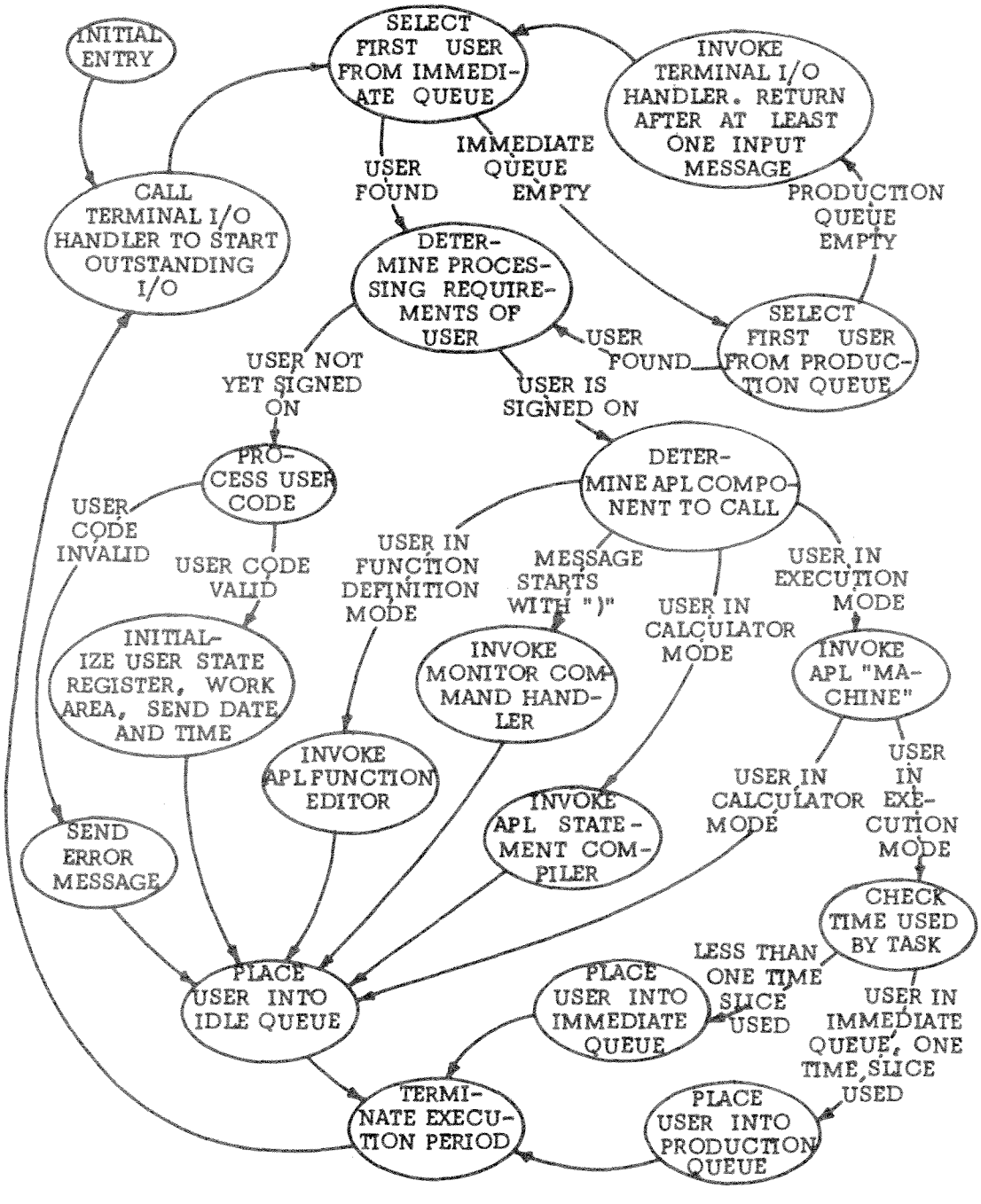


FIGURE 4

RESOURCE MANAGEMENT STATE DIAGRAM

allow user codes and user phrases to be added to or deleted from the APL system.

2. The Debug Bit. The debug bit can be set by a user operating in master mode. When this bit is set, various system diagnostic information is provided at the user's terminal.

3. The Nosyntax Bit. The nosyntax bit of the User State Register can be set with the "NOSYN" monitor command. Input lines typed by the user in function definition mode are not checked by APL for syntactic correctness when the nosyntax bit is set.

4. The Suspension Bit. The suspension bit is set when the user's APL execution encounters an error. The function in error is "suspended" and may be re-activated at a later time.

The remaining fields of the User State Register contain values of run time parameters along with variable information used by the various components of the APL system. The actual content of the "miscellaneous run parameters" field, shown in Figure 3a, is discussed in detail when the individual components are considered.

It should be noted that the system components, other than the Resource Manager, need not be concerned with keeping track of the active system users. Once the Resource Manager selects a user for execution the other system components refer to the User State Register indicated by the current user index in the User State Table. Thus, the individual components act upon data either located in the current User State Register or upon data addressed by fields within the current User State Register.

A simplified state diagram, given in Figure 4, shows the actions of the Resource Manager.

THE TERMINAL INPUT/OUTPUT HANDLER

The Terminal Input/Output Handler provides an interface between the APL system components and the terminal users. This interface includes message queueing facilities, I/O interrupt handling, input message scanning and translation, and output formatting capabilities. In addition, the Terminal I/O Handler adds items to the immediate queue for processing at the appropriate times. The I/O functions are grouped into the following types:

1. Message Queue and Table Maintenance. Information is maintained describing the status of each terminal port. I/O buffers and queues are also kept in order by this component.

2. Input Message Scanner. The scanner provides a common facility for extracting lexical elements from the input message corresponding to the current user.

3. Output Formatting Routines. All preparation of output messages from the APL system components is handled by the Output Formatting Routines.

The message queue and table maintenance component of the I/O Handler maintains the status of each terminal port in the Station Table, shown along with the relevant fields of a Station Element in Figure 5. The Terminal I/O Handler maintains the Station Table in order that it may determine for each terminal port:

1. if the station is physically connected to the B5500 system (physical connection bit),

2. if the terminal has an input message or messages waiting to be processed by APL (read ready bit),

3. if the terminal is set-up to accept a message (write ready bit),

4. if a message has been sent to a station but transmission has not been completed (output finish wait bit),

5. if some component of the APL system has requested that a message be readied for processing (input request bit),

6. if an APL component has passed a message through the I/O Handler to be written on the terminal (output request bit),

7. if the "break key" at the terminal has been depressed by the APL user (break key depression bit),

8. if an APL system component has acknowledged that the break key has been depressed, has taken the appropriate action, and has requested that the break key depression bit be reset (break key reset bit),

9. the number of messages from the terminal which have not yet been processed by APL (input queue size),

10. the number of messages produced by APL which have not yet been sent to the terminal because the station is not write ready (output queue size),

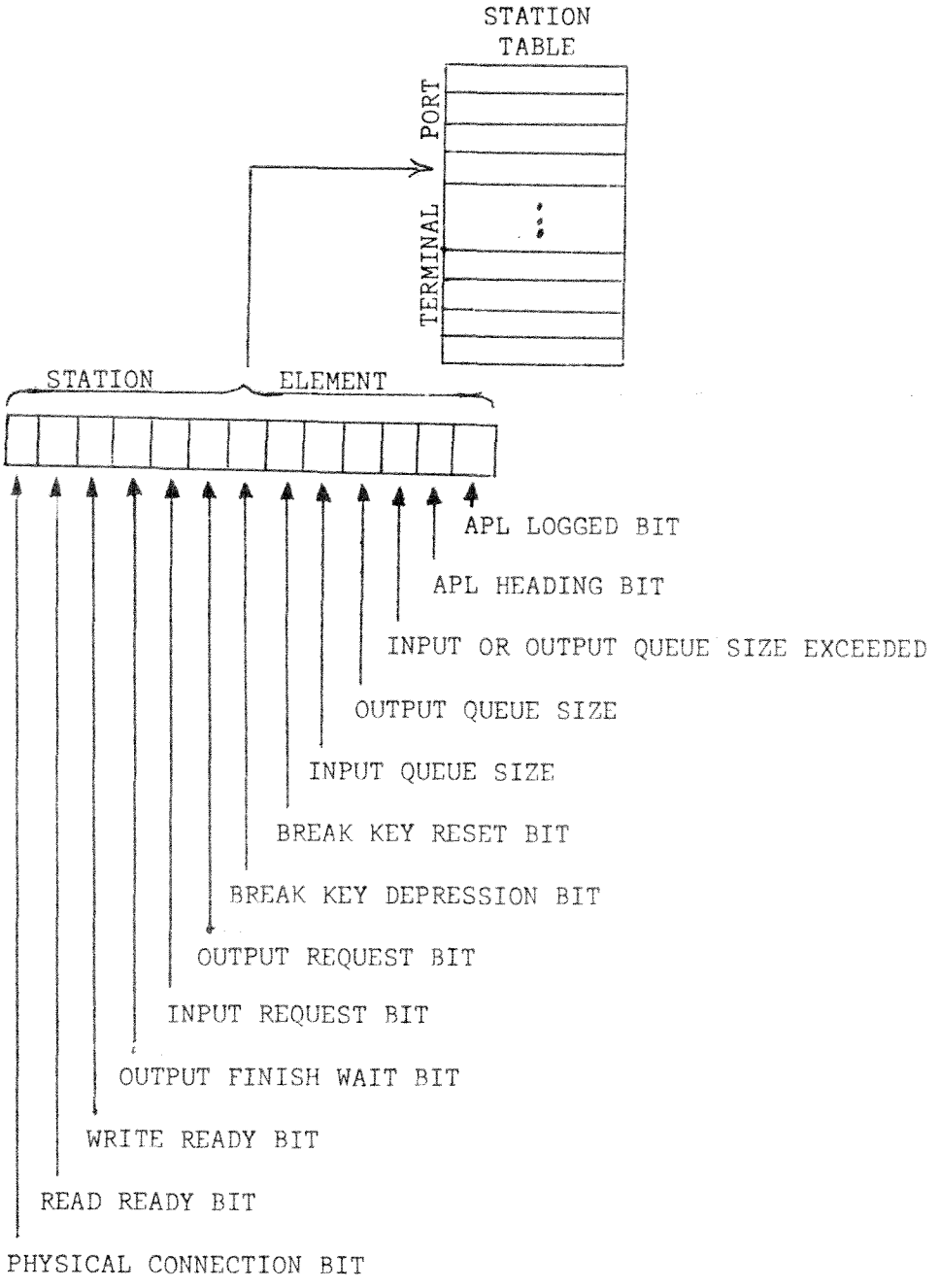


FIGURE 5

STATION TABLE WITH CORRESPONDING STATION ELEMENT

11. if the maximum number of messages in the input queue or in the output queue has been reached (input or output queue size exceeded bit),

12. if the APL\B5500 heading has been printed at the station (APL heading bit), and

13. if the user has successfully signed-on to the APL system (APL logged bit).

Thus, since there is a Station Element for each terminal port, the I/O Handler can immediately determine the I/O status of any terminal.

Input/Output buffers are maintained for active APL users as shown in Figure 6. An I/O queue is maintained on back-up storage with forward pointers (starting at the input or output buffer) connecting all elements of the queue for a particular user. The disk I/O queue may, at a particular point in time, contain both input and output messages in transit to or from the APL system.

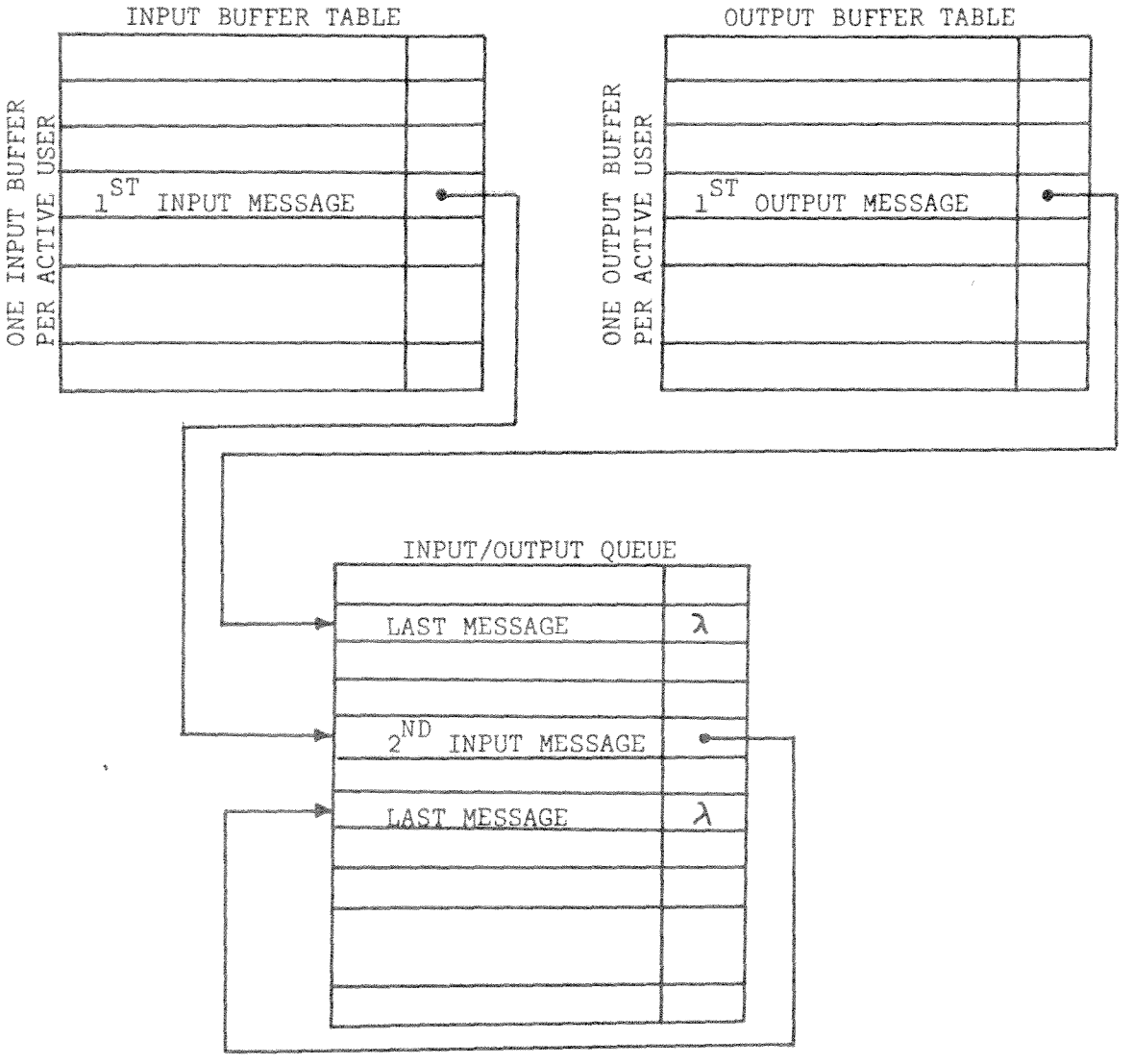
The I/O queues are, of course, maintained on a first-in first-out basis except when a time-slice "jiggle" is sent by an APL component. The jiggle is a null message which rattles the teletype typing mechanism letting the system user know that APL processing is in progress. The jiggle message goes to the front of the output queue for a particular user.

Since the terminal user normally waits for a response from APL for each line of input, the input queue will rarely contain more than one message.

The Terminal I/O Handler has access to the scheduling queues which are searched by the Resource Manager. Thus, when a particular user sends a message for APL processing while in the idle queue, the I/O Handler places the user in the immediate queue for processing. The Resource Manager allocates processor time to the user when the user gets to the front of the immediate queue.

The state diagram of Figure 7 shows the logic of the I/O Handler in processing terminal messages.

The scanning and formatting routines provide APL system component interface with the I/O Handler. The input message scanner provides lexical analysis of input messages upon request by APL components. The terminal input buffer referenced by the scanner is always that of the current user, as defined by the Resource Manager. The scanner provides translation from external symbols to internal coded form, identifies and converts positive and negative integer and real numbers as well as numbers in power-of-ten notation. In addition, APL variable, function, and command identifiers are isolated by the scanner. Except



Note: The end of list is denoted by "λ."

FIGURE 6

INPUT/OUTPUT BUFFERS AND QUEUES

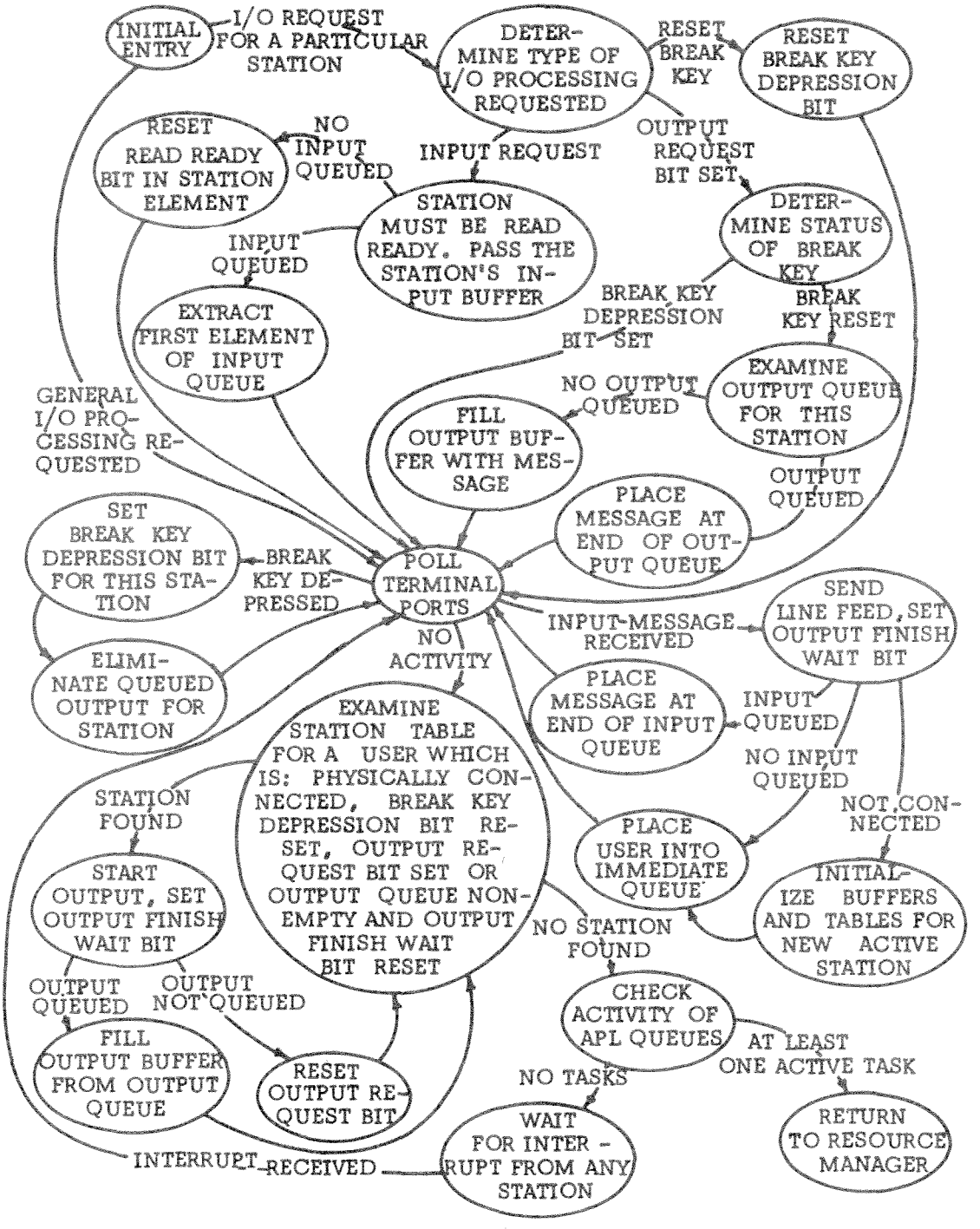


FIGURE 7

TERMINAL INPUT/OUTPUT HANDLER LOGIC

for string constants, all input to the APL components is processed by the input scanner.

Conversely, all output from APL components directed to terminal users is funneled through the output formatting routines. The formatting routines provide the APL system with primitive formatting capabilities; character strings are appended to the output buffer belonging to the current user according to the following output controls:

1. append characters to those already in the current user's output buffer, but do not send the message to the station (there is more to come),
2. append characters to the characters in the current user's output buffer and queue the message for output,
3. first send the contents of the current user's output buffer; place characters in the output buffer, but do not send the message,
4. first send the contents of the current user's output buffer to the terminal, place characters in the output buffer and send this second message also.

In conclusion, the Terminal Input/Output Handler processes all terminal messages sent from other APL components or sent to APL components from the users' terminals. Functional concurrency is maintained by initiating as many terminal message transfers as possible without causing unnecessary delays before returning to the Resource Manager.

APL VIRTUAL MEMORY MANAGEMENT

As mentioned previously, a fundamental design criterion was that APL\B5500 interfere as little as possible with normal B5500 operations. In particular, the central memory requirements for APL must be minimized without causing an excessive increase in overall response time. One solution to the storage problem might be to use the automatic overlay feature of the B5500 MCP. Automatic overlay, however, cannot be directly controlled by APL\B5500. Thus, the data areas used by APL are extended beyond the central memory areas through the use of an APL-suited virtual memory structure.

The APL Virtual Memory is a completely independent component of the APL\B5500 system, but is used in conjunction with a central memory data area

called the "scratch pad." The scratch pad data areas make use of the automatic overlay features of the MCP while the virtual memory is directly controlled by APL.

Although the physical structure of the virtual memory is described elsewhere[7], enough detail is given here in order that one may understand its use by the various APL system components.

The virtual memory is physically structured using simple demand paging techniques[8]. A file on back-up storage is divided into "pages" (the page size is determined at compile time) with an index to these pages residing in central memory. In addition, a number of central memory "page frames" are maintained in central memory to hold most-recently accessed pages. The number of page frames can be altered dynamically by other APL components, depending upon APL storage requirements and the number of active APL users.

Virtual memory access routines provide the interface between APL system components and the APL Virtual Memory. The virtual memory access routines give the virtual memory a conceptual structure which is quite different from the physical structure.

Conceptually, the APL virtual memory is divided into "storage units." The storage units can be created dynamically by APL components and are of no predetermined size (except for the maximum extent of the disk file).

The storage units, in turn, can be one of two types: "ordered" storage or "sequential" storage. Ordered storage units contain data in tables appearing to the APL components as contiguous alphabetically arranged data with fixed field lengths. Sequential storage units contain data elements of variable length, but are only accessible through a fixed address within the storage unit. The maximum number of storage units which can be active at any given time is 512.

Figure 8 shows the conceptual structure of virtual memory. Ordered and sequential storage units are often related through a right-most field in ordered storage unit elements. This field might contain the address of a data element in an associated sequential storage unit, although this assumption is not made by the virtual memory routines.

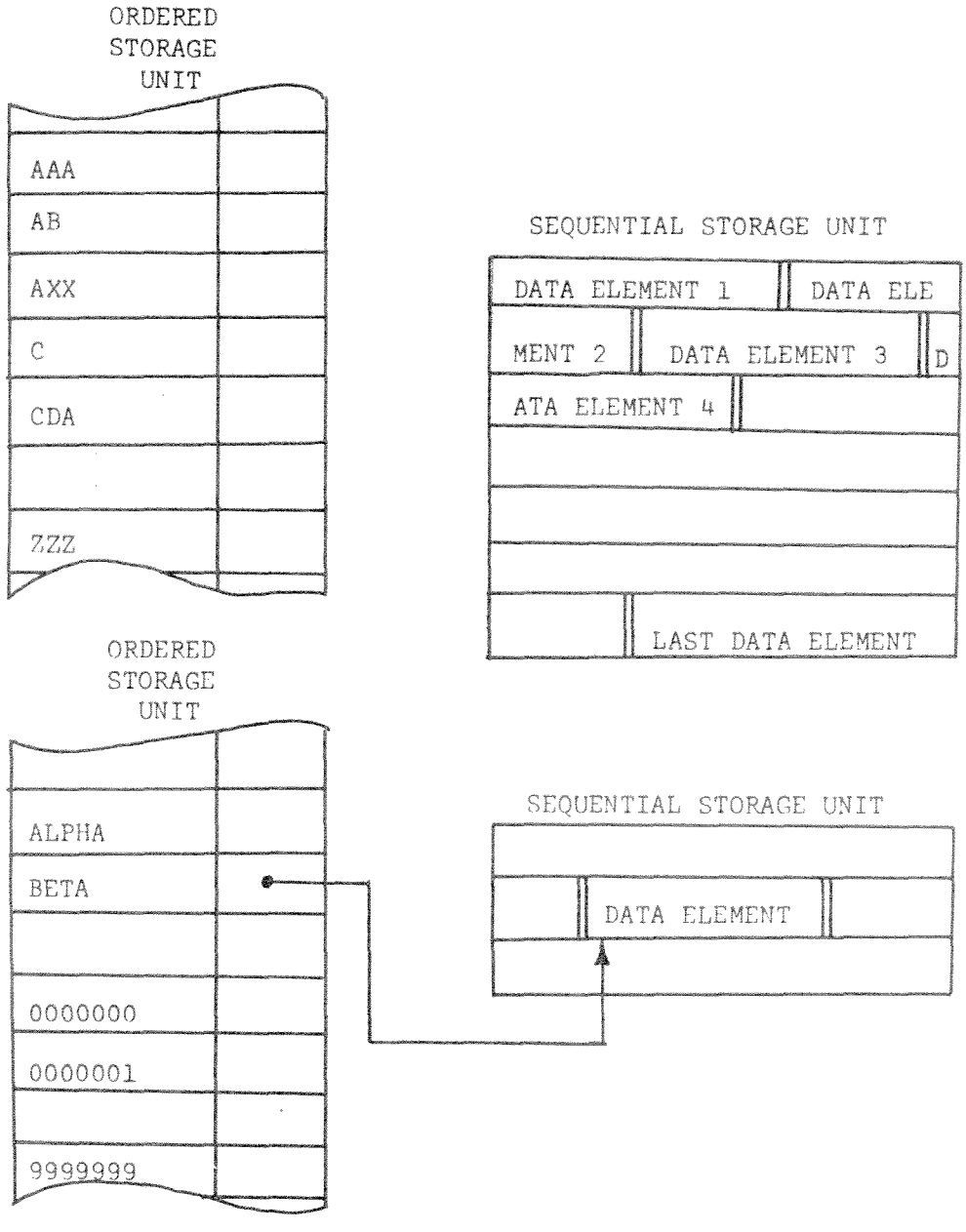


FIGURE 8

CONCEPTUAL STRUCTURE OF VIRTUAL MEMORY

APL initializes with storage units one and two reserved for user codes and associated user phrases as shown in Figure 9. Initially, storage unit one contains only the system supervisor's user code, and unit two contains his user phrase. Units one and two increase in size as the system supervisor adds more user codes and user phrases.

The services provided for APL components by the virtual memory access routines can be categorized as follows:

1. Storage Maintenance.
   a. A particular direct access disk file can be named for use as a virtual memory back-up storage area.
   b. Particular storage units can be created and destroyed.
   c. The number of active page frames can be increased or decreased.
   d. APL components may designate that vital information, necessary for proper system recovery in case of failure, be written onto back-up storage.
2. Storage Interrogation and Alteration.
   a. Variable-length data can be stored in a specified sequential storage unit.
   b. Ordered storage units may be searched for a particular data item on demand by an APL component.
   c. Information can be inserted into a specified ordered storage unit.
   d. The contents of a particular address in either ordered or sequential storage can be retrieved.
   e. Elements in either ordered or sequential storage units can be deleted.
   f. Entire storage units can be erased with the corresponding data areas added to free storage.
3. Storage Utility Functions.
   a. The number corresponding to the next available storage unit can be obtained by an APL component. The unit can then be used for storage.
   b. The size (number of data elements) in a specific storage unit can be requested by an APL component.

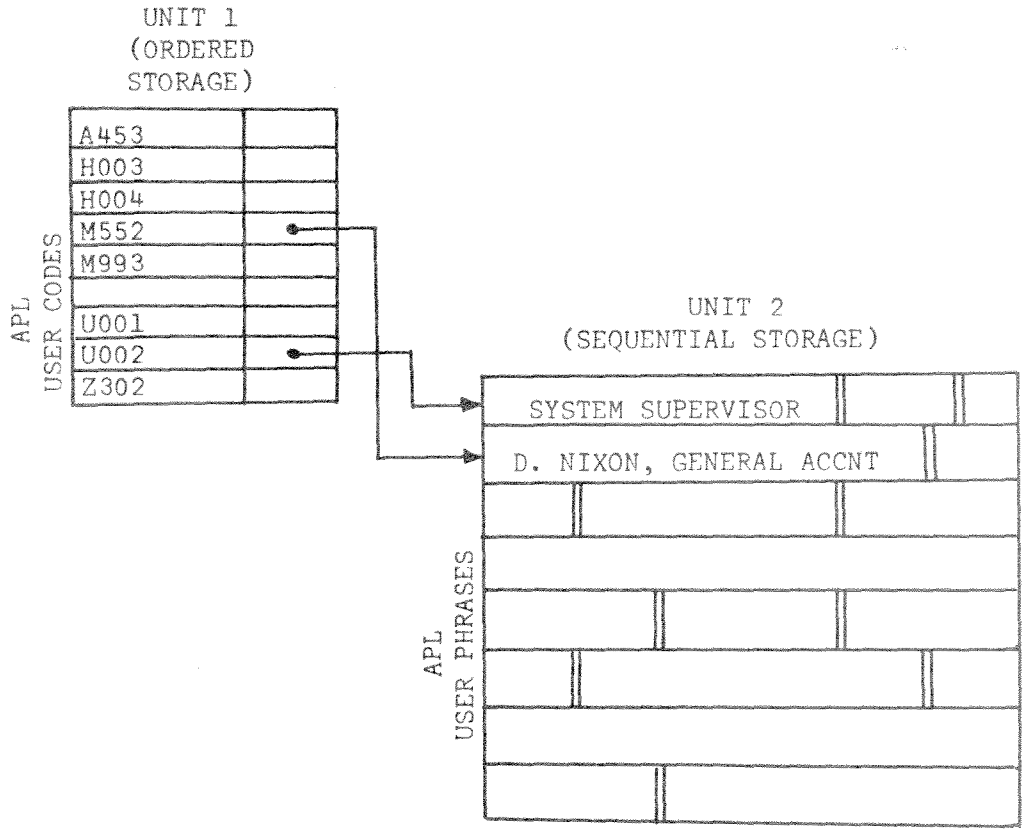


FIGURE 9

USER CODES AND USER PHRASES

c. The mode of a particular unit can be determined (i.e., whether the unit has been designated as an ordered or a sequential storage unit).

The work area of an APL user may consist of several ordered and sequential storage units. At sign-on time, the user is assigned an ordered storage unit, called "names," and a sequential storage unit, called "data." The "names" unit contains variable and function names, along with additional information about the names. The "data" storage unit holds numeric and character array results computed during the APL run. As shown in Figure 10, the "data" storage unit also contains a "recent" copy of the user's User State Register. This recent copy of the User State Register allows a user to restart an APL session with very little loss of work in the case of a hardware or software failure.

The important concept here is that at any point in the APL run, a portion of the user's work area, consisting of ordered and sequential storage units, is located in central memory page frames while the remainder is located on back-up storage. The portion in central memory is based entirely on data access activity. This, of course, is a fundamental concept in any implementation of demand paging. The scheme does, however, allow the storage units to become much larger than would be possible if all tables and data were to remain in central memory.

It will become evident in later sections that the ordered and sequential storage units, along with the virtual memory access routines are well-suited to the needs of the APL system.

As a final note, the APL Virtual Memory Manager, like other APL components, maintains functional concurrency. When the virtual memory manager has back-up storage maintenance to perform, it does so in small increments each time it is called. Thus, control returns to the Resource Manager as soon as possible.

THE APL FUNCTION EDITOR

The APL Function Editor component of the APL\B5500 system provides function definition and editing capabilities for APL users. The Function Editor

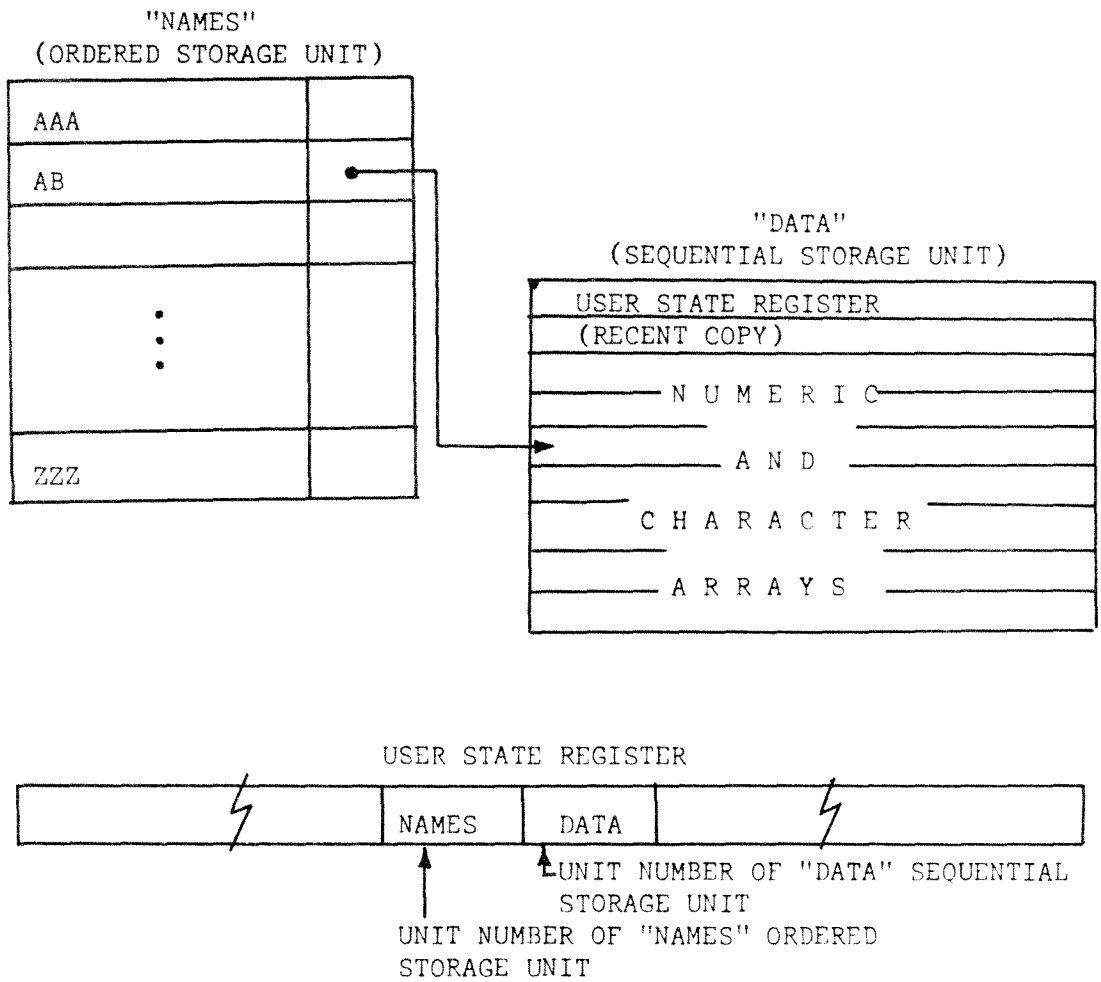


FIGURE 10

STORAGE UNITS FOR NAMES AND DATA

handles the syntax of the function header and creates internal data structures from the header to pass to the other APL components. The Editor relies upon the virtual memory access routines in implementing the editing functions.

Every function defined by the APL user causes two units of storage to be allocated: an ordered storage unit called a "function label unit," and a sequential storage unit called a "function text unit." The function label unit contains entries corresponding to the line numbers of the function along with addresses of lines of function text in the corresponding function text unit.

Figure 11 shows the interconnection of the function label unit and the function text unit. The left-most field of each entry of the function label unit contains the line numbers in full character form (without a decimal point). The right-most field contains the address of the corresponding line in the function text unit. Note that the function header is assumed to be line zero of the function.

Addition and deletion of text and line numbers is accomplished by using the corresponding virtual memory access routines.

The Function Editor also keeps track of local variables and labels in functions. As shown in Figure 12, a number of cases occur:

1. the local variables are all marked with a right-most field in the function label unit which is less than or equal to zero:
    a. the local variable which contains the value to be returned at the end of function execution is marked with a negative one,
    b. the arguments (formal parameters) are marked with a minus two and minus three,
    c. all other local variables are marked with zeroes;
2. labels are marked with the full character representation of their corresponding line numbers.

Case (2) above allows access to lines of text through the line labels.

Like all other APL system components, the Function Editor must maintain functional concurrency. Clearly, there are many situations where functional concurrency becomes a problem (e.g., displaying lines of text). Thus, the Function Editor maintains a number of variables in the User State Register,

FUNCTION DEFINITION

```
      $STDEV
[1]   AVE := (+/X)% N :=RHO X
[2]   ( (+/(X-AVE)*2)%N-1 )*.5
[3]   $
```

STORAGE UNIT STRUCTURE

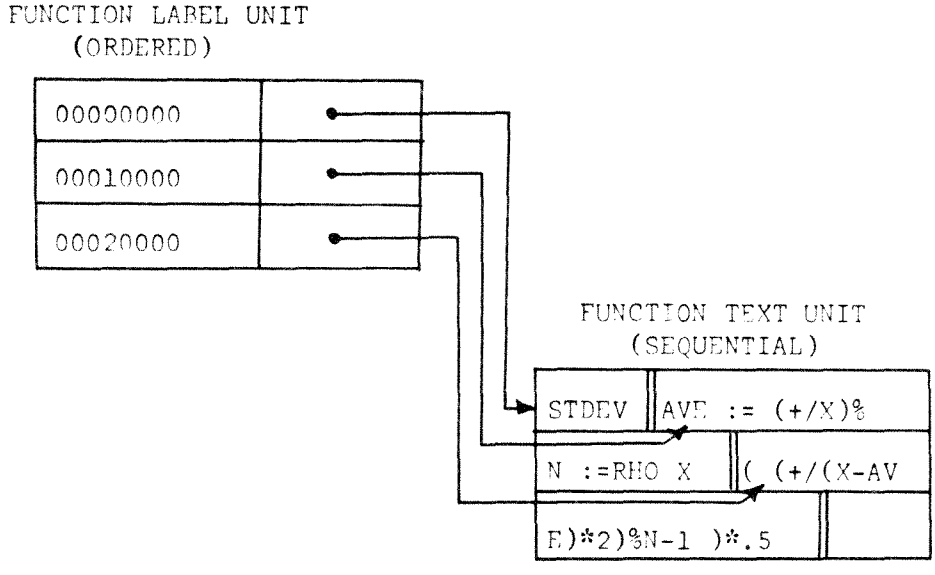
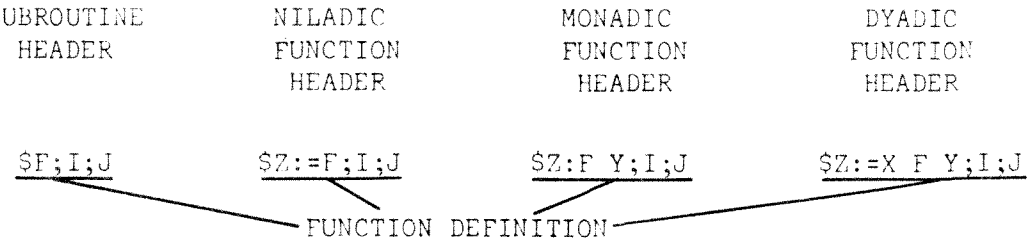


FIGURE 11

FUNCTION STORAGE UNITS




```
[1]   L: X:=IOTA 3
[2]   Z:=X RHO 1
[2.1] M: N: Z+5$
```

FUNCTION LABEL UNITS
(ORDERED STORAGE)

SUBROUTINE STRUCTURE

I	00000000
J	00000000
L	00010000
M	00021000
N	00021000
00000000	D_1
00010000	D_2
00020000	D_3
00021000	D_4

NILADIC FUNCTION STRUCTURE

I	00000000
J	00000000
L	00010000
M	00021000
N	00021000
Z	-1
00000000	D_1
00010000	D_2
00020000	D_3
00021000	D_4

DYADIC FUNCTION STRUCTURE

I	00000000
J	00000000
L	00010000
M	00021000
N	00021000
X	-3
Y	-2
Z	-1
00000000	D_1
00010000	D_2
00020000	D_3
00021000	D_4

MONADIC FUNCTION STRUCTURE

I	00000000
J	00000000
L	00010000
M	00021000
N	00021000
Y	-2
Z	-1
00000000	D_1
00010000	D_2
00020000	D_3
00021000	D_4

NOTE: F is the function name, I and J are local variables, X and Y are formal parameters, and Z denotes the value returned by F. The values D_1, D_2, D_3, D_4 represent the addresses of the corresponding lines of text in the function text unit.

FIGURE 12

FUNCTION LABEL UNIT STRUCTURE

as shown in Figure 13. The Function Editor fields are only defined when the user is in function definition mode when they contain:

1. the number of the ordered storage unit assigned as the function label unit,
2. the number of the sequential storage unit assigned as the function text unit,
3. the name of the function being edited,
4. the current line of the function being defined,
5. the current line increment for this user,
6. the editing "submode" (i.e., deleting text, editing, or displaying lines of text),
7. the editing submode boundaries (e.g., the starting and ending line numbers for the display command).

A corresponding entry is made in the "names" ordered storage unit for this user as soon as the function definition is closed. The entry for a function consists of the name of the function in the left-most field and the numbers corresponding to the function label unit and the function text unit in the right-most field.

Note also that the Function Editor examines the nosyntax bit of the user mask whenever a new line is inserted or an old line is edited in a function. If the nosyntax bit is reset then the Editor passes the APL statement to the APL Statement Compiler for a syntax check. The user is notified if errors are detected.

The state diagram of Figure 14 shows the basic logic of the Function Editor.

THE APL MONITOR COMMAND HANDLER

After a terminal user has initially signed-on to the APL\B5500 system, the Resource Manager passes all messages which begin with a ")" to the Monitor Command Handler for processing. The Monitor Command Handler processes the monitor commands shown in Table 2, along with the system supervisor commands listed in Table 4. The entire set of monitor commands can be categorized as:

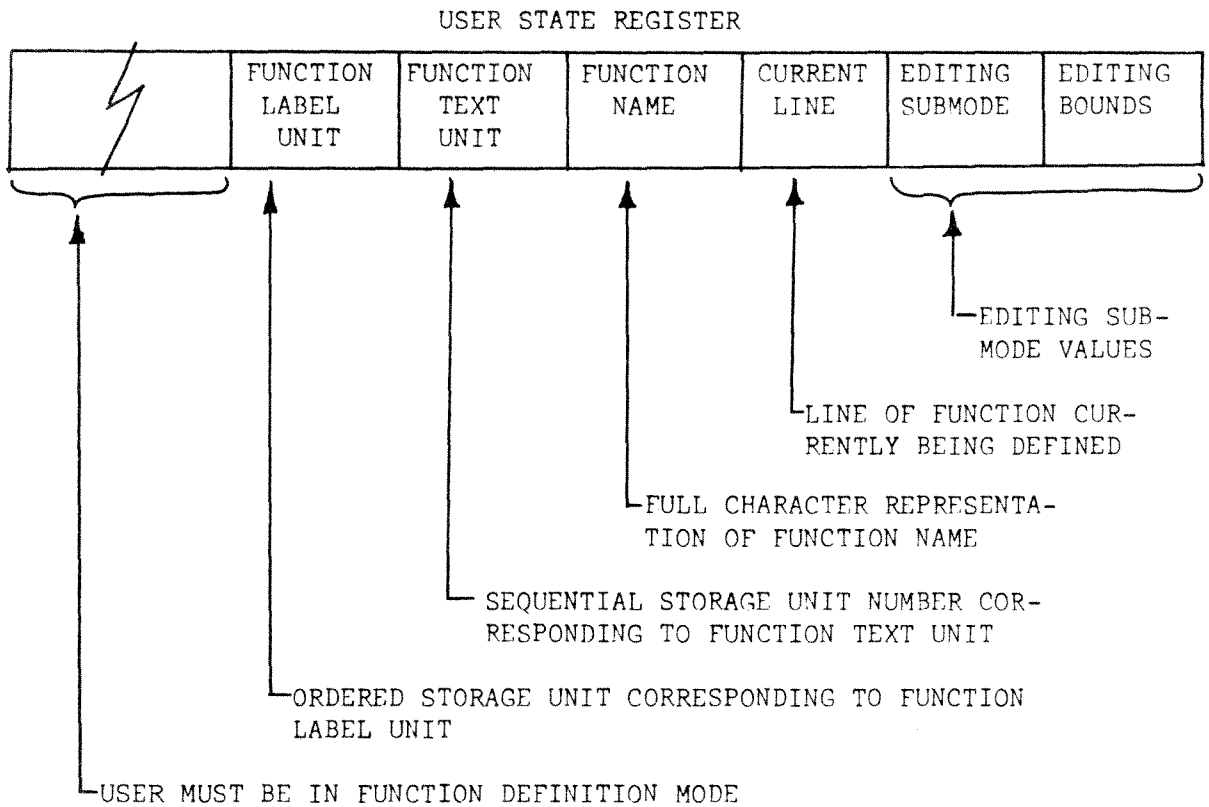


FIGURE 13

USER STATE REGISTER ENTRIES FOR THE FUNCTION EDITOR

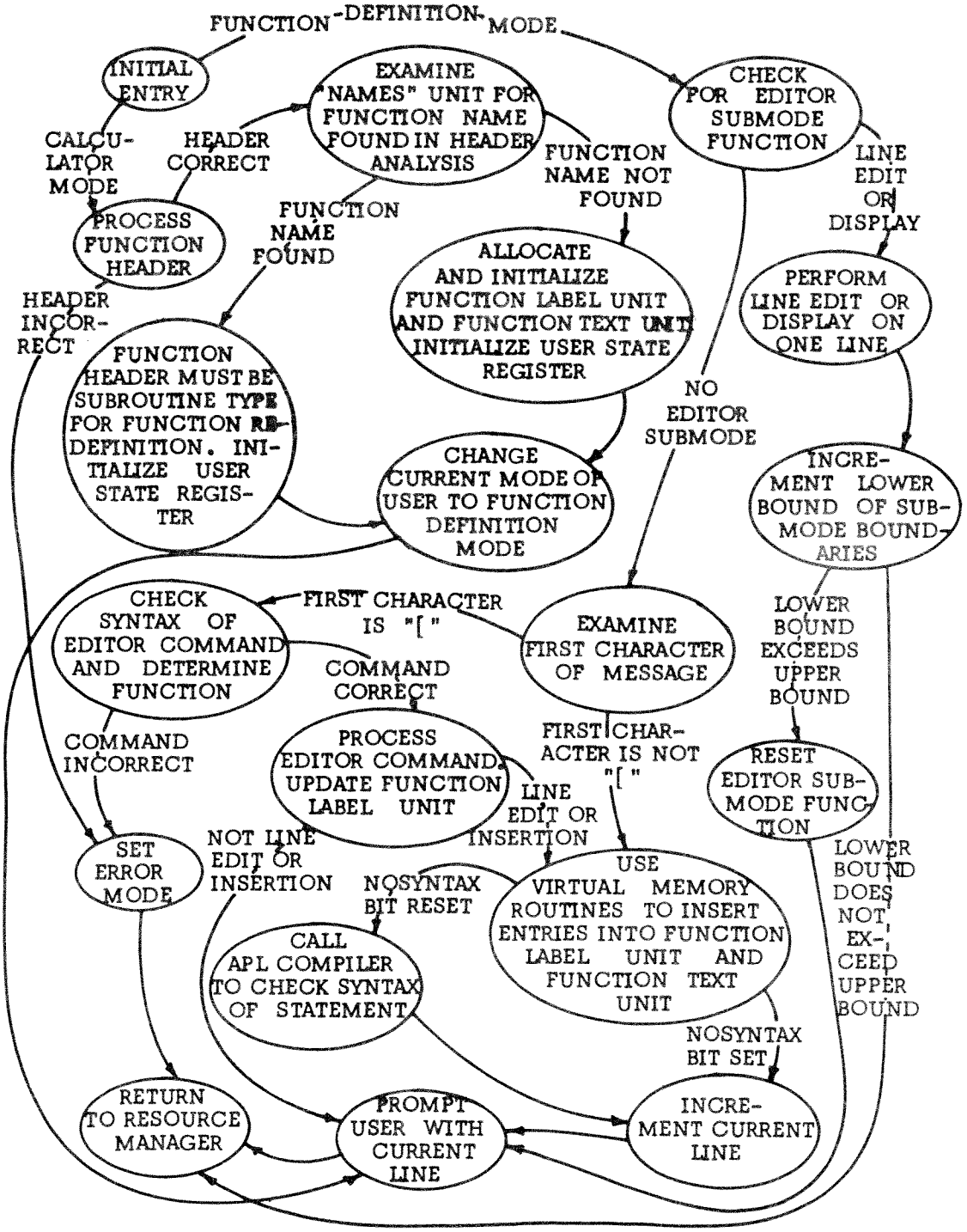


FIGURE 14

FUNCTION EDITOR STATE DIAGRAM

TABLE 4

PRIVILEDGED MONITOR COMMANDS

MONITOR COMMAND	COMMAND FUNCTION
)ASSIGN <user code> <user phrase>	This command assigns a new user code to be recognized by APL. The <user code> goes into the user code ordered storage unit. The <user phrase> goes into the user phrase sequential storage unit, and serves to identify the user to other APL users.
)DELETE <user code>	This command removes a user code and associated user phrase from the APL system.
)LIST CODES	This command provides a listing at the system supervisors terminal of all assigned user codes.
)LIST USERS	This command provides a complete listing of all assigned user codes and user phrases.
)DEBUG MEMORY <integer>	This command specifies that a trace of APL virtual memory activity be given. The <integer> specifies trace options.
)DEBUG POLISH	This command causes the APL statement compiler to print a trace of the code produced for each APL statement executed by the system supervisor.

1. System Maintenance Commands. The system maintenance commands allow the APL system supervisor to add, delete, and alter user codes and user phrases. In addition, the supervisor can set system diagnostic flags. These commands are recognized only when the master mode bit is set in the user's User State Register.

2. Work Area Maintenance Commands. Work area maintenance commands allow the APL user to add or delete items from his associated work area. The user may also save the work area in a separate file, and later reactivate the work area.

3. APL Run Parameter Specification. Variables which affect the APL run for a user can be displayed and altered through APL monitor commands (e.g., "WIDTH" and "ORIGIN").

4. Line Edit Command. The last line entered by each user can be altered and re-submitted, as discussed previously using the Line Edit command.

5. Function Suspension Commands. The function suspension commands allow the user to control function execution when functions have been suspended due to errors.

6. Run Termination Commands. The APL user may terminate the APL run using a number of different options.

The implementation of most of the monitor communication algorithms is straightforward. It is useful, however, to examine the data structures involved in these operations.

If the monitor command to be executed is a system maintenance command, the master mode bit of the User State Register for the current user is examined. If this bit is reset then the user is issued an error message. Otherwise the Monitor Command Handler uses the virtual memory access routines to examine, add to, or delete from the user code ordered storage unit and the user phrase sequential storage unit.

The work area maintenance commands access the "names" ordered storage unit. The variables and functions can be listed and deleted by application of the appropriate virtual memory access routines. In addition, the total content of the work area may be copied to an external library for later use.

This operation involves accessing and copying all ordered and sequential storage units allocated for the user's work area. The "names" ordered storage unit provides an entry point for referencing all variables and functions. The library is constructed by first constructing a dictionary, as shown in Figure 15. All non-scalar data is copied into the library from the "data" sequential storage unit, with appropriate addresses in the library dictionary. Whenever functions are encountered in the "names" unit, the corresponding function label unit and function text unit are accessed through the unit numbers in the right-most field of the function entry. The line label, along with the function text for each line, is forward-chained for each function. The dictionary entry for the function addresses the head of this chain.

Library load and copy operations reference the directory of a particular library to obtain addresses and data lengths in the library. The load and copy operations occur in just the opposite order from the save operation. The situation arises, however, when copying functions into an active work area, where the function name being copied is identical to a variable name occurring in the "names" unit. In this case, the variable, along with the corresponding data, is removed from the work area before copying the function.

All of the above operations make use of the virtual memory access routines in searching and altering storage units.

The APL run parameter specification commands are easily implemented. Display is accomplished by referencing the corresponding field of the User State Register (e.g., the "digits" field). Similarly, the fields may be altered directly on command by the user. Thus, if the command is "DIGITS," the "digits" field is retrieved from the User State register and displayed. If the message typed by the user is "DIGITS 3" then a new value of three is inserted in the user's User State Register.

The Line Edit command is implemented by retaining a copy of the last message typed by the user in calculator mode (initially the null message). The Line Edit is processed according to the rules given earlier, and a "simulated" teletype input is performed with the new edited line. The simulated input, however, goes to the beginning of the input queue for the user. The Line Edit command does not replace the last message typed by the user; hence,

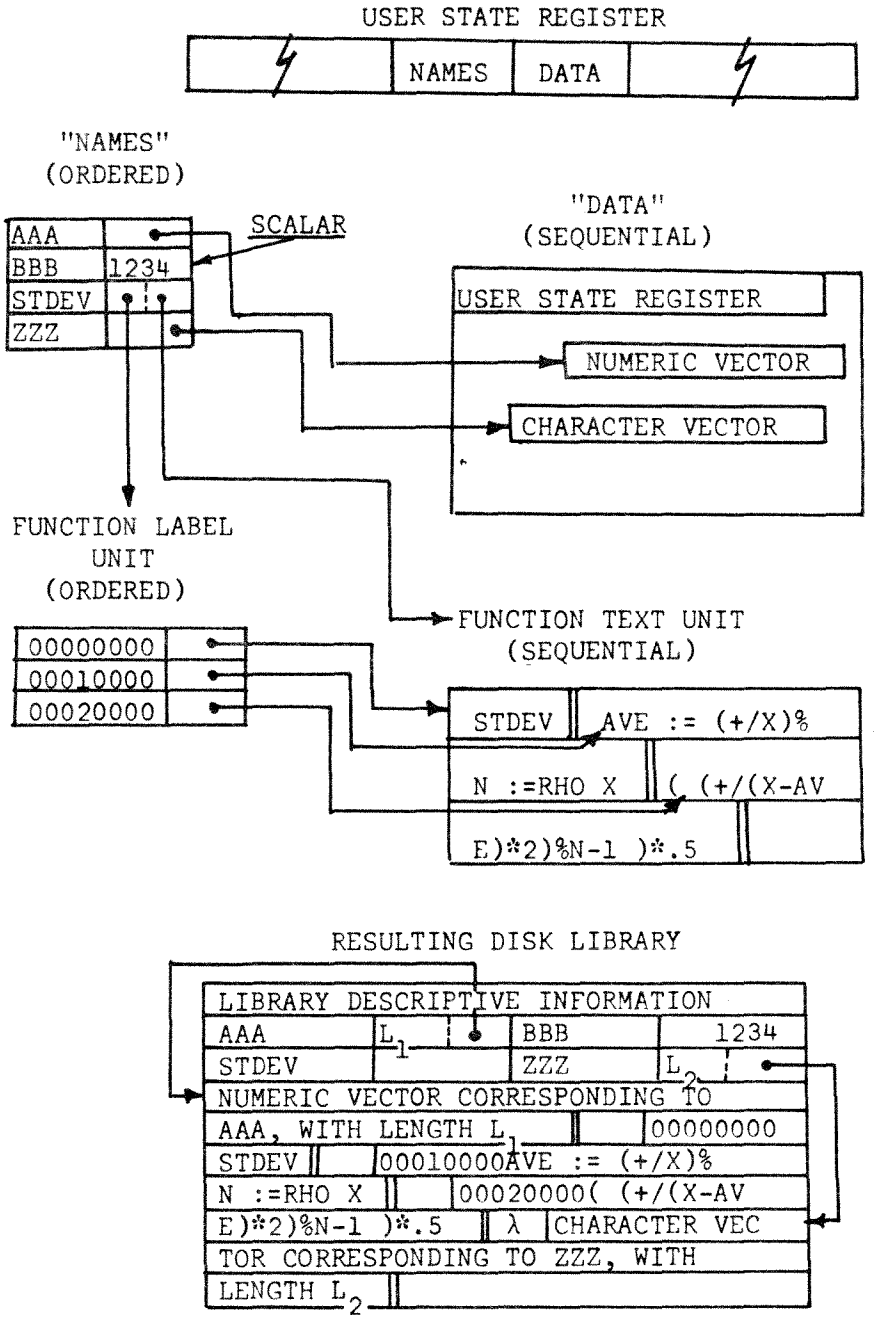


FIGURE 15

THE FORMAT OF A LIBRARY

it is possible to edit the same line several times.

User to user communication is made possible with two monitor commands: the "LOGGED" command and the "MSG" command. The first command displays the user phrases corresponding to each active APL user, along with the user's station number. The Monitor Command Handler refers to the user phrase storage unit to obtain this information.

The users may communicate as shown in Figure 16. The user specifies the station number with the "MSG" command of the user which is to receive the message. The message is extracted from the originator's input and placed (with the proper prefix) at the beginning of the output queue of the station receiving the message.

The last command to consider is the "OFF" command. This command informs the APL system that the user wishes to discontinue the APL session. Two options are available:

1. "OFF," and
2. "OFF DISCARD."

In case (1), APL assumes the user wishes to be physically disconnected from the system with the active work area saved under the library name "CONTINUE." The appropriate bits are reset in the station table entry for the port, and the library is constructed. A termination message is then printed, followed by deallocation of data areas (storage units, buffers, and registers). Case (2) is similar to the first except that a library is not constructed. In either case, APL assumes that some user wishes to sign-on again after a short period. The terminal is not physically disconnected, and the buffers are retained for this port until a fixed time has elapsed without a sign-on at the terminal.

The monitor command handler is distinct from the other APL components, but provides a command language and command facilities which are useful in the APL environment.

In conclusion, it can be easily seen that the Monitor Command Handler makes use of the virtual memory access routines in the implementation of nearly all the commands.

```
      FROM (2): ARE YOU GOING TO BE WORKING LATE TONIGHT...
      )MSG 2 I THINK I WILL QUIT ABOUT MIDNIGHT←
      X←
3
      FACT X*2←
362880
      LOG 473←
6.1591
      FROM (2): HAVE  YOU FINISHED THENUMERICAL ANALYSIS ASSIGNMENT...
      )MSG 2 I ALMOST HAVE THE BIG ANSWER←
      LØG 474←
SYNTAX ERROR AT  474
      LOG 474←
6.16121
      (LOG 3)+(LOG 4)←
      2.48491
      LOG 12←
2.48491
      FROM (2): DO YOU HAVE A SAVED COPY THAT I CAN COMPARE WITH.
      )MSG 2 YES I SAVED ONE ABOUT 1 HOUR AGO←
      LOG 362880←
12.80183
      FROM (2): IS IT UNLOCKED...  WHAT IS  THE  NAME  OF THE LIBRARY..
      )MSG 2 IT IS INTERPOLLY...READY TO GO←
      )VARS←
 INTERP (F) STRING  X        Y
      STRING←
A VERY FAT CAT

      INTERP←
FROM (2): OK... I AM GOING TO LOAD IT...
      INTERPOLATION PROBLEM C1

INPUT X VALUES

[]:
      V:= 2 4 6 8 10 12 15 20←
INPUT Y VALUES

[]:
      LOG V←
INPUT VALUE TO INTERPOLATE

[]:
      13←
INTERPOLATED VALUE IS 2.56564

      LOG 13←
2.56495
      )OFF←
      END OF RUN
```

FIGURE 16

USER/USER COMMUNICATION

STORAGE AND REPRESENTATION OF APL DATA STRUCTURES

The methods used in data storage and representation are fundamental in the understanding of the two APL components remaining to be discussed: the APL Statement Compiler, and the APL "Machine."

The fast-access data area mentioned earlier, called the scratch pad, contains data which is "active." Further, each data item residing in the scratch pad has an associated "descriptor" which gives the characteristics of the data. The organization of the scratch pad, data layout, and descriptor formats are the subjects of this section.

The scratch pad may be considered the memory of the simulated APL machine. The scratch pad is, in fact, an array which increases and decreases in size as the requirements for working storage increase and decrease. Space is allocated within the scratch pad using a variation of simple segmenting[9].

All APL data in a particular user's work area can be considered "active" or "passive." Data can be active for a user only when the user is executing an APL statement or function, and the data has been referenced during the execution. Passive data is that data which can be referenced through the "names" ordered storage unit assigned to the user. References to passive data may occur when the user is executing an APL statement. In this case, a copy of the passive data is brought into the scratch pad during the computation. Active data in the scratch pad may replace passive data in the user's work area at the end of execution (i.e., the user returns to calculator mode from execution mode). In addition, new results may have been computed during execution causing additions to the "names" and "data" storage units.

At any point in the execution of several users' APL programs, the scratch pad contains active data for all of these users. The passive data, however, is kept distinct in the individual "names" and "data" storage units referenced through the corresponding User State Registers.

Data which is active in the scratch pad is identified through the use of "descriptors." The descriptors, shown in Figure 17, identify data by providing the following information:

1. Descriptor Identification Bit. The descriptor identification bit is set if the descriptor refers to APL data.

2. Data Presence Bit. The data presence bit is set when data corresponding to the descriptor is present in scratch pad memory.

3. Named Bit. The named bit is set in a descriptor when the data associated with a descriptor is not a temporary result.

4. Scalar Bit. The scalar bit is set in descriptors which reference scalar data.

5. Character Bit. The character bit is set in a descriptor when the descriptor refers to a character array rather than numeric data.

6. Back Pointer Field. The back pointer field is primarily used to identify the origin of the descriptor in scratch pad memory.

7. Rank Field. The rank field of a descriptor contains the number of dimensions in the data associated with the descriptor.

8. Scratch Pad Field. The scratch pad field holds the actual scratch pad address of the data associated with the descriptor.

The size of each field depends upon the maximum value that can be assumed in each case.

Data in array form is stored in row-major order with the dimensionality of the array in the first few locations, as shown in Figure 18.

The use of descriptors allows execution-time determination of the complete meaning of a particular operator. Thus, the meaning of the statement

X+Y

cannot be exactly determined at compile-time since the "+" could represent a scalar-scalar, scalar-array, or array-array operation. The exact operation is determined at execution time by examining the data descriptors involved in the operation.

The use of descriptors is also extended to APL functions. Referring again to Figure 17, function descriptors contain the following information:

1. Descriptor Identification Bit. The descriptor identification bit is reset for function descriptors.

2. Argument Field. The argument field contains the number of arguments (parameters) required for function execution.

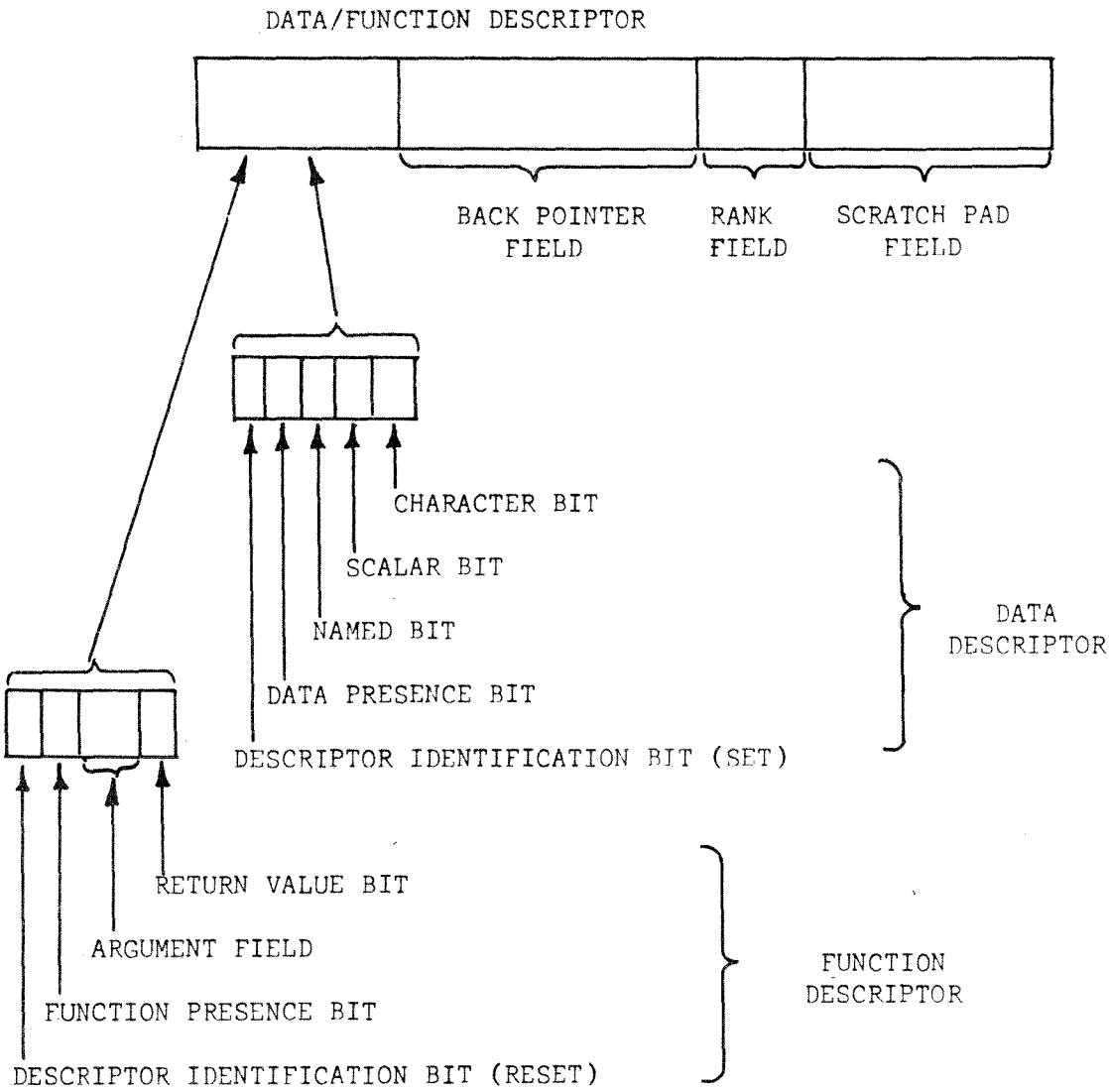


FIGURE 17

DATA AND FUNCTION DESCRIPTORS

APL STATEMENT

2 3 RHO IOTA 6

SCRATCH PAD REPRESENTATION

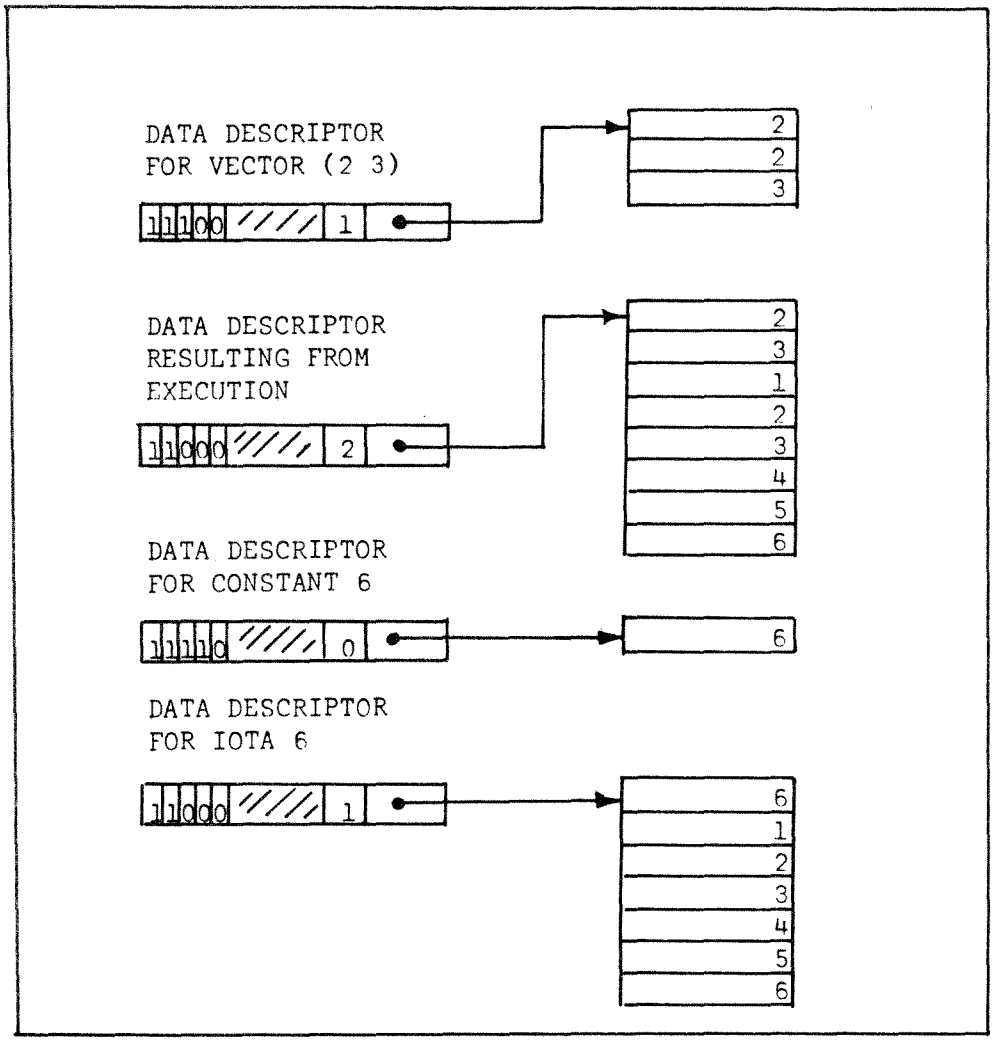


FIGURE 18

SCRATCH PAD DATA REPRESENTATION

3. Return Value Bit. The return value bit is set for function descriptors corresponding to functions which return a value from execution.

The presence bit, back pointer field, and scratch pad field are used in the same manner as in the data descriptor.

Descriptor access is accomplished through the symbol tables described in the following section.

ACTIVE AND PASSIVE SYMBOL TABLES

Corresponding to active and passive data and functions in APL\B5500 there are active and passive symbol tables. The passive symbol table is just the "names" ordered storage unit shown in Figure 10. The details of the passive symbol table entries are shown in Figure 19. The contents of the right-most field of a passive symbol table entry depends upon the type of entry. In particular, a non-present (presence bit reset) data or function descriptor may appear with the name, or simply a scalar value will appear if the name represents a scalar.

Each passive symbol table entry is identified by the entry identification field. The entry identification field may take on one of the following values:

1. Scalar. The name corresponding to the entry is a scalar. The scalar value is contained in the right-most field of the passive symbol table entry.
2. Array. The entry corresponds to an array variable. The right-most field contains a non-present data descriptor. The scratch pad field contains an address in the "data" sequential storage unit where the corresponding data can be found. The data is loaded into the scratch pad when the variable becomes active and is accessed.
3. Function. The entry represents a defined function. The right-most field contains a non-present descriptor. The back pointer field, however, contains the unit number of the function label unit, and the scratch pad field contains the number of the function text unit corresponding to the function.

The passive symbol table is always searched using the virtual memory access routines.

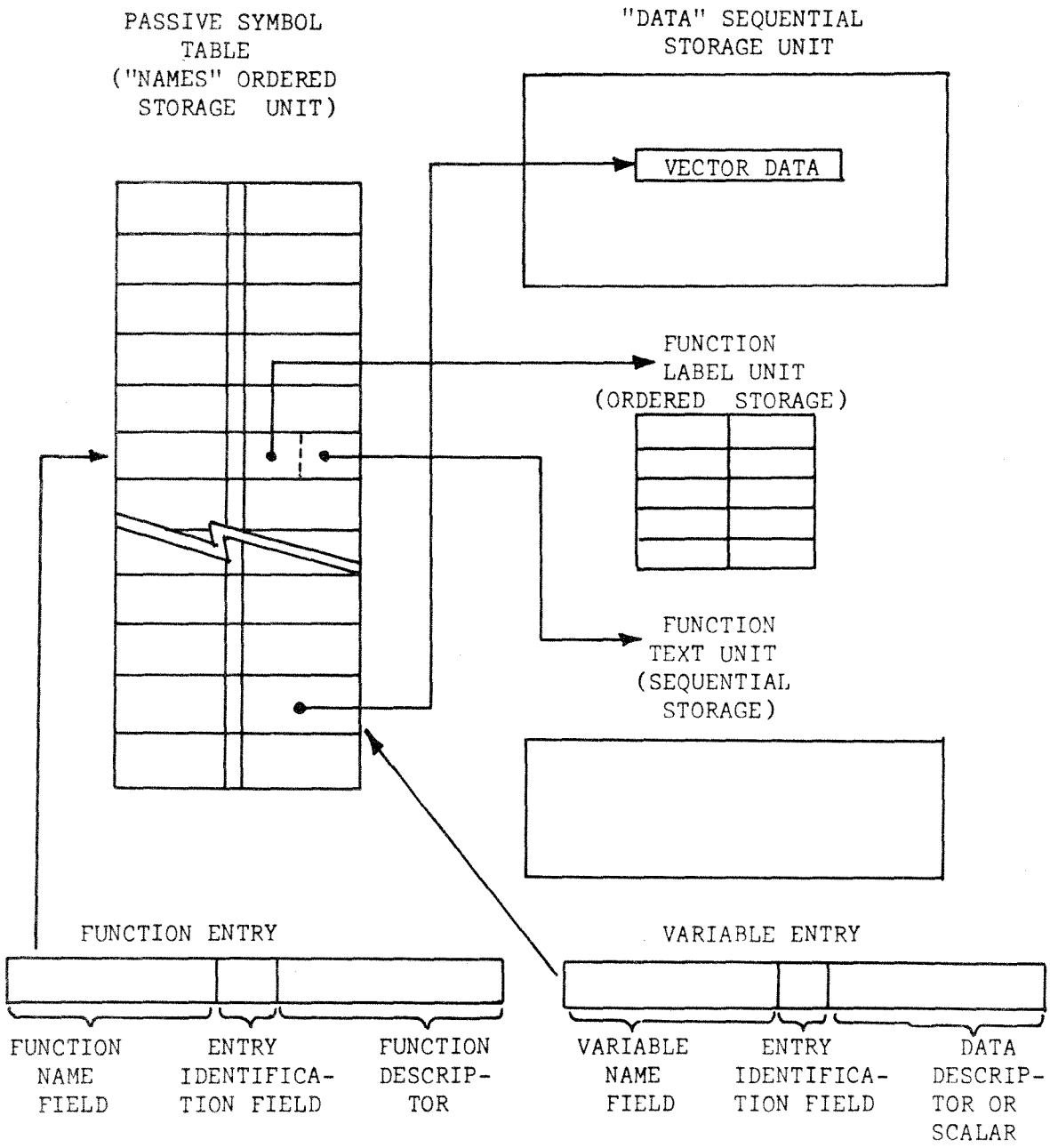


FIGURE 19

PASSIVE SYMBOL TABLE

The active symbol table exists for a particular user only during execution of a statement or function. The active symbol table, shown in Figure 20, is located in the scratch pad and is addressed through the symbol base field of the user's User State Register. At any given time, there may be several active symbol tables in the scratch pad; one for each user of APL in the process of executing APL statements. The active symbol table contains entries for the active data, not including constants, temporary results, or local variables. The descriptors in the active symbol table may or may not have their presence bits set.

The entry identification field (switched to the front of the name for machine-dependent reasons) has an additional bit position, called the "altered bit," in the active symbol table. The altered bit indicates whether or not changes have been made to data which is active and thus needs to be changed in the "data" storage unit. In addition, the altered bit is set for variables which are created during execution and do not yet exist in the passive symbol table. All variables with their altered bit set are changed or entered in the passive symbol table when the user returns to calculator mode from execution.

Another important symbol table used in compilation and execution of APL statements is called the function label table, shown in Figure 21. The function label table is essentially an extension of the active symbol table in the scratch pad. A function label table is constructed whenever an APL statement references a function with the presence bit reset in the corresponding function descriptor. The function descriptor is replaced by a data descriptor referencing the function label table as soon as the table is constructed. The information found in the function label table is derived from the function's corresponding function label unit.

The function label table is logically an APL numeric vector referenced by the (now present) data descriptor in the active symbol table. The rightmost fields of the numeric labels are initially all non-present data descriptors with the scratch pad fields referencing the corresponding lines of text in the function text unit.

Thus, it is possible to address all variables and functions through the

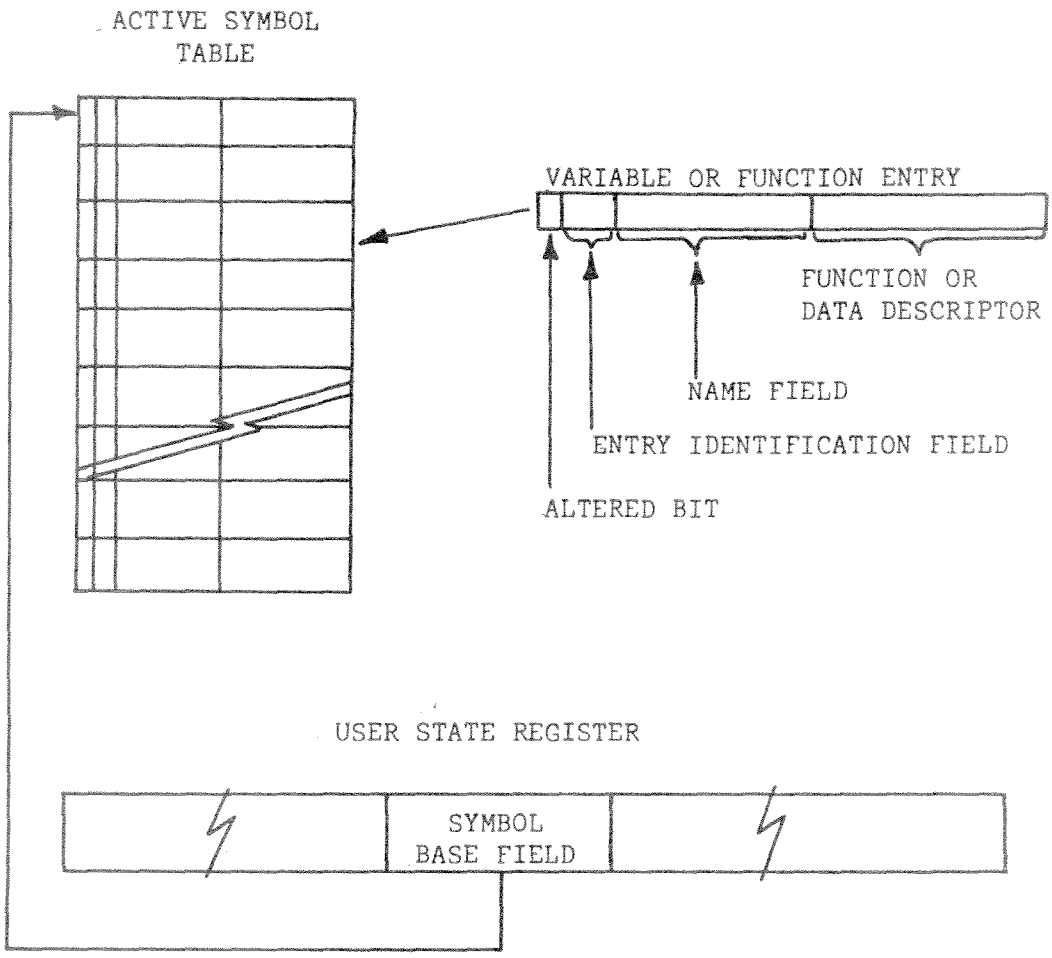
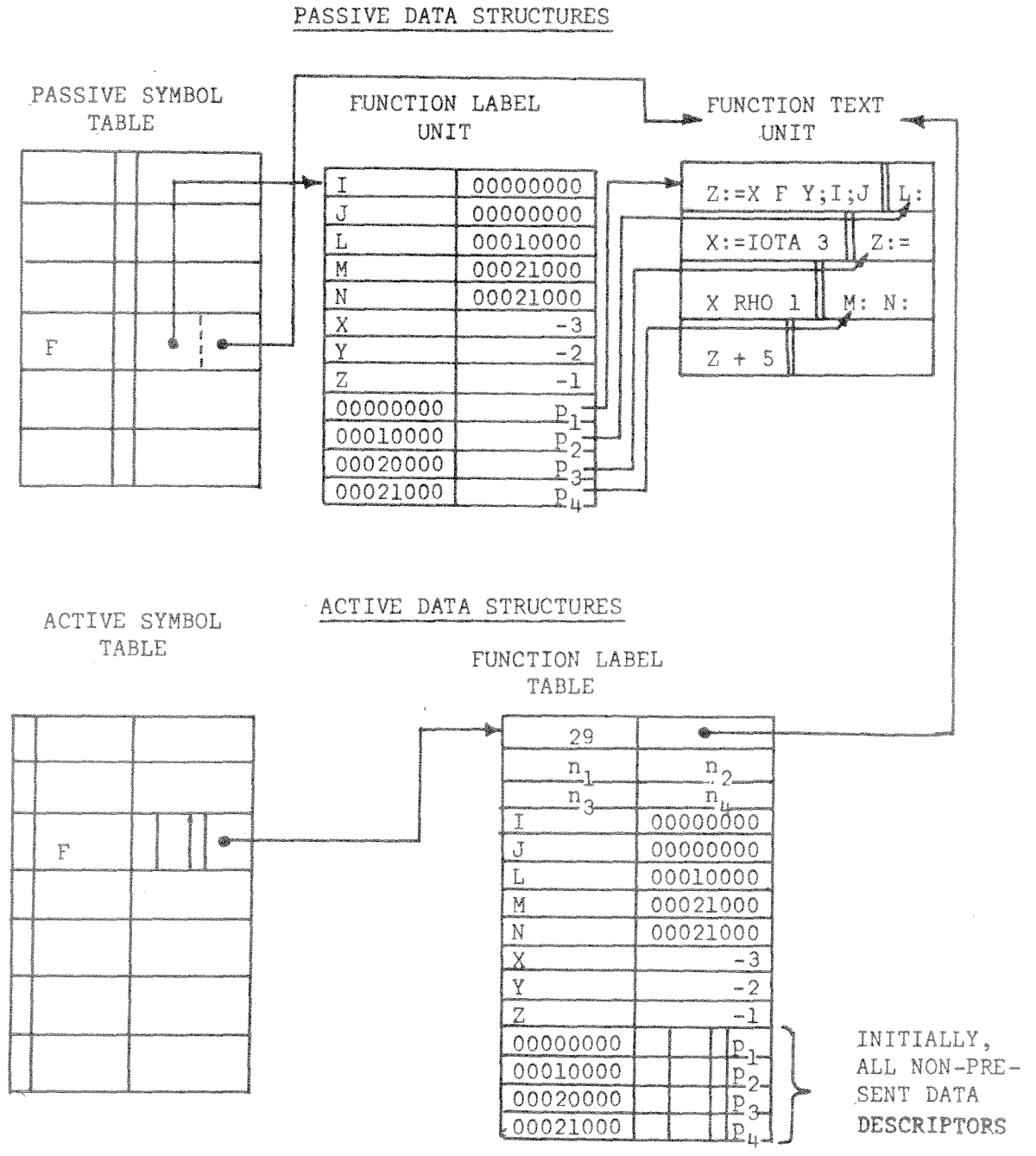


FIGURE 20

ACTIVE SYMBOL TABLE FORMAT



NOTE: n_1 is the relative location of the first numeric label, n_2 is the relative location of the first argument, n_3 is the relative location of the second argument, and n_4 is the relative location of the result.

FIGURE 21

FUNCTION LABEL TABLE STRUCTURE

passive symbol table for a particular user. In addition, all active data and functions, along with function labels and local variables are accessible through the active symbol table and function label table.

The reasons for maintaining the active and passive areas can be stated as follows: the amount of data and the number of functions in a work area may be voluminous. Further, during the execution of a calculator mode statement (which places the user in execution mode) it may happen that only a small fraction of the work area is actually referenced. Although passive data and functions are readily accessible through the virtual memory, it happens that little used areas remain on back-up storage (because of the demand paging). Active data and functions, however, are not paged out of central memory since they reside in the scratch pad. The assumption, of course, is that the accessed data has the highest probability of being accessed again before the user returns to calculator mode.

It should also be noted that lines of text in an active function are compiled only on demand. That is, there must be an attempt by the user's APL program to execute a particular line of an APL function before that line is compiled. Once the line is compiled, it remains in the scratch pad in a compiled form until the user returns to calculator mode. Again, the assumption is that function lines which have been executed have the highest probability of being re-executed. Further discussion of demand compilation is found in the sections which follow.

THE APL STATEMENT COMPILER

The APL Statement Compiler generates an internal pseudo-code string corresponding to APL statements submitted for compilation. The APL Statement Compiler can be called while the current user (as defined by the Resource Manager) is in function definition mode or in execution mode. If the user is in function definition mode, no code is generated, nor is any scratch pad memory allocated.

In execution mode, the compiler returns a present data descriptor addressing an APL numeric vector in the scratch pad. This vector has, as its first element, a present data descriptor addressing an APL character vector containing the original APL statement, as shown in Figure 22. The original statement is maintained with the compiled code in order that error messages may be printed

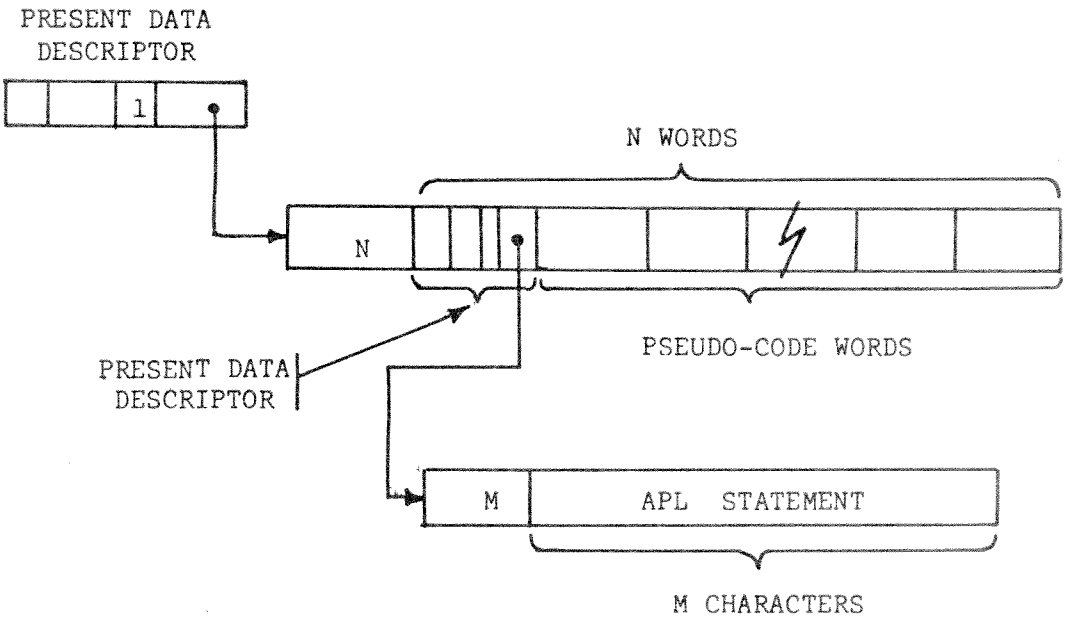


FIGURE 22

CODE STRING FORMAT

during execution. In any case, the APL Statement Compiler is called upon to compile only one statement at a time, thus maintaining functional concurrency.

APL pseudo-code words, shown in Figure 23, are interpreted by the simulated APL machine during execution of the statement. The type field of the code word indicates whether the code word represents an operand or constant fetch, or an APL operation or defined function call. If the code word represents an operand or constant then the fields are defined as follows:

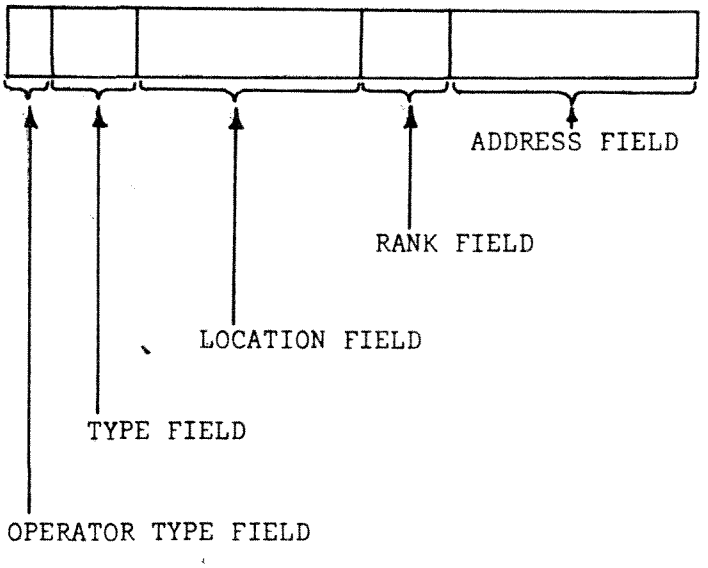
1. Operator Type. The operator type is not used for operands and constants.
2. Type Field. The type field indicates whether the code word represents an operand or a constant.
3. Location Field. The location field contains the scratch pad address of the data descriptor corresponding to the code word (i.e., the descriptor for the operand or constant).
4. Rank Field. The rank field is not used.
5. Address Field. The address field contains the location of the operand or constant within the original APL statement. This location allows exact error reporting during execution.

If the code word represents a function or operator then the fields are:

1. Operator Type. The operator type indicates the number of operands involved in the operation or function.
2. Type Field. The type field indicates whether the code word represents a function or operand.
3. Location Field. If the code word represents an APL operator then the location field contains an integer number assigned to this operator. If the code word describes a function then the location field contains the address of the descriptor for this function in the active symbol table.
4. Rank Field. The rank field contains an operator "subscript" (see the following explanation of the "[" operator) if the code word represents an operator. If the code word represents a function then the rank field indicates whether or not the function returns a value.

The final code string for an APL statement is generated in two passes:

PSEUDO-CODE WORDS



OPERATOR TYPE FIELD	TYPE FIELD	LOCATION FIELD
NILADIC	OPERAND	DESCRIPTOR
MONADIC	CONSTANT	ADDRESS
DYADIC	OPERATOR	OPERATOR
TRIADIC	FUNCTION	CODE

RANK FIELD	ADDRESS FIELD
VALUE RETURNED OR NOT RETURNED OPERATOR SUBSCRIPT	LOCATION OF LEXICAL ITEM WHICH GENERATED THE CODE WORD (USED FOR ERROR REPORTING)

FIGURE 23

PSEUDO-CODE WORD FORMAT

a forward pass, called the lexical pass, and a backward pass, called the code generation pass. The two pass approach is taken in order to arrange the code words in the proper order for a right-to-left execution of the statement.

The lexical pass involves the identification of each lexical item in the infix expression (operators, constants, variables, and functions). During this pass, shown in Figure 24, a push-down stack, called Infix, is loaded with code words corresponding to the lexical items. All table look-ups in the function label table, active symbol table, and passive symbol table occur during the lexical pass. In addition, all scalars, along with numeric and character vector constants, are placed into the scratch pad.

During the code generation pass, the Infix stack is examined and a form of suffix notation is generated by rearrangement of the code words. This form will be termed "reverse inverted polish" since it involves not only suffix form, but also a rearrangement of the operands (this form can also be thought of as direct polish written backwards). The reverse inverted polish form is suitable for execution by the simulated APL machine and is sufficient for the proper right-to-left execution. The fundamental transformations from infix to reverse inverted polish form are shown in Table 5. Note the "subscript computation" operator in Table 5, denoted by "$[_n$." This operator is "subscripted" by n; that is, the number of operands involved in the subscript computation is denoted by n.

One might think that the usual reverse polish form would be sufficient for proper APL statement execution. However, statements such as:

A + A:=5

where the variable "A" initially has a value other than five is evaluated incorrectly if reverse polish form is used.

The logic of the code generation pass is shown in Figure 25 in simplified form. There are a number of special cases not covered by the diagram in Figure 25 such as the occurrence of the quad or quote quad; however, the diagram does cover most cases. Error conditions are not shown in the state diagram for the code generator, but may be detected in a number of ways, including:

1. attempting to mark an operator or function as dyadic when it is defined as a monadic operator or function, or vice-versa;

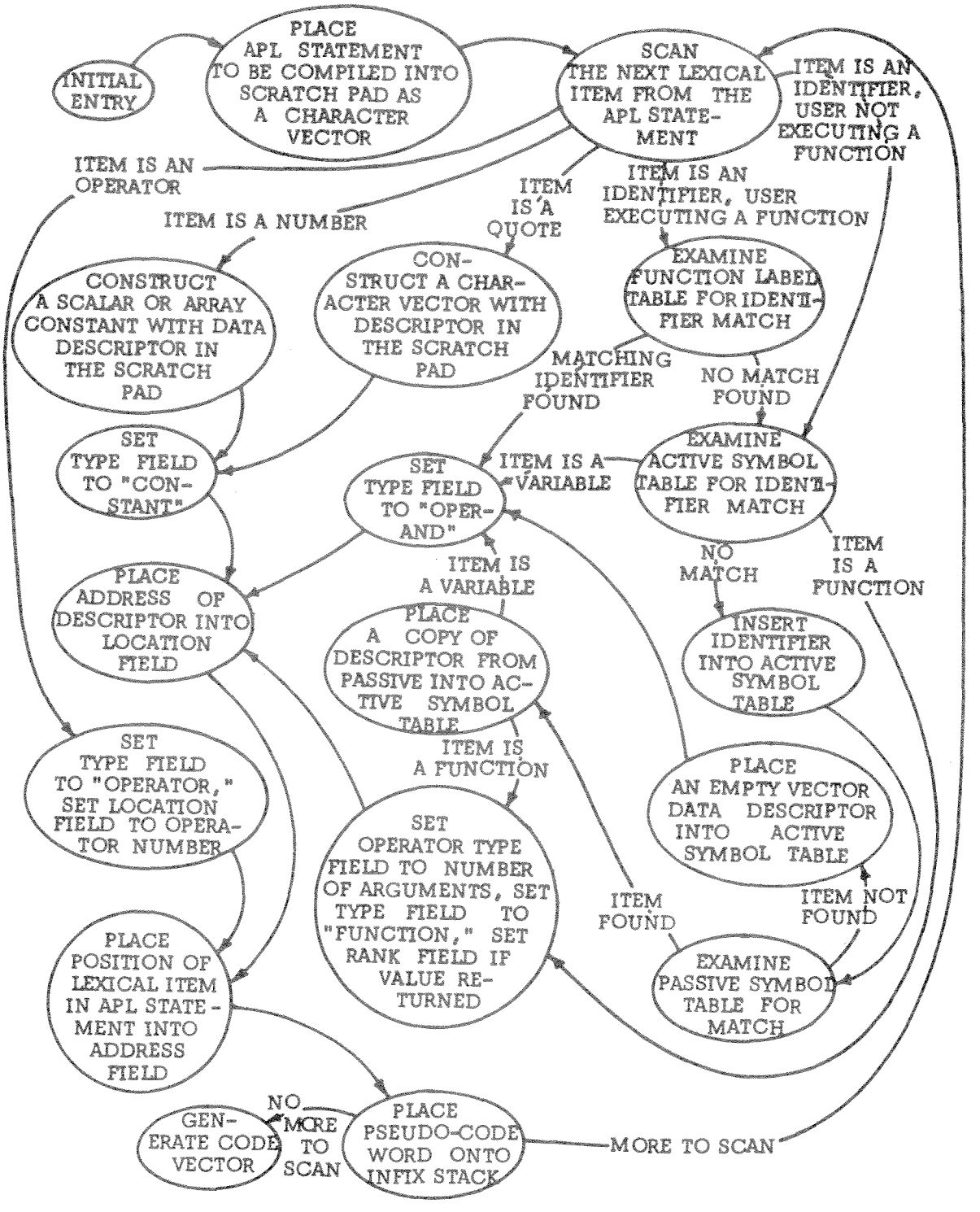


FIGURE 24

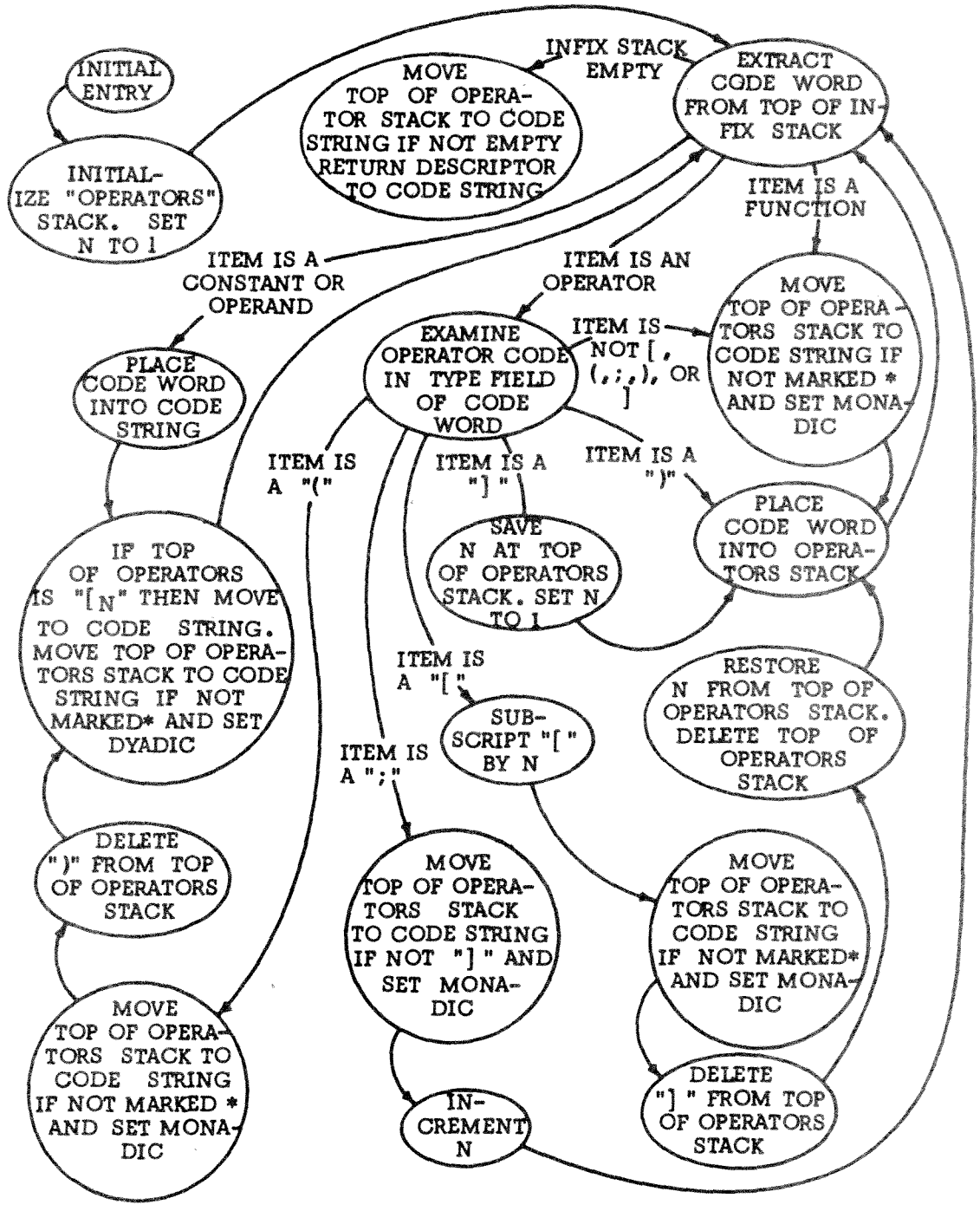
LEXICAL PASS STATE DIAGRAM

TABLE 5

INFIX TO REVERSE INVERTED POLISH TRANSFORMATIONS

INFIX	REVERSE INVERTED POLISH
m X	X m
X d Y	Y X d
M X	X M
X D Y	Y X D
$X[S_1;S_2; \ldots ;S_n]$	$S_n\ S_{n-1} \cdots S_2\ S_1\ X[_n$

NOTE: m denotes a monadic operator, d denotes a dyadic operator, M denotes a monadic function, D denotes a dyadic function, and X, Y, S_1, S_2 through S_n denote valid APL expressions.



NOTE: The Operators stack is marked if the stack is empty, or if the top element is "]" or ")."

FIGURE 25

CODE GENERATION PASS

2. the absence of an anticipated code word on the operators stack (e.g., delete")" from the top of the operators stack anticipates the occurrence of the ")");

3. examination of the operators stack for extraneous symbols at the end of the transformation.

Figure 26 shows the steps occurring in the transformation of an APL statement into reverse inverted polish form. For purposes of illustration, the symbolic equivalent of each code word in the Infix stack is used, rather than the code word itself. Note also that the stacks extend to the right for readability. Operators in the final code string which have been recognized as monadic are marked with a prime (').

Figure 27 shows the evaluation of the resulting reverse inverted polish taken from Figure 26. Circles enclose operands and operators which result in a single operand.

All constants created during the lexical pass are attached to the code string through the data descriptor returned by the statement compiler, as shown in Figure 28. Thus, the data descriptor returned by the statement compiler provides access to the constants associated with the APL statement, the APL statement itself, and the pseudo-code words corresponding to the original statement.

The connection between the APL statement compiler and the simulated APL machine is given in the following section.

THE APL "MACHINE"

The APL "Machine" component of APL\B5500 is a software simulation of a fictitious APL processor. The architecture of this "machine" is similar to that of the B5500 in that it is a stack-oriented, descriptor-based processor. The simulated machine, however, executes an order-code which is suitable for APL statement execution. Thus, the simulated machine does not directly use the B5500 hardware stack mechanism. The Machine is capable of interpreting APL statements when expressed in reverse inverted polish form. In addition, the Machine provides control functions including transfer-of-control within APL functions, and execution interruption facilities necessary for maintaining functional concurrency within the component.

APL EXPRESSION
(D IS A DYADIC FUNCTION, M IS A MONADIC FUNCTION)

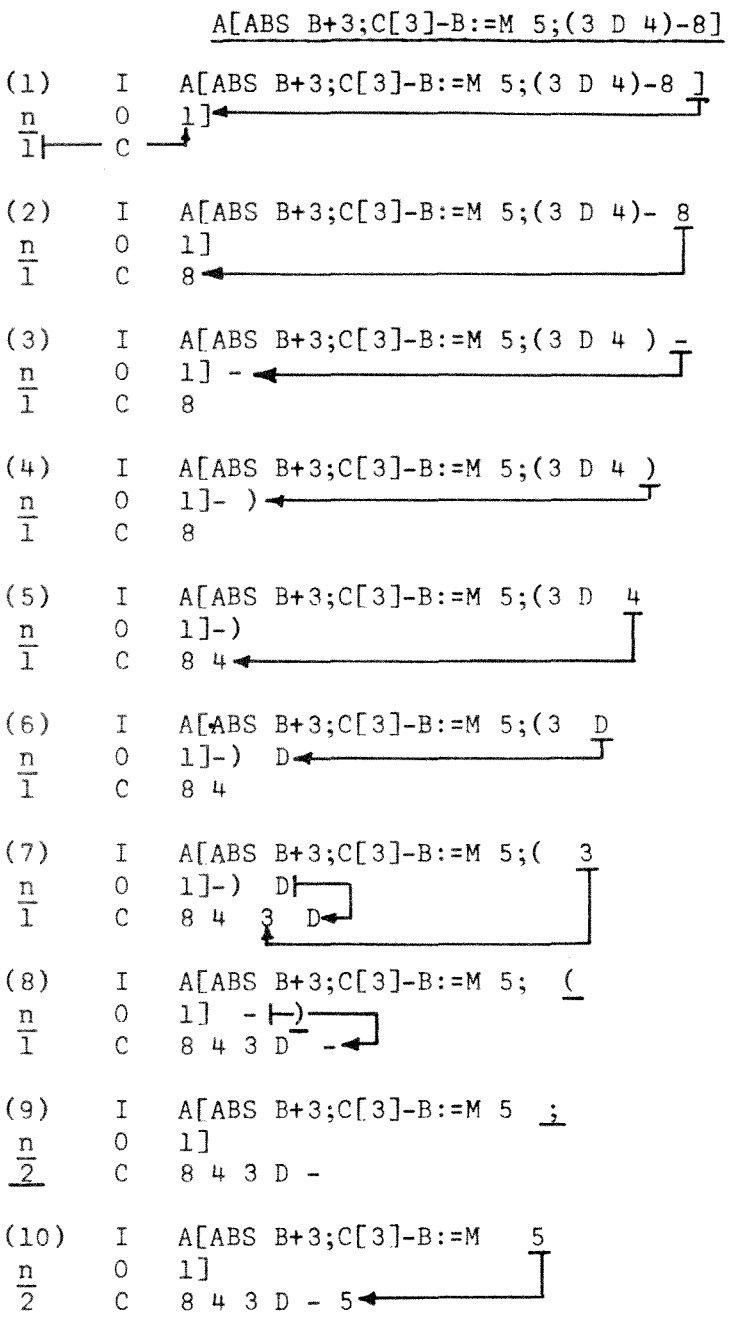


FIGURE 26

TRANSFORMATION OF AN APL STATEMENT

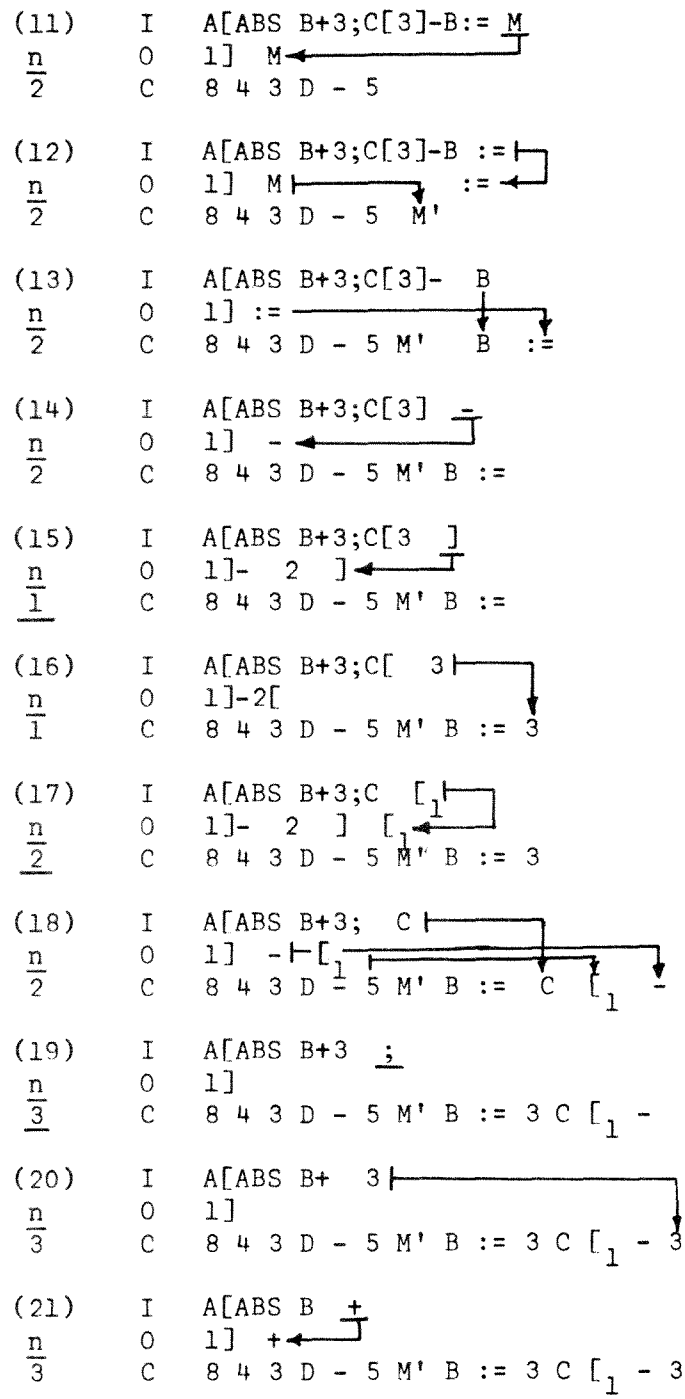
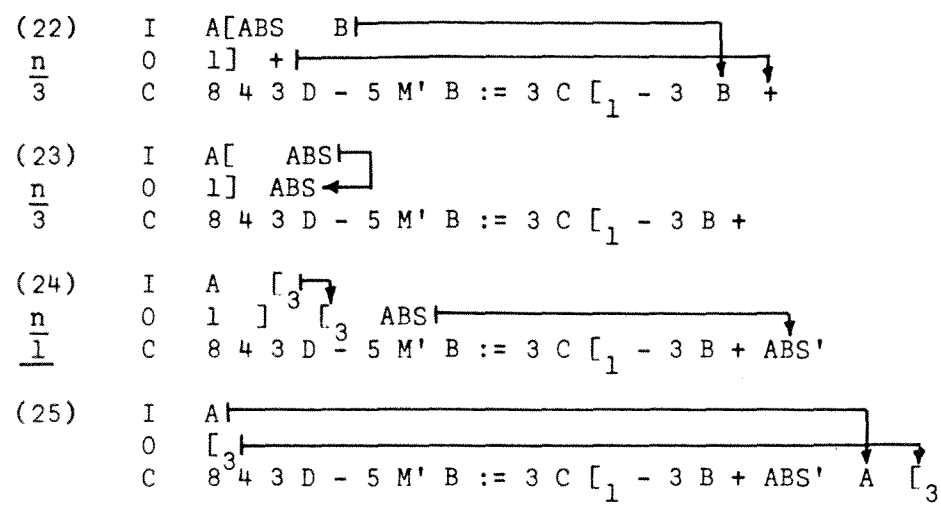


FIGURE 26

(CONTINUED)



RESULTING REVERSE INVERTED POLISH

8 4 3 D - 5 M' B := 3 C [1 - 3 B + ABS' A [3

NOTE: "I" represents the Infix stack (in symbolic form), "O" represents the operators stack, and "C" represents the code string. Each step shows the effects of one circuit through the diagram of Figure 25.

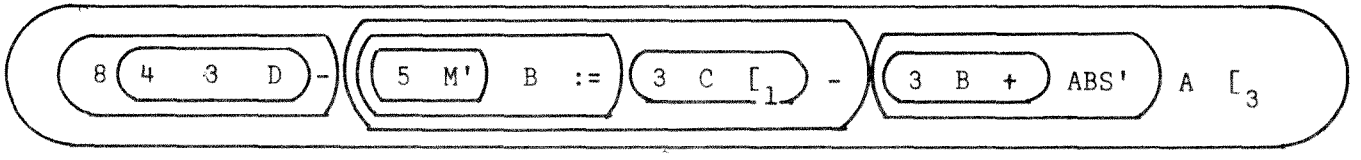


FIGURE 27

EVALUATION OF REVERSE INVERTED POLISH

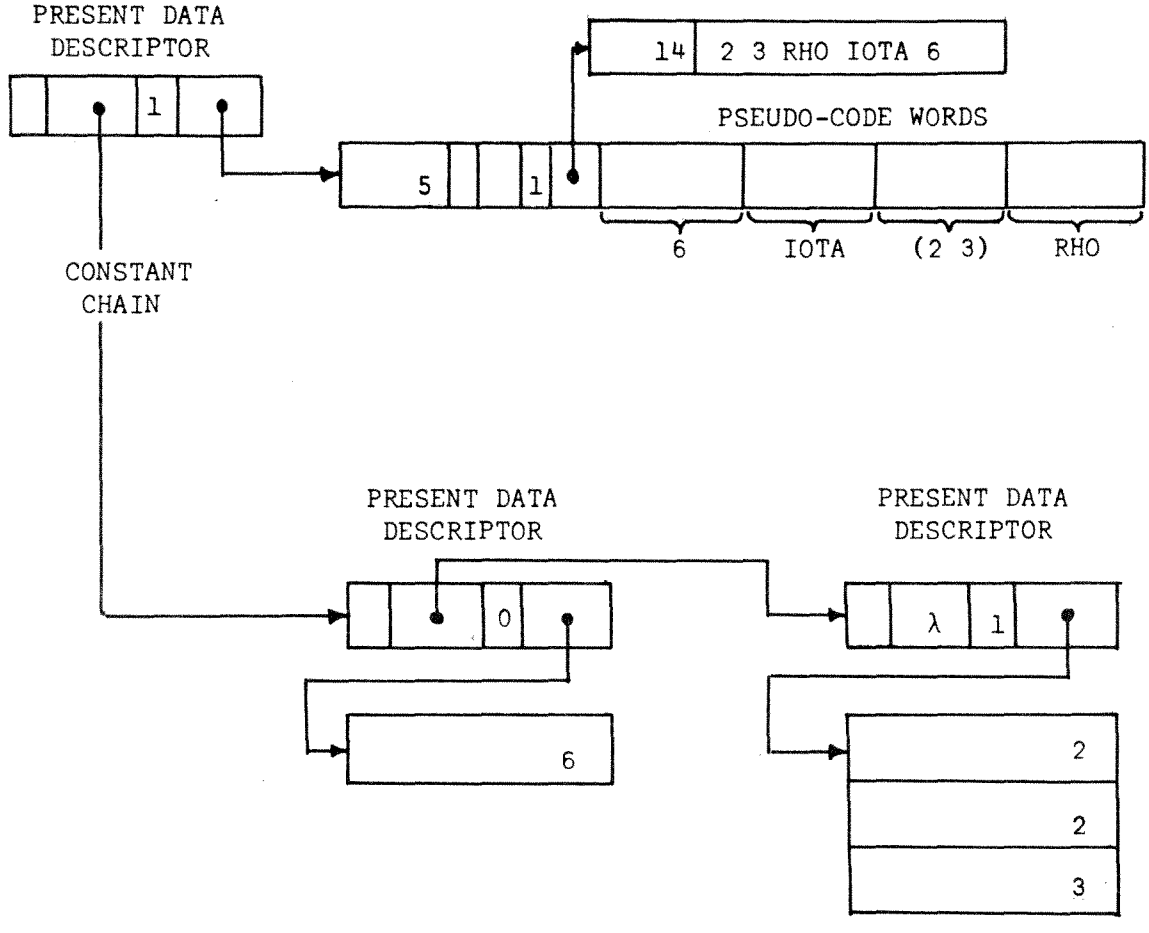


FIGURE 28

DATA STRUCTURE RESULTING FROM STATEMENT COMPILATION

Each APL user in execution mode is provided with an execution stack located in the scratch pad and addressed through the stack base field of the User State Register. The execution stack, shown in Figure 29, has the top-of-stack index as its first element. The remaining elements of the execution stack consist of either descriptors or execution control words (described below).

In addition to the execution stack, a control index (CI) is maintained which points to the current pseudo-code word being processed for the user. As the CI moves through a code string, code words representing operands cause the corresponding descriptor to be loaded onto the execution stack. Code words which represent operators or user-defined functions, however, cause the corresponding operation or function to be applied to the top descriptors. The descriptor resulting from the operation or function call replaces those descriptors involved in the operation or function call. Data descriptors with a reset named bit (temporary data) cause the corresponding data to be removed from the scratch pad when "unstacked." Figure 30 shows the steps involved in the execution of the simple APL statement of Figure 28. This method of APL statement interpretation is, of course, both natural and straightforward.

Control words mark various positions in the execution stack. The control words appear in a number of forms, as shown in Figure 31. The control words have the following functions:

1. Interrupt Mark Stack Control Word (IMS). The interrupt mark stack control word is placed at the top of the execution stack at the end of the user's execution period. Information in this control word allows later recovery of additional information for restarting execution.
2. Program Mark Stack Control Word (PMS). The program mark stack control word contains information leading to the code string in execution directly above the control word in the stack.
3. Function Mark Stack Control Word (FMS). The function mark stack control word is inserted into the execution stack whenever defined functions are invoked during execution.
4. Quad Input and Quote Quad Input Mark Stack Control Words (QMS and QQMS). The quad input and quote quad input mark stack control words are

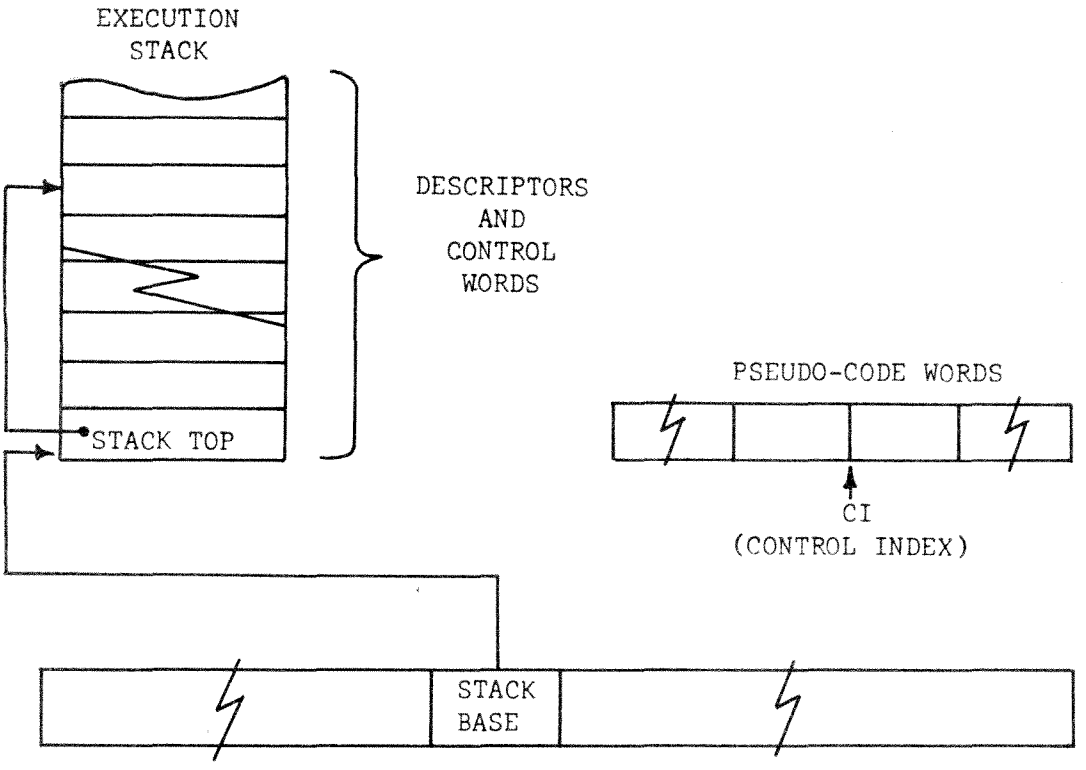


FIGURE 29

EXECUTION STACK AND CONTROL INDEX

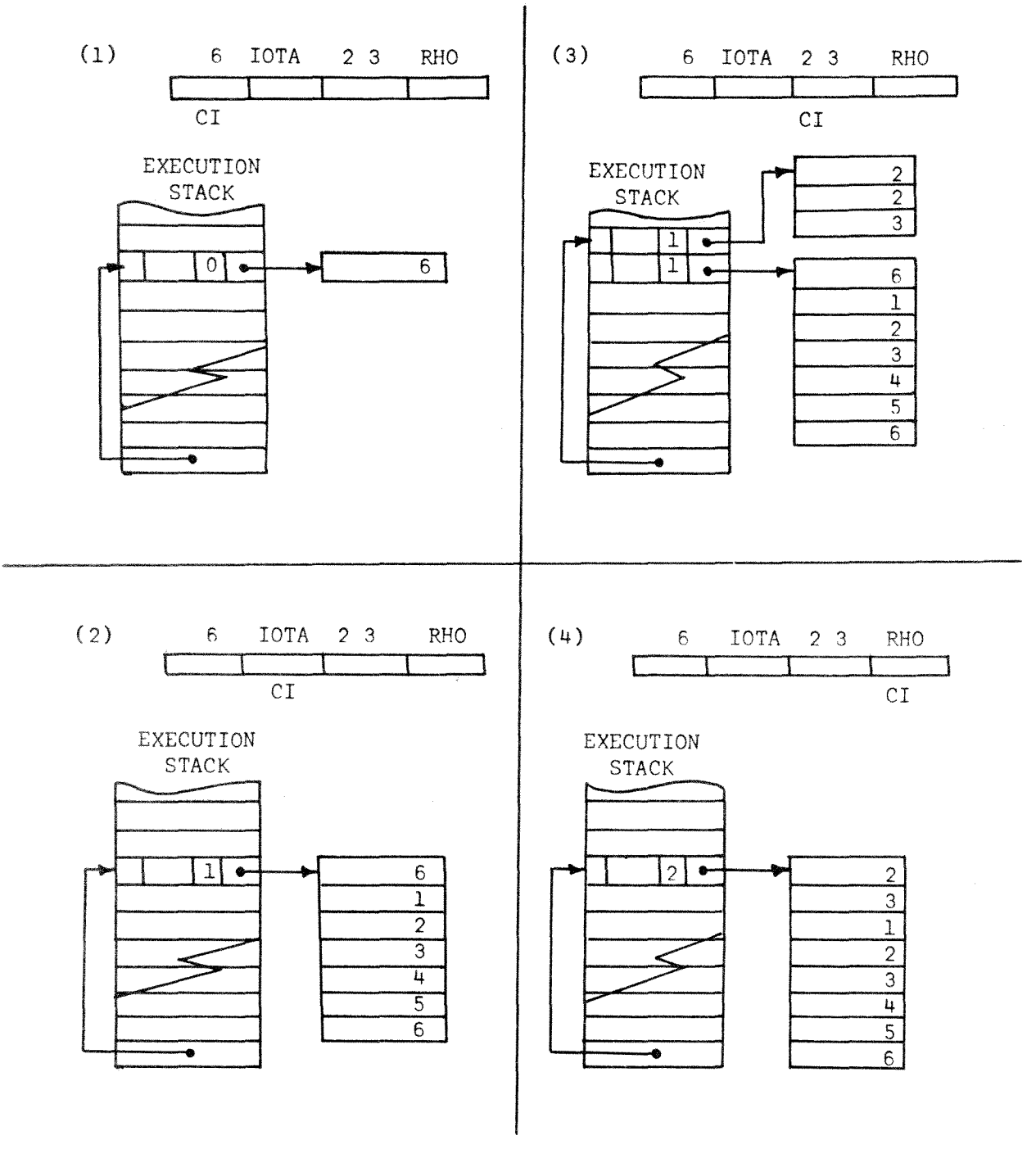


FIGURE 30

INTERPRETATION OF APL CODE STRINGS

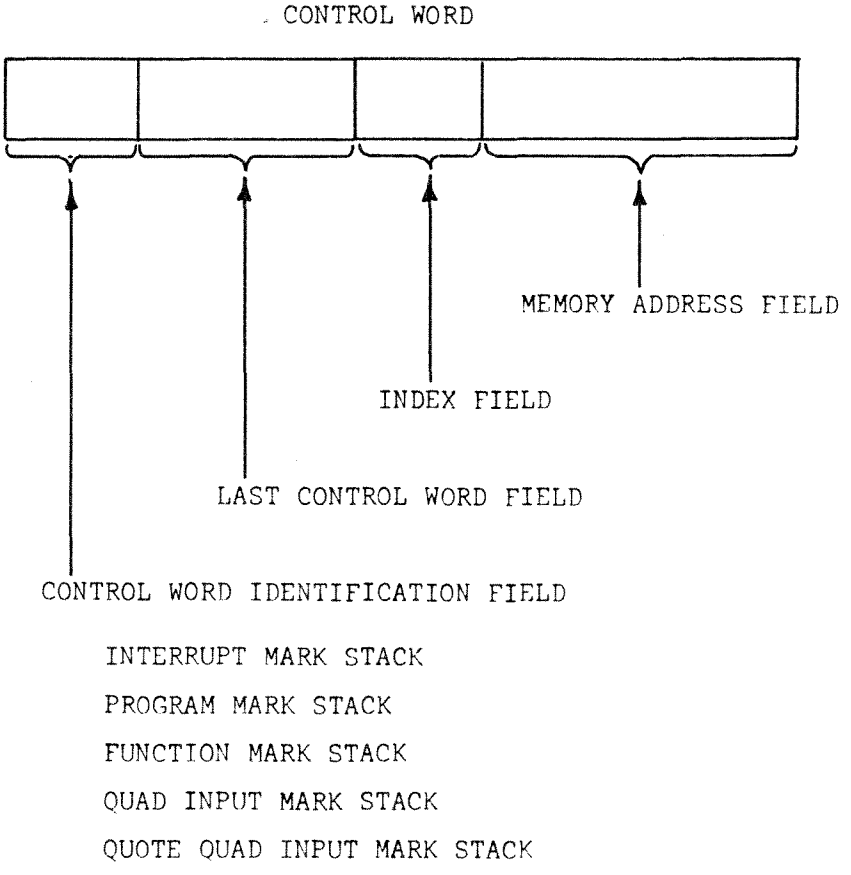


FIGURE 31

CONTROL WORD FORMAT

inserted into the execution stack whenever the user's APL program requests quad or quote quad input from the terminal.

All control words are linked together in the execution stack through the "last control word" field of each control word. The use of the "index" field and the "memory address" field depends on the type of control word.

The execution stack is initialized with a program mark stack upon entry to execution mode from calculator mode. The program mark stack addresses the data descriptor corresponding to the compiled calculator mode statement, as shown in Figure 32a.

An interrupt mark stack control word is inserted at the top of the execution stack whenever time-slice interruption of execution occurs, as shown in Figure 32b. At the time of the interruption, the control index (CI) is placed into the index field of the program mark stack.

A number of actions take place in the case that a calculator mode statement invokes a function, or a function invokes another function. The arguments to a function are at the top of the execution stack at the time of the call because of the form of the reverse inverted polish. The function descriptor is examined in the active symbol table (addressed directly by the pseudocode word) and, if not present, the function label table is constructed as shown in Figure 21. The function mark stack control word is inserted into the execution stack, followed by the descriptors for each argument and local variable. In order to obtain call-by-value parameters, all data described by data descriptors with a set named bit cause a copy operation on the data items before passing the new descriptor to the function.

The descriptors corresponding to local variables, including labels, are kept in the stack area above the function mark stack. Local variables which do not correspond to formal parameters are initially set to "null" by placing a null vector (rank field zero) data descriptor into the stack. Labels are treated as any other local variable except that the descriptor in the stack is initially a present scalar data descriptor addressing a numeric scalar corresponding to the line on which the label appears.

The simulated APL Machine also keeps track of the current line being executed by a user when the user is executing a function. The current line

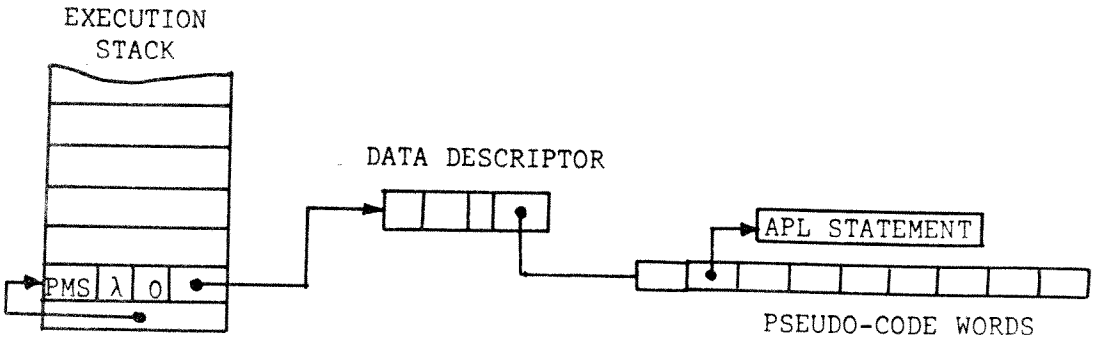


FIGURE 32A

INITIAL EXECUTION STACK CONTENTS

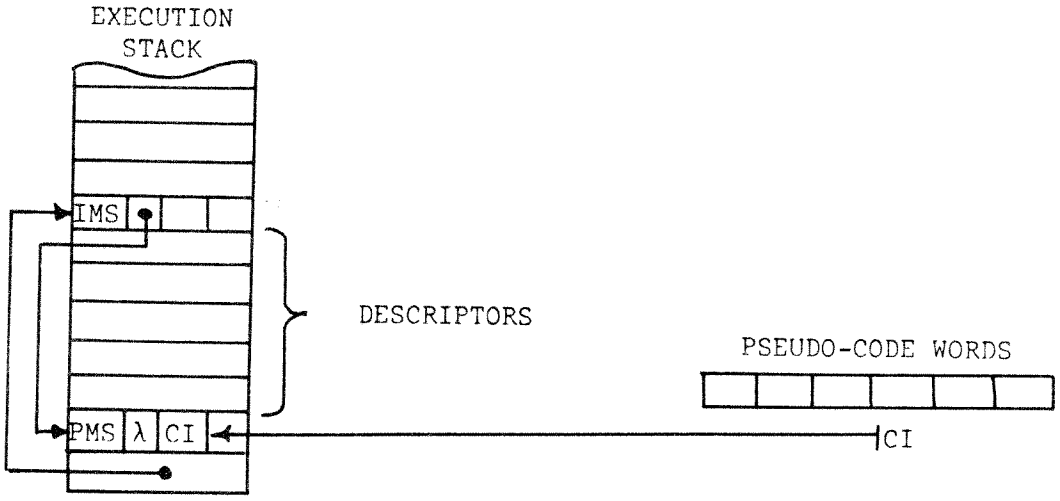


FIGURE 32B

EXECUTION STACK AFTER INTERRUPTION

is called the line index (LI), and is essentially an index into the corresponding function label table. A program mark stack control word is inserted into the execution stack after the parameters and local variables in order to start the function, as shown in Figure 33.

Two points should be made about function execution. First, because of the index field in each of the control words, functions may be invoked at any point in the execution of a function. The CI is saved in the previous program mark stack, and the LI is saved in the previous function mark stack (if it exists). Upon return from the function execution, the CI and LI can be recovered, and control is returned to the pseudo-code word which follows the function call. Secondly, since the descriptors for parameters, local variables, and labels are maintained in the execution stack, and since the pseudo-code strings are "pure" (i.e., they are not self-modifying), recursive function invocation is permitted.

At the end of function execution, the function mark stack is deleted. If the function returns a value, the descriptor representing the value is placed at the top of the execution stack. The LI and CI are then recovered from the control words which are "lower" in the execution stack.

Note also that the data descriptors in the function label table are initially marked non-present (refer to Figure 21). Any reference to a non-present data descriptor causes the APL Machine to make the data present (i.e., in the case of data, the "data" sequential storage unit is referenced with the corresponding data brought into the scratch pad). In the case of data descriptors in the function label table, the corresponding APL statement is retrieved from the function text unit and the APL Statement Compiler is called to compile the line. The resulting data descriptor replaces the previously non-present data descriptor in the function label table. The compiled form of the statement then remains in the scratch pad until the user returns to calculator mode.

This "demand compilation" avoids unnecessary compilation of statements which are never executed. In addition, functional concurrency is more easily attained since the compilation is incremental.

When the user returns to calculator mode from execution, the Resource

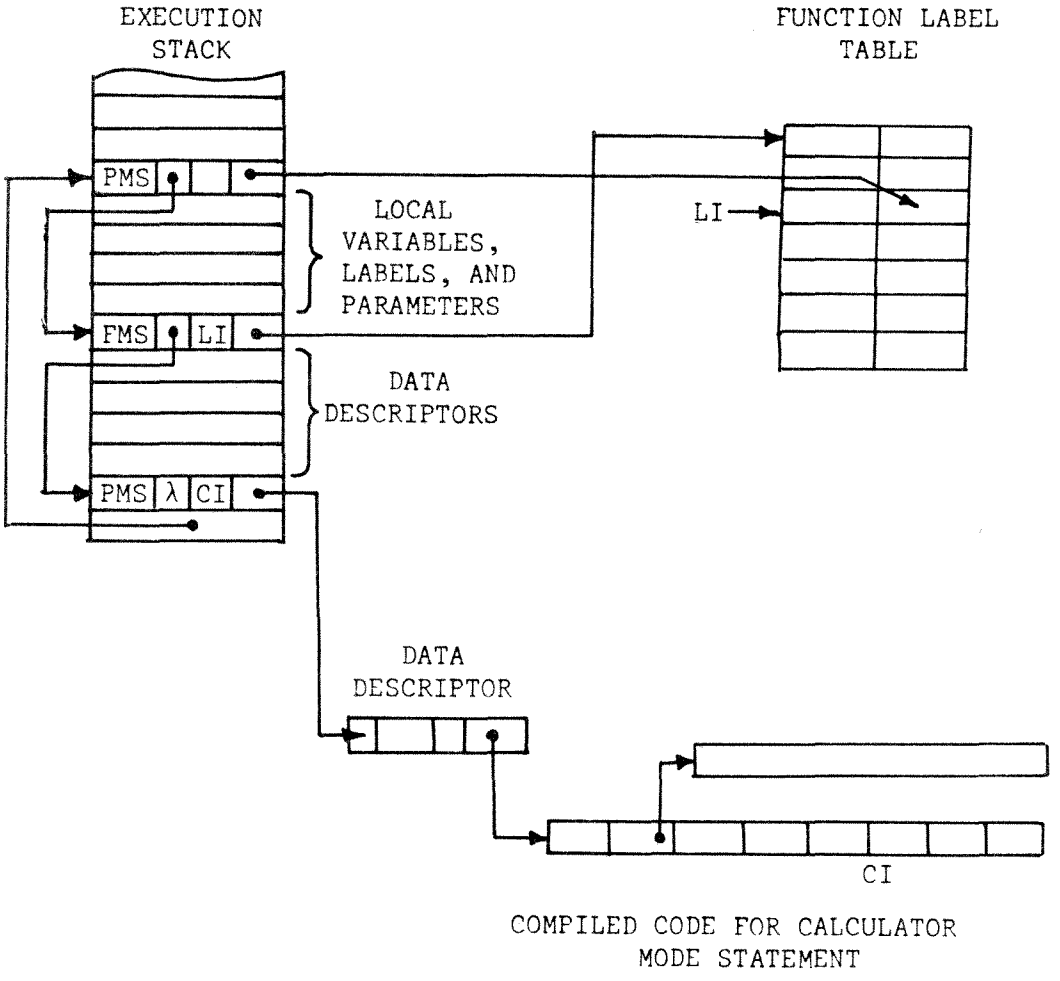


FIGURE 33

STACK ORGANIZATION FOR FUNCTION EXECUTION

Manager calls upon the APL Machine to make active data into passive data. The active symbol table is examined for variables which have the altered bit set. Entries are then made into the "names" and "data" storage units for these variables. Thus, the passive data retains its original form until the completion of execution. Passive data is not altered if an error is encountered during the execution of the APL program unless the STORE monitor command is issued by the user.

If an execution error is encountered, the user is notified and the execution is suspended. During suspension, the user may examine the active symbol table, the stack locations corresponding to the local variables of the most-recently executing function, and the local variables of any other suspended functions. The user may alter these variables and continue function execution, or abort the execution. If the function is aborted before a STORE command is issued, then the active symbol table values are destroyed and the passive symbol table values are retained. Thus, the function can be restarted without re-initialization of global quantities.

Additional functions of the simulated APL Machine include:

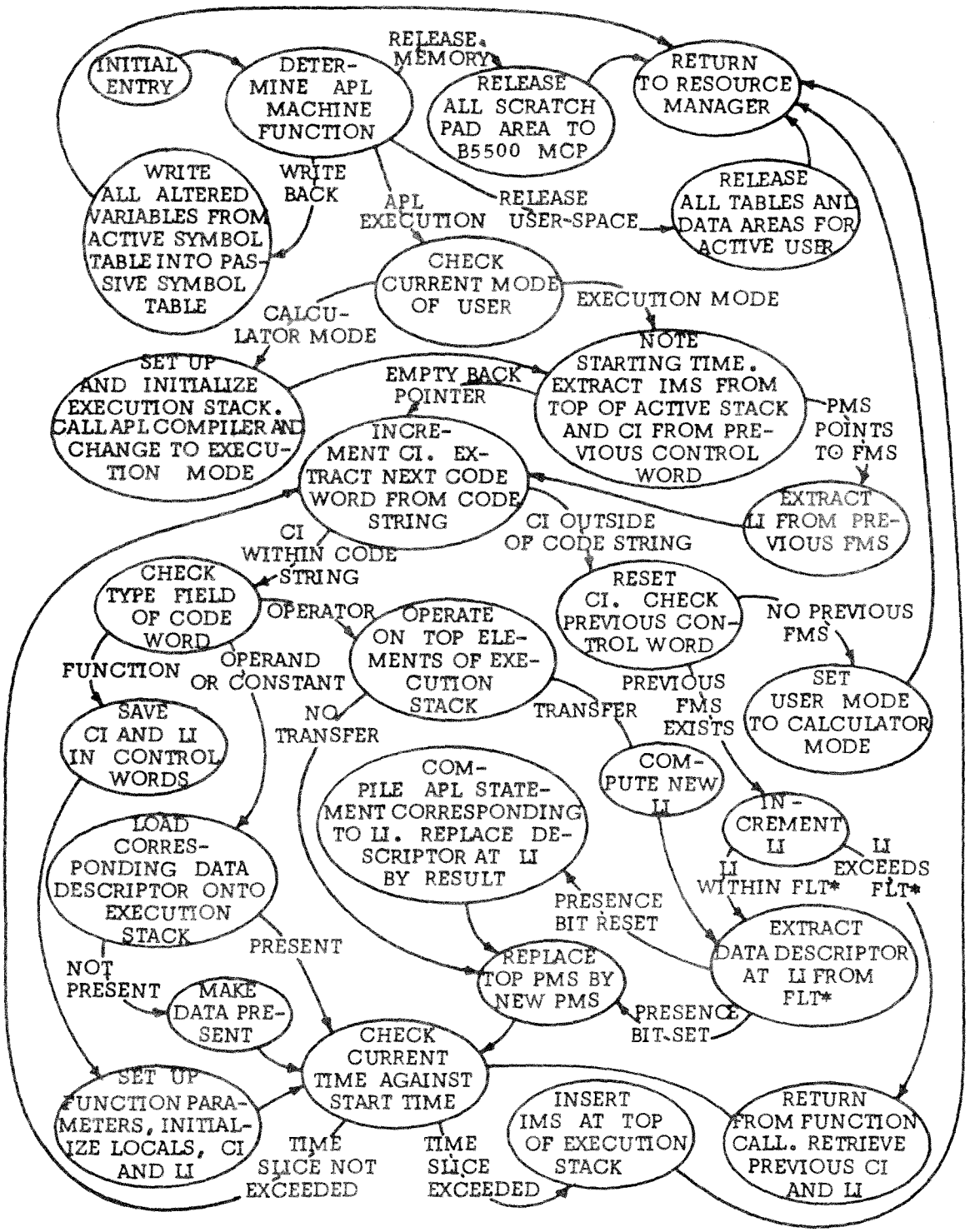
1. deallocation of all scratch pad memory cells (returning the storage areas to the B5500 MCP) when no users are in execution mode, and
2. deallocation of areas reserved for a particular user returning to calculator mode from execution mode.

Although the above discussion is a simplification of the functions of the simulated APL Machine, it does provide an outline of the operations and data structures involved. The state diagram given in Figure 34 shows the logic of the APL Machine.

A detailed discussion of efficient APL "machine" organization and data representation, along with an extensive bibliography concerning APL-related topics, is given by Abrams[10].

CONCLUSION

The APL\B5500 system is a self-contained time-sharing submonitor for the Burroughs B5500 computer providing full APL\360 processing capabilities. Although the design of APL\B5500 was affected by limited computer resources,



NOTE: FLT denotes the function label table.

FIGURE 34

APL MACHINE LOGIC

such as central memory, the overall design is thought to be sufficiently general to be applicable to other APL implementations.

The APL\B5500 system is presently in a stable condition: no major modifications in design are foreseen. It is necessary, however, to measure the effectiveness of the various APL components in an attempt to make minor modifications and adjustments to tune the system for best performance.

REFERENCES

1. Iverson, K. E., and Falkoff, A. D. APL\360: User's Manual. International Business Machines Corporation, 1968.

2. Iverson, K. E. A Programming Language. Wiley, New York, London, 1962.

3. Kildall, G., Smith, L., Swedine, S., and Zosel, M. Preliminary APL\B5500 Manual. University of Washington Computer Center, 1970.

4. B5500 Information Processing Systems Reference Manual. Burroughs Corporation, Detroit, Michigan.

5. A Narrative Description of the Burroughs B5500 Disk File Master Control Program. Burroughs Corporation, Detroit, Michigan.

6. Stimler, S. Some criteria for time-sharing system performance. Comm. ACM, 12, 1 (January 1969), 41-53.

7. Kildall, G. Experiments in large scale computer direct access storage manipulation. Tech. Rep. No. 69-1-01, Computer Science Group, University of Washington, Seattle, Washington, January 1969.

8. Kuehner, C., and Randell, B. Demand paging in perspective. AFIPS conference Proceedings, 33, Part 2, 1968, 1011-1018.

9. Randell, B. A note on storage fragmentation and program segmentation. Comm. ACM, 12, 7 (July 1969), 365-372.

10. Abrams, P. An APL machine. SLAC Report No. 114, Stanford Linear Accelerator Center, Stanford University, Stanford, Ca., February, 1970.

APPENDIX A - SAMPLE TERMINAL SESSION

```
XXXXXXXXXXXX
 MARY LOGGED IN THURSDAY 10-22-70  09:27
      )VARS←
 INTERP (F) NEWTON (F) STRING  X        X0
      )FNS←
 INTERP  NEWTON
      )ERASE STRING←
      )VARS←
 INTERP (F) NEWTON (F) X        X0
      2+2←
4
      2-2←
0
      -2←
-2
      #2←
-2
      2 #2←
2  -2

      2#2←
2  -2

      2&3+4←
14
      (2&3)+4←
10
      )"")"-"4←
(2&3)-4
      2
      3.4 MAX 4.5←
4.5
      )DIGITS←
3
      )DIGITS 9←
      4 & 3 MAX 5.1←
20.4
      (4&3) MAX 5.1←
12
      CIRCLE 1←
3.141592654
      CIRCLE 1 2←
3.141592654  6.283185307

      1 CIRCLE 1←
0.841470985
      IOTA 4←
1  2  3  4

      CIRCLE IOTA 2←
3.141592654  6.283185307

      SG:=M GCD N←
[1]   G:=M←
[2]   M:=M RESD N←
[3]   =:(M NEQ 0)/XIT←
[4]   [3[""]]/"CON"T←
.4]   [3[]]←

[3]   =:(M NEQ 0)/CONT

[4]   N:=G←
```

```
[5]    [4[']]"CONT:"←
[5]    =:1←
[6]    [[]]←

       G:=M GCD N
[1]    G:=M
[2]    M:=M RESD N
[3]    =:(M NEQ Ø)/CONT
[4]    CONT:N:=G
[5]    =:1

[6]    [CONT[]]←

[4]    CONT:N:=G

[6]    [2[]4]←

[2]    M:=M RESD N
[3]    =:(M NEQ Ø)/CONT
[4]    CONT:N:=G

[6]    [CONT-2[]CONT+1]←

[2]    M:=M RESD N
[3]    =:(M NEQ Ø)/CONT
[4]    CONT:N:=G
[5]    =:1

[6]    $←
       2 GCD 2←


       )SI←
       GCD      S
       )SI V←
       GCD      S  CONT     G         M         N
       CONT←
4
       G,M,N←
Ø  2   2

       =:Ø←
       $GCD←
[6]    [3.1]=:Ø←
[3.2]  [[]]←

       G:=M GCD N
[1]    G:=M
[2]    M:=M RESD N
[3]    =:(M NEQ Ø)/CONT
[3.1]  =:Ø
[4]    CONT:N:=G
[5]    =:1

[3.2]  $←
       2 GCD 2←
2
       36 GCD 64←
4
       $GCD←
[6]    [3][3.1]←
[6]    [[]]←

       G:=M GCD N
[1]    G:=M
[2]    M:=M RESD N
```

```
[4]   CONT:N:=G
[5]   =:1

[6]   [CONT["]]""N←
[6]   [4[]]←

[4]   NT:N:=G

[6]   [4["]]""N:←
[6]   [3]=:L& M NEQ 0←
[4]   [4["]]"L:"←
[4]   [4[]]←

[4]   L:N:=G

[4]   [[]]$←

      G:=M GCD N
[1]   G:=M
[2]   M:=M RESD N
[3]   =:L& M NEQ 0
[4]   L:N:=G
[5]   =:1

      36 GCD 64←
4



      $Z:=FIB N←
[1]   =:N+2 MIN 4←
[2]   =:Z:=0←
[3]   =:1-Z:=L<1←
[4]   =:Z:=(FIB N-1)+FIB N-2←
[5]   [[]]$←

      Z:=FIB N
[1]   =:N+2 MIN 4
[2]   =:Z:=0
[3]   =:1-Z:=1
[4]   =:Z:=(FIB N-1)+FIB N-2

      FIB 0←
0
      FIB 1←
1
      FIB 2←


      )SI←
      FIB      S
      )SIV←
      FIB      S  N        Z
      N←
2
      Z←
1
      )ABORT←
      $FIB[4["]]""Z$←
[5]   [4[]]←

[4]

[5]   [4]=:Z:=(FIB N-1)+FIB N-2←
[5]   [4["]]""Z←
```

```
[5]     [4[]]←
[4]     Z:=(FIB N-1)+FIB N-2

[5]     S←
        FIB 2←
1
        FIB 4←
        FIB 5←
        )SIV←
 NULL.
        $FIB[[]]$←

        Z:=FIB N
[1]     =:N+2 MIN 4
[2]     =:Z:=0
[3]     =:1-Z:=1
[4]     Z:=(FIB N-1)+FIB N-2

        $FIB[1["]]:"(N+2)" M←
[5]     [1[]]←

[1]     =:(N+2) MIN 4

[5]     S←
        FIB 2←
1
        FIB 4←
3
        FIB 6←
8
        FIB 8←
21
        $INTERP[[]]$←

        INTERP;X;Y;Z;D;N
[1]     "INTERPOLATION PROBLEM C1"
[2]     =:(0=&/(IOTA N:=RHO X)=X IOTA X:=[])/UNIQERR,0 RHO []:="INPUT X VA
LUES"
[3]     =:(N NEQ RHO Y:=[])/DIMERR,0 RHO []:="INPUT Y VALUES"
[3.5]   =:(N GEQ D:=X IOTA Z:=[])/FOUNDZ,0 RHO []:="INPUT VALUE TO INTERP
OLATE"
[4]      =: 0,0 RHO []:="INTERPOLATED VALUE IS";+/(Y&(&/D)%D:=Z-X)%&/(N,N-
1)RHO((N*2)RHO 0,N RHO 1)/,X CIRCLE . -X
[5]     FOUNDZ: =:0,0 RHO []:="INTERPOLATED VALUE IS";Y[D]
[6]     UNIQERR: =:0,0 RHO []:="X VALUES NOT UNIQUE  ERROR"
[7]     DIMERR: "DIMENSIONS DO NOT MATCH   ERROR"

        $INTERP[IOTA]←
[9]     [FOUNDZ-1[]FOUNDZ+1]$←

[5]      =: 0,0 RHO []:="INTERPOLATED VALUE IS";+/(Y&(&/D)%D:=Z-X)%&/(N,N-
1)RHO((N*2)RHO 0,N RHO 1)/,X CIRCLE . -X
[6]     FOUNDZ: =:0,0 RHO []:="INTERPOLATED VALUE IS";Y[D]
[7]     UNIQERR: =:0,0 RHO []:="X VALUES NOT UNIQUE  ERROR"

        INTERP←
INTERPOLATION PROBLEM C1

INPUT X VALUES

[]:
        1 4 6 10←
INPUT Y VALUES

[]:
```

```
      3 8 12 40←
INPUT VALUE TO INTERPOLATE

[]:
      5←
INTERPOLATED VALUE IS 9.592592593

      $X:=FX NEWTON DFX; ERR←
LABEL ERROR AT  X:=FX NEWT

      )FNS←
 FIB     GCD     INTERP  NEWTON
      $NEWTON[[]]$←

      X:=FX NEWTON DFX;ERR
[1]   X:=X0
[2]   =:((ABS ERR) GEQ @-6)/2, 0 RHO X:=X-ERR:=EPS FX, "%", DFX

      "((X*2)-2)" NEWTON "2&X"←
1.414213562
      "((X*2-2" NEWTON "2&X"←
SYNTAX ERROR AT  (X*2-2%2&X

NEWTON
[2]   SYNTAX ERROR AT  EPS FX, "%

      )"-2")"←
"((X*2-2)" NEWTON "2&X"
      SYNTAX ERROR AT  (X*2-2)%2&

NEWTON
[2]   SYNTAX ERROR AT  EPS FX, "%

      )SIV←
      NEWTON  S  DFX     ERR     FX      X
      NEWTON  S  DFX     ERR     FX      X
      =:0←
1
      )ABORT←
      )SIV←
 NULL.
      )"2")←
=:0
      SYNTAX ERROR AT  0

      "WALLA WALLA WASH"←
WALLA WALLA WASH

      (" " NEQ STRING)/STRING:=[]←
[]:
      WALLA WALLA WASH←
SYNTAX ERROR AT  WASH
SYNTAX ERROR

      (" " NEQ STRING)/STRING:=    []←
[]:
      "WALLA WALLA WASH"←
WALLAWALLAWASH

      STR:=["]←
A FAT CAT←
      STR←
A FAT CAT

      STRING[ 2 10 RHO 6 + IOTA 10]←
WALLA WASH
```

```
WALLA WASH

      2 10 RHO 6 DROP STRING←
WALLA WASH
WALLA WASH


      )WIDTH 30←
      SNEWTON[[]]$←

      X:=FX NEWTON DFX;ERR
[1]   X:=X0
[2]   =:((ABS ERR) GEQ @-6)/2,
 0 RHO X:=X-ERR:=EPS FX, "%",
DFX

      )DIGITS←
9
      1%30←
SYNTAX ERROR

      1%30←
0.033333333
      )DIGITS 3←
      1%30←
0.033
      )WIDTH 72←
      13 RNDM 52←
49  43  27  14  26  21  44  9  16  29  6  30  32


      )OFF
      END OF RUN
```

APPENDIX B - SYNTAX

<apl program> ::=)<login>←<statement set>←)<logout>←

<login> ::= <user code>

<user code> ::= <identifier>

<logout> ::= ØFF<off option>

<off option> ::= DISCARD|<empty>

<statement set> ::= <statement>|<statement set>←<statement>

<statement> ::= <monitor command>|<apl statement>|<empty>

<monitor command> ::=)<command>

<command> ::= <library maintenance>|CLEAR|ERASE<identifier list>|FNS|
VARS|SI|SIV|ABØRT|STØRE|<buffer edit>|<run parameter>|
LØGGED|<message>

<library maintenance> ::= LØAD<library name>|<copy>|<clear>|<save>

<library name> ::= <library prefix><library suffix>

<library prefix> ::= <job number>,|<empty>

<job number> ::= {user account number}

<library suffix> ::= <identifier>

<copy> ::= CØPY<library name><copy name>

<copy name> ::= <stored program name>|<variable name>

<stored program name> ::= <identifier>

<variable name> ::= <identifier>

<clear> ::= CLEAR<library suffix>

<save> ::= SAVE<library suffix><lock option>

<lock option> ::= LØCK|<empty>

<identifier list> ::= <identifier>|<identifier list><space><identifier>

<buffer edit> ::= "<line edit>

<line edit> ::= <search string>"<insert string><quote option>

<search string> ::= <proper string>|<empty>

<insert string> ::= <proper string>|<empty>

<quote option> ::= "<search string>|<empty>

<run parameter> ::= <parameter type><number>|SYN|NØSYN|<parameter type>

<parameter type> ::= ØRIGIN|WIDTH|DIGITS|SEED|FUZZ

<message> ::= MSG<station><improper string>

<station> ::= <unsigned integer>

<improper string> ::= <improper string element>|<improper string>
<improper string element>

APPENDIX B (Continued)

<improper string element> ::= <visible string character>|"|<space>
<apl statement> ::= <stored program definition>|<basic statement>
<stored program definition> ::= $<definition entry><stored program body>$
<definition entry> ::= <stored program name>|<header>
<header> ::= <stored program options><local variables>←
<stored program options> ::= <function specifier><parameter options>
<function specifier> ::= <variable name> := |<empty>
<parameter options> ::= <niladic name>|<monadic name><formal parameter>|<formal parameter><dyadic name><formal parameter>
<niladic name> ::= <niladic subroutine name>|<niladic function name>
<dyadic name> ::= <dyadic subroutine name>|<dyadic function name>
<monadic name> ::= <monadic subroutine name>|<monadic function name>
<niladic subroutine name> ::= <identifier>
<niladic function name> ::= <identifier>
<dyadic subroutine name> ::= <identifier>
<dyadic function name> ::= <identifier>
<monadic subroutine name> ::= <identifier>
<monadic function name> ::= <identifier>
<formal parameter> ::= <identifier>
<local variables> ::= <local set>|<empty>
<local set> ::=; <identifier>|<local set>;<identifier>
<stored program body> ::= <stored program statement>|<stored program body>←<stored program statement>
<stored program statement> ::= <edit>|<compound statement>|<empty>
<edit> ::= [<edit command>
<edit command> ::= <display>|<insertion>|<change>|<delete>
<display> ::= <line option>[]<line option>]
<line option> ::= <line reference>|<empty>
<line reference> ::= <label expression>|<number>
<label expression> ::= <identifier><relative location>
<relative location> ::= <direction><number>|<empty>

APPENDIX B (Continued)

```
<direction> ::= +|-
<insertion> ::= <line reference>]<compound statement>
<change> ::= <line option>["]<line option>]<line edit>
<delete> ::= <line reference>]<delete option>
<delete option> ::= [<line reference>]|<empty>
<compound statement> ::= <label set><basic statement>
<label set> ::= <label>|<label set><label>|<empty>
<label> ::= <identifier>:
<basic statement> ::= <expression>|<subroutine call>|<transfer
                      statement>
<expression> ::= <operand>|<assignment statement>|<left part>
                 <expression>
<operand> ::= <constant>|<identifier><subscript option>|
              (<expression>)|[]|["]|<niladic function name>
<subscript option> ::= [<subscript list>]|<empty>
<subscript list> ::= <subscript>|<subscript list>;<subscript>
<subscript> ::= <expression>|<empty>
<assignment statement> ::= <assign operand>:=<expression>
<assign operand> ::= <identifier><subscript option>
<left part> ::= <monadic operator>|<operand><dyadic operator>
<monadic operator> ::= <monadic function name>|<monadic scalar
                       operator>|<monadic mixed operator>|
                       <monadic suboperator>
<monadic scalar operator> ::= +|-|&|%|*|LØG|CEIL|FLR|ABS|FACT|RNDM|
                              NØT|CIRCLE
<monadic mixed operator> ::= ,|RHØ|IØTA|BASVAL|TRANS|EPS
<monadic suboperator> ::= <monadic suboperator type><dimension part>
<monadic suboperator type> ::= <reduction type operator>|PHI|
                               SØRTUP|SØRTDN
<reduction type operator> ::= <dyadic scalar operator>/|<dyadic
                              scalar operator>\
<dimension part> ::= [<expression>]|<empty>
<dyadic operator> ::= <dyadic function name>|<dyadic scalar operator>|
                      <dyadic mixed operator>|<dot operator>|<dyadic
                      suboperator>
```

APPENDIX B (Continued)

<dyadic scalar operator> ::= +|-|&|*|LØG|MAX|MIN|%|RESD|CØMB|
AND|ØR|NAND|NØR|LSS|LEQ|=|GEQ|GTR|
NEQ|CIRCLE

<dyadic mixed operator> ::= ,|EPS|RHØ|IØTA|BASVAL|REP|RNDM|TAKE|DRØP|;

<dot operator> ::= <dyadic scalar operator>.<dyadic scalar operator>

<dyadic suboperator> ::= <dyadic suboperator type><dimension part>

<dyadic suboperator type> ::= PHI|/|\

<subroutine call> ::= <operand><dyadic subroutine name><expression>|
<monadic subroutine name><expression>|<niladic
subroutine name>

<transfer statement> ::= =:<expression>

APL SYNTAX - CONSTANTS & IDENTIFIERS

<data element> ::= <identifier>|<constant>

<identifier> ::= <letter>|<identifier><letter>|<identifier><digit>

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|Ø|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<constant> ::= <number>|<string>

<number> ::= <decimal number><exponent part>|<decimal number>|
<exponent part>

<decimal number> ::= <integer><decimal fraction>|<integer>|<decimal
fraction>

<integer> ::= <unsigned integer>|+<unsigned integer>|#<unsigned
integer>

<unsigned integer> ::= <digit>|<unsigned integer><digit>

<decimal fraction> ::= .<unsigned integer>

<exponent part> ::= <exponent symbol><exponent sign><unsigned
integer>

<exponent symbol> ::= @|E

<exponent sign> ::= #|-|+|<empty>

<empty> ::= {the null string of symbols}

<string> ::= "<proper string>"

<proper string> ::= <string element>|<proper string><string element>

APPENDIX B (Continued)

<string element> ::= <string character>| ""

<string character> ::= <visible string character>| <space>

<visible string character> ::= <letter>| <digit>| <special symbol>

<special symbol> ::= .|(|)|,|&|$|*|+|;|:|#|%|=|@|/|\|[|]|-

<space> ::= <single space>| <space><single space>

<single space> ::= {a single unit of horizontal spacing which is blank}