HEWLETT PACKARD

HP 3000 Series II Computer System

APL\3000
Reference Manual

HP 3000 Series II Computer System

APL\3000

Reference Manual

HEWLETT PACKARD

5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA 95050

Part No. 32105-90002
Product No. 32105A

Printed in U.S.A. 11/76

PRINTING HISTORY

New editions incorporate all update material since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover changes only when a new edition is published. If minor corrections and updates are incorporated, the manual is reprinted but neither the date on the title page and back cover nor the edition change.

First Edition. . . . . November 1976

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the most recent date on which the technical material on any given page was altered. If a page is simply re-arranged due to a technical change on a previous page, it is not listed as a changed page.

Page	Effective Date	Page	Effective Date
Title	Nov 1976	11-1 to 11-14	Nov 1976
ii to xiv	Nov 1976	12-1 to 12-9	Nov 1976
1-1 to 1-7	Nov 1976	A-1 to A-7	Nov 1976
2-1 to 2-12	Nov 1976	B-1 to B-4	Nov 1976
3-1 to 3-61	Nov 1976	C-1 to C-2	Nov 1976
4-1 to 4-23	Nov 1976	D-1 to D-2	Nov 1976
5-1 to 5-7	Nov 1976	E-1 to E-2	Nov 1976
6-1 to 6-14	Nov 1976	F-1 to F-2	Nov 1976
7-1 to 7-7	Nov 1976	G-1	Nov 1976
8-1 to 8-12	Nov 1976	H-1	Nov 1976
9-1 to 9-12	Nov 1976	I-1 to I-11	Nov 1976
10-1 to 10-15	Nov 1976		

PREFACE

This publication is the reference manual for APL\3000, a high-level programming language developed for use on the HP 3000 Series II Computer System.

Because of the unique structure of APL, this manual differs from most reference manuals, in that function descriptions are not arranged in alphabetical order, and more comprehensive descriptions are provided than would be necessary for better known languages such as FORTRAN or COBOL. Examples of all functions, however, are contained in alphabetical order in Appendix B.

Although it is possible to learn how to program in APL\3000 using this manual, such is not its main purpose, and therefore this manual assumes a knowledge of APL by the user. Further, because APL is an advanced computer language which has many applications in mathematical problem solving, it is assumed that readers have had mathematics training. For example, such terms as "non-singular arrays," "linearly independent columns," and so forth are introduced but not explained; and the reader is expected to be familiar with linear equations, logarithms, and pythagorean and hyperbolic functions.

Other publications which should be available for reference are:

<u>MPE Intrinsics Reference Manual</u> - Part Number 30000-90010
<u>MPE Commands Reference Manual</u> - Part Number 30000-90009
<u>Console Operator's Guide</u> - Part Number 30000-90013

This manual is divided into twelve sections, eight appendices, and a cross-reference index as follows:

CONVENTIONS USED IN THIS MANUAL

NOTATION	DESCRIPTION
[ ]	An element inside brackets is optional. Several elements stacked inside a pair of brackets means the user may select any one or none of these elements. Example: $\begin{bmatrix}\text{A}\\\text{B}\end{bmatrix}$ user may select A or B or neither
{ }	When several elements are stacked within braces the user must select one of these elements. Example: $\begin{Bmatrix}\text{A}\\\text{B}\\\text{C}\end{Bmatrix}$ user must select A or B or C.
underlining	Underlined words denote parameters which must be replaced by user-supplied variables. Example: CALL $\underline{\text{name}}$ — $\underline{\text{name}}$ one to 15 alphanumeric characters.
user input	Where it is necessary to distinguish user input from computer output, the input is underlined. Example: NEW NAME? $\underline{\text{ALPHA1}}$
$\underline{\text{return}}$	$\underline{\text{return}}$ underlined indicates a carriage return
. . .	A horizontal ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

TABLES

INTRODUCTION TO APL\3000

SECTION I

APL\3000 is a high-level programming language based on APL (A Programming Language) as developed by Dr. Kenneth Iverson.

Significant features of APL\3000 are as follows:

* APL\3000 is an interactive, terminal-oriented, problem solving language.

* APL\3000 provides a large set of functions and operators; thus programs may be written quickly and concisely and can be maintained with less effort than most high-level language programs.

* Intermediate code is compiled for each statement when it is first executed. Associated with the statement are binding parameters such as data types and array shapes. If these binding parameters are unchanged on subsequent executions, the statement need not be re-analyzed nor the intermediate code recompiled.

* A virtual memory scheme is used which allows extremely large, virtual work spaces.

* An additional structured-programming facility, APLGOL, is provided for creating user-defined functions.

* A modern cursor-oriented APL editor is provided to compose and edit APL programs.

* APL\3000 operates under control of the Multiprogramming Executive Operating System (MPE), allowing it to run in a multi-language environment.

APL\3000 CHARACTER SET

The APL\3000 character set consists of alphabetic characters, underscored alphabetic characters, numeric characters, the blank character, and special characters or graphic symbols. The complete set of characters is shown in figure 1-1. Note that the names for the special characters are for the characters themselves, and not necessarily for the functions they represent.

With the exception of ∆ ⍙ ⍵ ⍺ ∇ ⊂ ⊃ ∩ ∪ $ ⊢ ⊣ ◇ ⍝ ( ) [ ] ; : ¥ £ ⌶ , the special characters are used to denote primitive APL functions or APL operators (see Section III), and have fixed meanings in APL. Alphabetic characters are used to form names of variables and user-defined functions (see Section II). Numeric characters are used to form constants and may be used in conjunction with alphabetic characters to form names. The first character of a name must be alphabetic, or ∆ or ⍙. Any number of blank characters may be used to separate names, operators, functions, or constants, and may not be used to form names.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

¨	dieresis	¯	overbar	<	less
≤	not greater	=	equal	≥	not less
>	greater	≠	not equal	∨	or
∧	and	+	plus	-	bar
×	times	÷	divide	?	query
⍵	omega	∊	epsilon	⍴	rho
~	tilde	↑	up (arrow)	↓	down (arrow)
⍳	iota	○	circle	*	star
←	left (arrow)	→	right (arrow)	⍺	alpha
⌈	upstile	⌊	downstile	_	underbar
∇	del	∆	delta	∘	null
⎕	quad	[	open bracket	]	close bracket
⊂	open shoe	⊃	close shoe	∩	cap
∪	cup	⊥	base	⊤	top
|	stile	/	slash	\	slope
'	quote	(	open parenthesis	)	close parenthesis
;	semicolon	:	colon	,	comma
•	dot		space		
$	dollar	⊢	left tack	⊣	right tack
{	open bracket	}	close bracket	◇	diamond

The following characters are formed by overstriking

A̲ B̲ C̲ D̲ E̲ F̲ G̲ H̲ I̲ J̲ K̲ L̲ M̲ N̲ O̲ P̲ Q̲ R̲ S̲ T̲ U̲ V̲ W̲ X̲ Y̲ Z̲

⍙	delta under	⍒	del stile	⍋	delta stile
⍟	log	⊖	circle bar	⍉	circle slope
⌽	circle stile	⌿	slash bar	⍀	slope bar
⍲	nand	⍱	nor	⍫	del tilde
⍎	base null	⍕	top null	⍝	cap null
!	quote dot	⌶	I-beam	⍞	quote quad
⌹	domino				

Figure 1-1. APL\3000 Character Set

APL KEYBOARD

APL programs are generally composed and executed using terminal devices having special APL keyboards. The keyboard for the Hewlett-Packard HP 2641A APL terminal is shown in figure 1-2. Alphabetic characters are shown in uppercase but are accessed without using the shift key, while most special characters are accessed by depressing the SHIFT key (uppercase), then striking the special character key. Overstruck characters may be created by entering either character first, backspacing, then entering the other character. Alternatively, an expression may be created by entering characters in any order and overstriking in any order, as long as the visual effect is the correct expression. This is referred to in APL as visual fidelity. (Note that the letter E cannot be produced by entering F, backspace, then L.)

APL\3000 also permits the use of standard ASCII terminals to create and run programs. These terminals of course do not have the special APL character set shown on the keys. Appendix A shows how to form these special characters from such non-APL terminals.

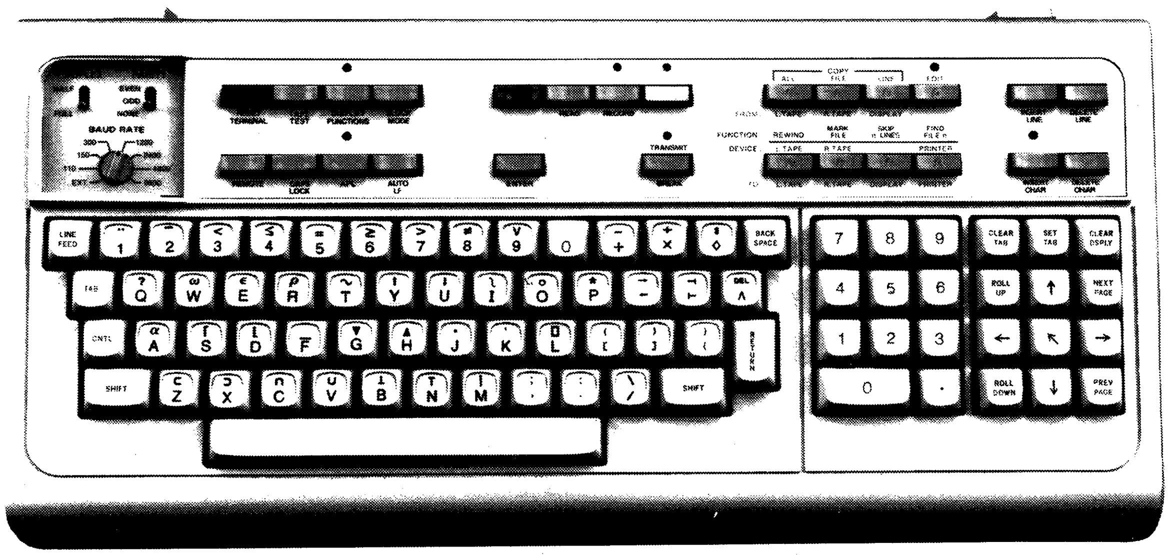


Figure 1-2. APL\3000 Keyboard

INITIATING AN APL\3000 SESSION

An APL\3000 session is initiated by entering

```
    (APL)  [sessionname,]username[/userpassw].acctname[/acctpassw]
           [,groupname[/grouppassw]]

           [;TERM = termtype]
           [;TIME = cpusecs]

                    BS
                    CS
           [;PRI =      ]
                    DS
                    ES

           [;INPRI = inputpriority]
           [;HIPRI]
```

where

sessionname	Arbitrary name used in conjunction with username and acctname parameters to form a fully-qualified session identity. Contains from 1 to 8 alphanumeric characters, beginning with a letter. Default: null session name. Note: A fully-qualified session identity consists of: [sessionname,]username.acctname and furnishes the minimum information required for log-on. Embedded blanks are forbidden in the username.acctname combination.
username	A user name, established by the Account Manager, that allows logging on under this account. This name is unique within the account and contains from 1 to 8 alphanumeric characters, beginning with a letter.
userpassw	User password, optionally assigned by the Account Manager. Contains from 1 to 8 alphanumeric characters, beginning with a letter. Separated from username by a slash with no surrounding blanks, as in username/userpassw.
acctname	Name of account, as established by the System Manager. Contains from 1 to 8 alphanumeric characters, beginning with a letter. Note: Must be preceded by a period as a delimiter.

acctpassw	Account password, optionally assigned by the System Manager. Contains from 1 to 8 alphanumeric characters, beginning with a letter. Separated from acctname by a slash with no surrounding blanks, as in acctname/acctpassw.
groupname	Name of file group to be used for local file domain and central processor unit time charges, as established by the Account Manager. Contains from 1 to 8 characters, beginning with a letter. Default: Home group if assigned by Account Manager.
grouppassw	Group password, optionally assigned by the Account Manager. Contains from 1 to 8 alphanumeric characters, beginning with a letter. Separated from groupname by a slash with no surrounding blanks, as in groupname/grouppassw. (Not needed when logging on under home group.)
termtype	Type of terminal used for input. Possible values are:

AJ	- Anderson-Jacobson
ASCII	- ASCII terminal
BP	- Bit-pairing
CDI	- Computer Devices, Inc.
CP	- Character-pairing
DM	- DataMedia
GSI	- GenCom Systems, Inc.
HP	- Hewlett-Packard

cpusecs	Maximum central processor unit time that session can use, entered in seconds. When this limit is reached, session is aborted. Must be a value from 1 to 32767. To specify no limit, enter question mark or omit this parameter.
PRI	The execution priority class that the command interpreter uses for the session, and also the default priority for all programs executed within the session. BS is highest priority; ES is lowest. If a priority is specified that exceeds the highest permitted for the account or user name by the system, MPE assigns the highest priority possible below BS. Default: CS.

inputpriority	Relative input priority used in checking against access restrictions imposed by the job fence, if one exists. (See the Console Operator's Guide for a description of the job fence.) Takes effect at log-on time. Must be a value from 1 (lowest priority) to 13 (highest priority). If a value is supplied that is less than or equal to the current job fence set by the Console Operator, session is denied access. Default: 8 if session/job initiation is enabled, 13 otherwise.
HIPRI	Request for maximum session-selection input priority, causing session to be scheduled regardless of current job fence or execution limit for sessions. Note: This parameter can be specified only by users with System Manager or System Supervisor capability.

The system prints the message

```
    APL\3000 HP32105 time and date
```

and awaits the first command.

RUNNING APL\3000

Once a session is initiated, APL can be run in either of two modes: calculator or immediate execution mode, or function definition mode.

In calculator (immediate execution) mode, expressions are created and the results may be displayed on the terminal immediately after entering a carriage return.

For example,

```
              6+7
        13
              6+7-9
        4
              6+7×(56÷7)
        62
              2 4 6 8+3
        5  7  9  11
```

Assign a value to the variable A:

```
        A←14+98.5
```

Note that a left arrow (assignment arrow) is used to specify the APL assignment function.

If just the name of the variable is entered, APL displays its value:

```
              A
         112.5
```

In function definition mode, the APL editor is used to form expressions into user-defined functions for later use. These user-defined functions formed with the editor may then be invoked from calculator mode or from within another function to perform the computation.

For example, CIRCLEAREA is a user-defined function to compute the areas of sectors of circles.

```
         [0]          AREA←RADIUS CIRCLEAREA DEGREES
         [1]          AREA←(○RADIUS*2)×DEGREES÷360
```

RADIUS and DEGREES are arguments of the function. RADIUS denotes the radius of the circle and DEGREES denotes the angle the sector subtends.

To run this user-defined function, the name is entered with the appropriate arguments as follows:

```
              163.2 CIRCLEAREA 37.4
         8692.791899
```

The value 163.2 is assigned to RADIUS and 37.4 is assigned to DEGREES. APL computes the area and displays the result.

The result can be assigned to the variable AREA by entering:

```
              AREA←163.2 CIRCLEAREA 37.4
              AREA
         8692.791899
```

TERMINATING AN APL\3000 SESSION

To terminate an APL session, either the)OFF or)CONTINUE command is used:

```
         )OFF
```

See Section XI for complete discussions of the)OFF and)CONTINUE commands.

ELEMENTS OF APL\3000

SECTION II

APL CONSTANTS

APL accepts both numeric and character constants. All numeric constants are decimal, and may include a decimal point if appropriate. They may be entered in the conventional manner as, for example,

```
      23
23
      3.14159
3.14159
```

or in scaled form. The scaled form consists of an integer or fractional decimal number called the fraction followed by the letter E followed by an integer called the scale. The scale is the power of ten by which the fraction is multiplied. Examples of scaled form are

```
      2E4
20000
      2E¯4
.0002
      .05E7
500000
```

Note that an overbar may be used to denote a negative scale but a plus sign may not be used with a positive scale.

Spaces are not allowed between the fraction and the E or between the E and the scale or an error message results. For example,

```
      .05 E7
SYNTAX ERROR
      .05 E7
      ↑
      .05E 7
SYNTAX ERROR
      .05E 7
         ↑
```

Negative numbers are specified by an overbar immediately preceding the number. For example,

```
      ¯45.6
¯45.6
      ¯53
¯53
      1E¯3
.001
      ¯1E3
¯1000
```

The overbar is used only in specifying a negative constant. It is not the equivalent of the bar (-), which is an APL function used either monadically to negate a value or dyadically to compute the difference between two arguments. For example,

```
      A←6.3
      A
6.3
      -A
¯6.3
```

SCALAR CONSTANTS

APL treats a single constant such as

```
297
2.97E8
34
5
```

as a <u>scalar</u> constant.

VECTOR CONSTANTS

A <u>vector</u> constant is entered as a sequence of numeric values. Each value must be separated from the next by one or more blank characters (spaces). The form of vector constants is

```
      ABC←2 4 6 8 10
      ABC
2  4  6  8  10
      XYZ←0 ¯2 1E12 2.34E¯4 ¯97.5 64 3.14159
      XYZ
0E00  ¯2E00  1E12  2.34E¯04  ¯9.75E01  6.4E01  3.14159E00
```

CHARACTER CONSTANTS

Character constants are entered by placing the characters between quote marks (' ') as follows:

```
      C←'A'
      C
A
```

APL displays the constant without the enclosing quotes as shown above.

APL\3000 treats a single character as a <u>scalar</u> character constant and a string of characters as a <u>vector</u> character constant. An <u>empty</u> vector (zero length) is specified by a consecutive pair of quote marks.

Examples:

```
      C←'CHARACTER VECTOR'
      C
CHARACTER VECTOR

      EMPTYVEC←''
      EMPTYVEC
```

If a quote character is to be included in a character string, it must be entered as a consecutive pair of quotes to distinguish it from the quotes enclosing the string. For example,

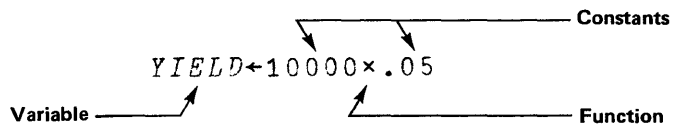
```
QUOTE←''''
TIME←'1 O''CLOCK'
```

is accepted and displayed by APL\3000 as

```
      QUOTE
'
      TIME
1 O'CLOCK
```

APL EXPRESSIONS

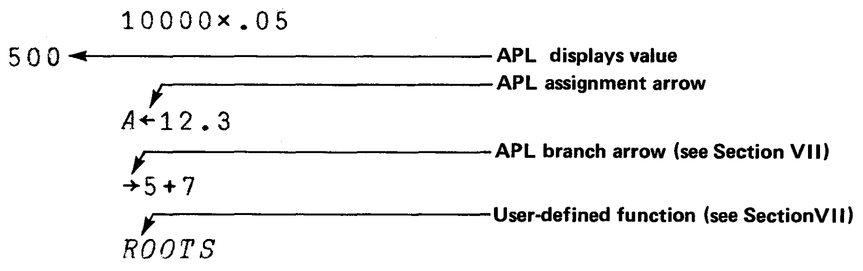
The expression is the basic executable unit in APL. An expression is written using names (variables and user-defined functions), constants, and APL functions or APL operators. For example,



The expression just shown assigns to the variable YIELD the value resulting when 10000×.05 (also an expression) is evaluated. The specification arrow (←) is an APL primitive function (see Section III) and means "is specified by." Thus YIELD is specified by the value 500. Several separate expressions may be written on one line if they are separated by diamonds (◊), as for example

```
YIELD←10000×.05◊INCOME←YIELD÷12
```

The result of an expression is displayed on the terminal unless the leftmost APL primitive function in the expression is an APL branch arrow (→), an APL assignment function (←), or the leftmost element is the name of a user-defined function which does not return a value. For example,



```
      10000×.05
500
      A←12.3
      →5+7
      ROOTS
```

APL displays value
APL assignment arrow
APL branch arrow (see Section VII)
User-defined function (see SectionVII)

Alternatively, if a variable has been assigned a value, that value can be displayed by entering the name of the variable.

```
      YIELD
500
      INCOME
41.66666667
```

The result of any portion of an expression can be displayed by assigning it to the output variable quad (⎕) (see Section III) at the appropriate point in the expression. For example,

```
      B←6+⎕←4+⎕←18
18
22
```

The specification arrow may appear any number of times in an expression and is treated in the same way as other primitive APL functions such as +, -, ×, ÷ and so forth.

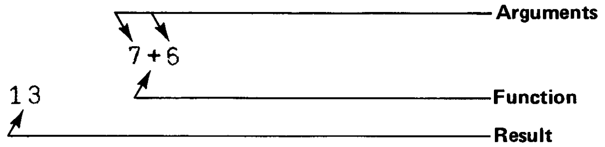
```
      A←6+B←4+C←14
      A
30
      B
18
      C
14
```

A second expression type is the branch expression, which may appear in a user-defined function to modify the normal order of execution. Typically, a branch evaluates the expression to the right of the arrow and transfers control to the line number of the APL function corresponding to the value of this expression. Branch expressions are described and illustrated in Section VII.

The final type of expression in APL is used to invoke a user-defined function. This type of expression also is described and illustrated in Section VII.

APL FUNCTIONS

An APL function may operate on zero, one, or two arguments, and optionally return a result. For instance, the primitive dyadic APL scalar function sum (+) takes two arguments and returns their algebraic sum as the result. This result then may be used as an argument for another function. For example,



```
      7+6
13
      5×13
65
```

MONADIC FUNCTIONS

A monadic function operates on only one argument. Negation, for example, is a monadic function which operates on the argument appearing to the right of the bar as follows:

```
      A←45
      -A
¯45
```

DYADIC FUNCTIONS

A dyadic function operates on two arguments, one to the left and one to the right of the function. Thus, the functions sum, difference, product, and quotient (represented by +, -, ×, and ÷, respectively) require two arguments. APL graphic symbols often have both monadic and dyadic meaning. For example, A-B signifies subtraction of B from A (dyadic), whereas -A signifies negation of A (monadic); and A÷B signifies the quotient of A and B (dyadic), whereas ÷A signifies the reciprocal of A (monadic).

```
     Dyadic Functions                  Monadic Functions

          7+6                                 +6
  13                                6
          7×6                                 ×6
  42                                1
          7÷6                                 ÷6
  1.166666667                       .1666666667
```

NILADIC FUNCTIONS

A niladic function has no argument. For example, if T is a user-defined function that returns the time of day, then entering T will cause APL to return the current time (no argument exists).

PRIMITIVE FUNCTIONS

A primitive APL function is a part of the APL language and cannot be redefined by the user. Such primitive APL functions are usually represented by a special graphic symbol. For example, + − × ⍋ ⍒ ← ÷ are primitive functions. A primitive function differs from a used-defined function in that a user-defined function consists of a number of expressions defined by a user to perform a specific computation.

The set of primitive functions is shown in figure 2-1. They are defined in Section III.

Primitive functions can produce different functional effects by combining an operator with the primitive function. For example, the sum of the elements of a vector constant will be computed if the sum (+) primitive scalar function is combined with the reduction (/) primitive operator, as follows:

```
      VEC←2 4 6 8 10
      +/VEC
30
```

Operators are discussed in Section III.

USER-DEFINED FUNCTIONS

A user-defined APL function is a series of APL expressions combined into one or more lines to form a function. This user-defined APL function then can be invoked from an APL expression to perform a computation on zero, one, or two arguments. For example, a user-defined function to return the distance traveled could be used in an APL expression as follows:

```
   30 D 10
```

If 30 represented miles per hour and 10 represented minutes, APL then would return 5. Note that spaces or other special characters must be used to separate the name of a user-defined function from its arguments. User-defined APL functions are discussed and illustrated in Section VII.

SYSTEM COMMANDS

In addition to using the APL language, it is also necessary to communicate directly with the APL system. A set of system commands is provided for this purpose. These commands are used for such things as logging on and off, saving a workspace for later use, and establishing passwords that lock workspaces so that they cannot be accessed by other users. System commands are discussed in Section XI.

PRIMITIVE SCALAR FUNCTIONS

Monadic		Dyadic	
+	conjugate	+	plus
−	negative	−	minus
×	signum	×	times
÷	reciprocal	÷	divide
|	magnitude	|	residue
⌊	floor	⌊	minimum
⌈	ceiling	⌈	maximum
?	roll		
*	exponential	*	power
⍟	natural logarithm	⍟	general logarithm
○	pi times	○	circular
!	factorial	!	binomial
~	not		
		∧	and
		∨	or
		⍲	nand
		⍱	nor
		<	less
		≤	not greater
		=	equal
		≥	not less
		>	greater
		≠	not equal

PRIMITIVE STRUCTURAL FUNCTIONS

Monadic		Dyadic	
⍴	shape	⍴	reshape
,	ravel	,	catinate/laminate
⌽ ⊖	reversal	⌽ ⊖	rotate
⍉	transpose	⍉	transpose

Figure 2-1. APL\3000 Primitive Functions (Sheet 1 of 2)

PRIMITIVE SELECTION FUNCTIONS

Dyadic	
↑	take
↓	drop
⌿ /	compress
⍀ \	expand
[ ]	index

PRIMITIVE SELECTOR GENERATOR FUNCTIONS

Monadic		Dyadic	
⍳	index generator	⍳	index of
⍋	grade up		
⍒	grade down		
		∊	membership
		?	deal

PRIMITIVE NUMERICAL FUNCTIONS

Monadic		Dyadic	
⌹	matrix inverse	⌹	matrix divide
		⊥	decode
		⊤	encode

PRIMITIVE TRANSFORMATION FUNCTIONS

Monadic		Dyadic	
⍎	execute	⍎	execute
⍕	format	⍕	format

Figure 2-1. APL\3000 Primitive Functions (Sheet 2 of 2)

APL ORDER OF ASSOCIATION

In APL, there is no hierarchy of association among functions (such as associating division before addition). Within a given level of parentheses in an expression, association is strictly right to left.

If parentheses are used, then the part of the expression within matching parentheses is associated right to left before applying its result to any function outside the parentheses. For example,

```
          18÷6+3
    2
          18÷(6+3)
    2
          (18÷6)+3
    6
```

ARRAYS

An array is a collection of zero or more values (elements), all of which may be represented by an array name. An array with zero elements is an empty array; a scalar (single) value is dimensionless; and a vector value such as

```
2 4 6 8 10
```

is a single-dimensional array and is considered to be of rank 1. A matrix, which has two dimensions, or axes, such as

```
2 4 6 8 10
1 3 5 7 9
```

is a two-dimensional array of rank 2. APL\3000 allows arrays up to and including a maximum of 63 dimensions.

The elements of a vector (one-dimensional) array may be selected by enclosing the indices of the desired elements in brackets, called indexing. For example, variable XQR has the following values

```
      XQR←2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
```

If 1-origin indexing is in effect (see page 2-11), elements 3, 4, and 8 can be indexed by entering XQR[3 4 8]. APL returns

```
          XQR[3 4 8]
    6  8  16
```

Another example:

```
          CHAR←'CHARACTER STRING'
          CHAR[5 6 7 14 15 16]
    ACTING
```

APL displays a vector array on one or more output lines. The vector can be formed into a more complex structure, containing more dimensions, with the reshape (ρ) function (see Section III):

```
        2 6ρXQR
   2   4   6   8  10  12
  14  16  18  20  22  24
        4 4ρCHAR
CHAR                      Name of vector
ACTE                      Reshape function
R ST                      Number of columns
RING                      Number of rows
```

The left arguments in the above examples (2 6 and 4 4) specify the shape of the resulting array. The first example produces an array with two rows and six columns. The second example produces an array with four rows and four columns. More complex shapes can be created. For example,

```
        CHAR←'1234567890ABCDEFGHIJKLMN'
        SHAPE←3 4 2ρCHAR
        SHAPE                     Vector name
12                                Reshape function
34                                Number of columns
56                                Number of rows
78                                Number of planes
                                  Name of array
90
AB
CD
EF

GH
IJ
KL
MN
```

Note that when all the values of one axis have been displayed, a line is skipped and the next set of axis values is then returned.

The shape of an array can be determined by entering the monadic shape (ρ) function (see Section III) followed by the array as its argument.

```
        ρRESHAPE1
2  6
        ρRESHAPE2
4  4
        ρSHAPE
3  4  2
```

The elements of a multi-dimensional array can be selected by indexing in the same manner as shown for vector arrays, except that an index is provided for each axis. For example, to select and display the fourth element in the second row of array RESHAPE1:

```
          RESHAPE1
      2   4   6   8  10  12
     14  16  18  20  22  24
          RESHAPE1[2;4]
    20
```

The next example selects the second, third, and fourth elements from the third and second rows of array RESHAPE2.

```
          RESHAPE2
    1234
    5678
    90AB
    CDEF
          RESHAPE2[3 2;2 3 4]
    0AB
    678
```

To select the second column of the first four rows of the second plane of SHAPE:

```
          SHAPE
    12
    34
    56
    78

    90
    AB
    CD
    EF

    GH
    IJ
    KL
    MN
          SHAPE[2;1 2 3 4;2]
    0BDF
```

The foregoing examples assume that the elements are numbered 1, 2, 3, ... n, and therefore is called 1-origin indexing. Indices may begin with 0, called 0-origin indexing, by setting the index origin to 0 with the system variable ⎕IO (see Section IV). For example,

```
          CHAR
    1234567890ABCDEFGHIJKLMN
          ⎕IO←1
          CHAR[2 4 6]
    246 ←──────────────── Selects elements 2, 4, and 6

          ⎕IO←0
          CHAR[2 4 6]
    357 ←──────────────── Selects elements 3, 5, and 7
```

WORKSPACES AND LIBRARIES

When an APL session is initiated, the system reserves a block of storage for this session. This storage is called a workspace, and contains all the information to perform calculations, save the results, etc. This workspace also contains the definitions of user-defined functions as well as the names and values of any variables. The workspace also includes areas used by the system for the temporary storage of intermediate results while a calculation is in process, etc. The workspace being used is called the active workspace. Workspaces may have names assigned to them so that they can be saved as duplicates of the active workspace for later use. These saved workspaces are called stored workspaces.

The set of saved workspaces is called a library. Each workspace is identified by group and account names as well as the actual name assigned to it. In referring to workspaces in the user's own library, however, the group and account names may be omitted, because they are supplied automatically.

In systems with multiple APL users, it is often convenient to use functions or variables contributed by others. A user may activate an entire workspace saved by another user, or he may copy selected items from another user's workspace. In order to copy another user's workspace, the group and account names, if different, must be supplied together with the workspace name.

Some libraries (usually identified by a special group and account name, for example, PUB.SYS) are not assigned to individual users, but are designated as public libraries. There may be restrictions, however, on who can save, delete, or modify a workspace in a public library. In general, a public library workspace can be re-saved or deleted only by the user who first saved it.

APL\3000 PRIMITIVE FUNCTIONS AND OPERATORS

SECTION III

Primitive functions in APL consist of two types: primitive scalar functions and primitive mixed functions. Primitive scalar functions operate on scalar arguments or arrays on an element-by-element basis, producing results of the same rank and shape. Primitive mixed functions also operate on scalars and arrays, but may produce results which differ in rank and shape from the original argument arrays.

Four primitive operators can be applied to the primitive scalar dyadic functions to produce different effects. Operators are discussed starting on page 3-17.

PRIMITIVE SCALAR FUNCTIONS

Primitive scalar functions are of two types: monadic and dyadic.

A monadic primitive scalar function applies to one scalar argument, or to each element of one array argument. If the argument is an array, the result is an array of the same shape as the argument. Each element of the resulting array is produced as the monadic function is applied to the corresponding element of the original argument array. For example,

```
          A
¯7  34.1  ¯6.035  155.64
          -A
7  ¯34.1  6.035  ¯155.64
```

A dyadic primitive scalar function applies to a pair of arguments. The arguments can be scalars or arrays. If arrays are used, both must be of the same rank and shape, or, if not, one must be a scalar or unit (one-element array).

When arrays of the same shape are used as arguments, each element of the left argument is paired with the corresponding element in the right argument. For example,

```
          A
¯7  34.1  ¯6.035  155.64
          B
¯3  1.2  ¯.35  10
          A×B
21  40.92  2.11225  1556.4
```

If one of the arguments is a scalar or unit (one-element) array, then that element is paired with every element of the other argument (extended) as follows:

```
          C
45.3
          B
¯3  1.2  ¯.35  10
          C×B
¯135.9  54.36  ¯15.855  453
```

Primitive scalar functions are typically applied to all numbers with the exception that arguments to the boolean functions (∧ ∨ ⍲ ⍱ ~) are restricted to the binary values 0 and 1. Additionally, the functions = and ≠ may be applied to character arguments.

Monadic primitive scalar functions are shown in table 3-1 and dyadic primitive scalar functions are shown in table 3-2. Note that most symbols (such as + and -) are used both monadically and dyadically; whether they are interpreted as monadic or dyadic depends on the context in which they are used.

Some primitive dyadic scalar functions possess a <u>left identity</u> and/or a <u>right identity</u>. A <u>left identity</u> is such that if L is the left identity for the function <u>fn</u>, then L<u>fn</u>X equals X for all X.

For a <u>right identity</u> R, X<u>fn</u>R equals X for all X.

Table 3-3 shows the identity elements of the primitive dyadic functions. Note that the relational functions equal (=), not equal (≠), less (<), greater (>), not less (≥), and not greater (≤) do not possess true identity elements when used as relational functions, but do when used as boolean functions (applied only to the values 0 and 1).

PLUS, MINUS, TIMES, AND DIVIDE FUNCTIONS

Plus (+), minus (-), times (×), and divide (÷) are dyadic functions which perform the same functions in APL as they do in standard arithmetic operation. (Note that in APL, 0÷0 returns a value of 1; however, when X ≠ 0, X÷0 results in an error.)

Examples of these four functions are:

```
      A
5  34.2  ¯7  ¯6.035  155.64  1
      B
¯3  1.2  ¯.35  10  ¯.75  ¯1
      A+B
2  35.4  ¯7.35  3.965  154.89  0
      A-B
8  33  ¯6.65  ¯16.035  156.39  2
      A×B
¯15  41.04  2.45  ¯60.35  ¯116.73  ¯1
      A÷B
¯1.666666667  28.5  20  ¯.6035  ¯207.52  ¯1
      A-5
0  29.2  ¯12  ¯11.035  150.64  ¯4
```

Table 3-1. Monadic Primitive Scalar Functions

SYMBOL	NAME	DEFINITION	EXAMPLE

+	Conjugate	+A is A	`      A`
			`6`
			`      +A`
			`6`

−	Negative	−A is 0 − A	`      A`
			`6`
			`      -A`
			`¯6`

×	Signum	×A is (A>0)−A<0	`      A`
			`6`
			`      ×A`
			`1`

÷	Reciprocal	÷A is 1 ÷ A	`      A`
			`6`
			`      ÷A`
			`.1666666667`

|	Magnitude	Absolute value	`      B`
			`4.743  ¯4.743`
			`      |B`
			`4.743  4.743`

⌊	Floor	Least integer	`      B`
			`4.743  ¯4.743`
			`      ⌊B`
			`4  ¯5`

⌈	Ceiling	Greatest integer	`      B`
			`4.743  ¯4.743`
			`      ⌈B`
			`5  ¯4`

?	Roll	?A is random choice from set of A consecutive integers beginning at ⎕IO.	`      C`
			`6  6  6  6  6  6`
			`      ?C`
			`1  3  4  2  5  2`

*	Exponential	e^A	`      A`
			`6`
			`      *A`
			`403.4287935`

⍟	Natural logarithm	ln A or $\log_e$ A	`      A`
			`6`
			`      ⍟A`
			`1.791759469`

○	Pi times	$\pi \times$ A	`      D`
			`1  2`
			`      ○D`
			`3.141592654  6.283185307`

Table 3-1. Monadic Primitive Scalar Functions (continued)

!	Factorial	!A = A × A − 1x. . .x1	A 6 !A 720
~	Not	~1 is 0, ~0 is 1. Truth table defined for 0 and 1 only.	E 1 0 ~E 0 1

Table 3-2. Dyadic Primitive Scalar Functions

SYMBOL	NAME	DEFINITION	EXAMPLE
+	Plus	Add	6+7 13
−	Minus	Subtract	6-7 ¯1
×	Times	Multiply	6×7 42
÷	Divide	Divide	6÷7 .8571428571
|	Residue	Remainder after divide	7|43.36 1.36
⌊	Minimum	Smaller of two values	6⌊7 6
⌈	Maximum	Greater of two values	6⌈7 7
*	Power	Product of B× B A times. $[B^A]$	2*8 256
⊛	General logarithm	$\log_B A$	10⊛1003 3.001300933

Table 3-2. Dyadic Primitive Scalar Functions (Continued)

SYMBOL	NAME	DEFINITION
○	Circular, Hyperbolic, and Pythagorean functions	¯7○X = Artanh X ¯6○X = Arcosh X ¯5○X = Arsinh X ¯4○X = (¯1+X*2)*.5 ¯3○X = Arctan X ¯2○X = Arccos X ¯1○X = Arcsin X 0○X = (1−X*2)*.5 1○X = Sine X 2○X = Cosine X 3○X = Tangent X 4○X = (1+X*2)*.5 5○X = Sinh X 6○X = Cosh X 7○X = Tanh X
!	Binomial	$\binom{A}{B}$
∧	And	
∨	Or	
⍲	Nand	
⍱	Nor	

A	B	A∧B	A∨B	A⍲B	A⍱B
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

SYMBOL	NAME	DEFINITION
<	Less	Result is 1 (TRUE) if relation holds and 0 (FALSE) if it does not hold. For example, 4<6 is 1, 4>6 is 0.
≤	Not greater	
=	Equal	
≥	Not less	
>	Greater	
≠	Not equal	

Table 3-3. Identity Elements of Dyadic Primitive Scalar Functions

FUNCTION	SYMBOL	IDENTITY ELEMENT	LEFT OR RIGHT
Plus	+	0	Both
Minus	−	0	Right
Times	×	1	Both
Divide	÷	1	Right
Residue	|	0	Left
Minimum	⌊	The largest representable number	Both
Maximum	⌈	The greatest in magnitude of representable *negative* numbers	Both
Power	*	1	Right
Logarithm	⍟	None	
Circle	○	None	
Binomial	!	1	Left
And	∧	1	Both
Or	∨	0	Both
Nand	⍲	None	
Nor	⍱	None	
Less	<	0	Left
Not greater	≤	1	Left
Equal	=	1 Apply for boolean	Both
Not less	≥	1 arguments only	Right
Greater	>	0	Right
Not equal	≠	0	Both

RESIDUE FUNCTION

Residue (|) is a dyadic primitive scalar function which returns the remainder when a value X is divided into a value Y; that is, X|Y returns the remainder when X is divided into Y.

The following rules apply for zero and non-zero values:

* If X = 0, X|Y ↔ Y.

* If X = 0, X|Y ↔ a value between 0 and X. The result can equal 0 but not X (equal to Y-N|X for some integer N).

Examples of the residue function are

```
              A
      5  34.2  ¯7  ¯6.035  155.64  1
              B
      ¯3  1.2  ¯.35  10  ¯.75  ¯1
              A|B
      2  1.2  ¯.35  ¯2.07  154.89  0
              B|A
      ¯1E00  6E¯01  2.220446049E¯16  3.965E00  ¯3.6E¯01  0E00
```

CONJUGATE FUNCTION

Conjugate (+), a monadic primitive scalar function, returns the value of its argument unchanged. For example,

```
              A
      5  34.2  ¯7  ¯6.035  155.64  1
              +A
      5  34.2  ¯7  ¯6.035  155.64  1
```

NEGATIVE FUNCTION

The monadic primitive scalar function negative (-) returns the value of its argument with the opposite sign. For example,

```
              A
      5  34.2  ¯7  ¯6.035  155.64  1
              -A
      ¯5  ¯34.2  7  6.035  ¯155.64  ¯1
```

SIGNUM FUNCTION

The signum function (×) is a monadic primitive scalar function which returns a value that is dependent upon the sign of its argument. If A is negative, then ×A is¯1; if A is positive, ×A is 1; if A is 0, then ×A is 0.

Examples of the signum function are:

```
              A
      5  34.2  ¯7  ¯6.035  155.64  1
              ×A
      1  1  ¯1  ¯1  1  1
```

RECIPROCAL FUNCTION

The monadic primitive scalar function reciprocal (÷) returns the value 1÷X for the argument X. For example,

```
      A
5   34.2  ¯7  ¯6.035   155.64   1
      ÷A
2E¯01  2.923976608E¯02  ¯1.428571429E¯01  ¯1.657000829E¯01  6.425083526E¯03
      1E00
```

Note that when X is 0, an error results.

MAGNITUDE FUNCTION

The magnitude (|) monadic primitive scalar function returns the absolute value of its argument. For example,

```
      A
5   34.2   ¯7   ¯6.035   155.64   1
      |A
5   34.2   7   6.035   155.64   1
```

BOOLEAN FUNCTIONS

The five boolean functions apply only to the values 0 and 1. APL interprets 1 as being true and 0 as being false.

Four of the boolean functions are dyadic, the other, not (~), is monadic. A truth table for the functions is:

		AND	OR	NAND	NOR	NOT	
X	Y	X∧Y	X∨Y	X⍲Y	X⍱Y	~X	~Y
1	1	1	1	0	0	0	0
1	0	0	1	1	0	0	1
0	1	0	1	1	0	1	0
0	0	0	0	1	1	1	1

RELATIONAL FUNCTIONS

The relational functions are dyadic primitive scalar functions and are listed below.

Less (<)
Not greater (≤)
Equal (=)
Not less (≥)
Greater (>)
Not equal (≠)

The functions < ≤ ≥ > only apply to numeric arguments, while = and ≠ apply to numeric and character arguments. Note that the result of '1'=1 is always 0 and that '1'≠1 is always 1.

The result is 1 (true) if the compared relation is true and 0 (false) if the compared relation is false. For example,

```
            A
5   34.2  ¯7  ¯6.035   155.64   1
            B
¯3   1.2  ¯.35   10   ¯.75   ¯1
            A<B
0   0   1   1   0   0
            A>B
1   1   0   0   1   1
```

The results of comparing the arguments of relational functions are not absolute, but are within a certain comparison tolerance whose value is contained in the system variable ⎕CT. The question "is A equal to B" is straightforward unless floating-point numbers represented in a finite number of bits (64 bits for APL\3000) are involved. The A=B question then becomes harder to answer because many floating-point numbers cannot be represented exactly in 64 bits. Thus, problems arise if the equals test is defined to be "exact." The following example illustrates this point.

```
          A←÷97◊A
1.030927835E¯02
          ⎕CT←0            ⍝  THIS MAKES '=' AN EXACT TEST
          1=97×A
0
          ⍝  BECAUSE 1/97 CANNOT BE STORED EXACTLY
          ⍝  THEN 'A' IS NOT A NUMBER THAT CAN
          ⍝  BE MULTIPLIED BY 97 TO RETURN 1
```

This particular way to define = is then not very consistent with the way = would be expected to act. Thus the definition of = (and some related functions) is not an "exact" definition, but is relative to the magnitude of the operands and the value of ⎕CT. The definition is

```
X←|A-B                   [1]
Y←⌈/(|A),|B              [2]
IF (Y×⎕CT)≥X THEN        [3]
A IS EQUAL TO B
```

Notice that the preceding set of equations, while concise and correct, is difficult to understand. Paraphrasing them as follows may help:

Equation [1] sets the variable X to the absolute value of the difference of the two arguments A and B.

Equation [2] sets Y to the absolute value of the larger of the two arguments A and B.

The third (and crucial) equation [3] states that the arguments are defined to be equal if ⎕CT times the larger of the arguments (Y) is larger that the difference between the arguments.

Note that ⎕CT does not specify the absolute difference between the arguments but the difference relative to the size of the arguments. Thus two big numbers need not be as close, in an absolute sense, as two small numbers. Note that under this definition, if ⎕CT is 0, the equals test is exact in that the difference between the arguments A and B must be 0, exactly, for equation [3] to be true.

There are several APL functions (such as index of, index generator, deal, roll, etc) which will result in an error unless the operand(s) are considered "integers." In APL\3000, this test for integer is done in the following way:

1) First, the integer closest to the argument is obtained.

2) Second, the integer obtained in 1) is compared in a relative sense to the argument.

3) If the integer from 1) is relatively equal to the argument, that integer is used as the argument.

An example:

```
            A←300ρι1000
            A[250]
      250
            ⎕CT←1E¯10
            A[250+1E¯11]
      250
            A[250+1E¯10]
      DOMAIN ERROR
             A[250+.1E¯09]
             ↑
```

The relational functions act as boolean functions when they are used with the boolean arguments 0 and 1. Table 3-4 shows the boolean functions and relational functions for all possible values of the two boolean arguments.

Table 3-4. Truth Table for Boolean Functions

		AND	OR	NAND	NOR	LESS	NOT GREATER	EQUAL	NOT LESS	GREATER	NOT EQUAL (XOR)	NOT	
X	Y	X∧Y	X∨Y	X⍲Y	X⍱Y	X<Y	X≤Y	X=Y	X≥Y	X>Y	X≠Y	~X	~Y
1	1	1	1	0	0	0	1	1	1	0	0	0	0
1	0	0	1	1	0	0	0	0	1	1	1	0	1
0	1	0	1	1	0	1	1	0	0	0	1	1	0
0	0	0	0	1	1	0	1	1	1	0	0	1	1

MINIMUM AND MAXIMUM FUNCTIONS

The minimum (⌊) and maximum (⌈) functions are dyadic primitive scalar functions that compare two values and return the smaller or larger of the two. Examples are

```
               A
5   34.2   ¯7   ¯6.035   155.64   1
               B
¯3   1.2   ¯.35   10   ¯.75   ¯1
               A⌊B
¯3   1.2   ¯7   ¯6.035   ¯.75   ¯1
               A⌈B
5   34.2   ¯.35   10   155.64   1
```

FLOOR AND CEILING FUNCTIONS

Floor (⌊) and ceiling (⌈) are monadic primitive scalar functions. The floor function returns the largest integer value which does not exceed the value of its argument. The ceiling function returns the smallest integer value which is not less than the value of its argument.

Examples are

```
               A
5   34.2   ¯7   ¯6.035   155.64   1
               ⌊A
5   34   ¯7   ¯7   155   1
               ⌈A
5   35   ¯7   ¯6   156   1
```

The results returned by the floor and ceiling functions depend on the value of the comparison tolerance (⎕CT). See page 3-9 for a description of results which are dependent on ⎕CT. An example is:

```
               ⎕CT←1E¯13
               X←97×1÷97
               ⌊X
1
               ⌈X
1
               ⎕CT←0
               ⌊X
0
               ⌈X
1
```

ROLL (RANDOM NUMBER) FUNCTION

Roll (?) is a monadic primitive scalar function (named after the roll of a die) which produces a pseudo-random choice with replacement between ⎕IO and A-1-⎕IO (depending on the index origin presently in effect). For example, if the argument is 6 and the index origin is 1, then ?6 will produce a random integer between 1 and 6.

Examples are

```
      ⎕IO←1
      ?6 6 6 6 6 6 6
6  4  1  3  2  2  6
      ?7 7 7 7 7 7 7
2  6  2  2  7  6  7
      ⎕IO←0
      ?6 6 6 6 6 6 6
3  4  0  5  1  3  0
      ?7 7 7 7 7 7 7
6  4  0  1  6  5  0
```

The result produced by the roll function is always a non-negative integer.

POWER FUNCTION

The power function (*) is a dyadic primitive scalar function which, in the form X*N, raises X to the power N. X*-N therefore is the reciprocal of X*N, and X*÷N is the Nth root of X.

Examples are

```
      A
5  34.2  ¯7  ¯6.035  155.64  1
      N←2 4 6 ¯2 ¯4 ¯6
      A*N
2.5E01  1.36805773E06  1.17649E05  2.745651746E¯02  1.704178616E¯09  1E00
      0*0
1
```

Note that APL defines the indeterminate case 0*0 as 1.

The power function results in a domain error if the following two restrictions are not observed for X*N:

1. If X = 0, N must be non-negative.

2. If X < 0, N must be an integer or a rational number with an odd denominator.

EXPONENTIAL FUNCTION

The exponential function (*) is a monadic primitive scalar function where *X is e*X and e is the natural logarithm base, which is 2.718281828459045.

Examples are

```
      A
5  34.2  ¯7  ¯6.035  155.64  1
      *A
1.484131591E02  7.126417816E14  9.118819656E¯04  2.393496527E¯03  3.922772873E67
      2.718281828E00
```

NATURAL LOGARITHM FUNCTION

The natural logarithm function (⍟) is a monadic primitive scalar function and the inverse of the exponential function. The domain of the natural logarithm function is limited to positive numbers.

Examples of the natural logarithm function are

```
          ⍟*1
      1
          X←2 4 6 8 10
          ⍟X
      .6931471806  1.3862943611  1.7917594692  2.0794415417  2.302585093
```

GENERAL LOGARTIHM FUNCTION

The general logarithm function (⍟) is a dyadic primitive scalar function in which B⍟A is the "log base B of A." The general logarithm function is the inverse of the power function in that B*B⍟A and B⍟B*A both equal A.

Examples of the general logarithm function are

```
          X
      2 4 6 8 10
          2⍟X
      1 2 2.584962501 3 3.321928095
          10⍟X
      .3010299957  .6020599913  .7781512504  .903089987  1
```

CIRCULAR HYPERBOLIC AND PYTHAGOREAN FUNCTIONS

The symbol ○ signifies a monadic primitive function which returns a value equal to PI times the argument. For example,

```
          Y←0 1 2 4 6 8 10
          ○Y
      0  3.141592654  6.283185307  12.566370614  18.849555922  25.132741229
          31.415926536
```

The same symbol also can be used to specify a dyadic primitive scalar function to signify 15 circular, hyperbolic, and pythagorean functions. When used in this manner, an integer in the range ¯7 to 7 as the left argument signifies the particular function:

```
    ¯7○X = Arctanh X
    ¯6○X = Arccosh X
    ¯5○X = Arcsinh X
    ¯4○X = (¯1+X*2)*.5
    ¯3○X = Arctan X
    ¯2○X = Arccos X
    ¯1○X = Arcsin X
     0○X = (1-X*2)*.5
     1○X = Sine X
     2○X = Cosine X
     3○X = Tangent X
     4○X = (1+X*2)*.5
     5○X = Sinh X
     6○X = Cosh X
     7○X = Tanh X
```

The six circular functions are:

```
    1○X = Sin
    2○X = Cos
    3○X = Tan
    ¯1○X = Arcsin
    ¯2○X = Arccos
    ¯3○X = Arctan
```

The right argument of the above circular functions is in radians. For example,

```
      Z←¯7 ¯5 ¯3 ¯1 0 1 3 5 7
      1○Z
¯.6569865987  .9589242747  ¯.1411200081  ¯.8414709848  0  .8414709848
      .1411200081  ¯.9589242747  .6569865987
      2○Z
.7539022543  .2836621855  ¯.9899924966  .5403023059  1  .5403023059
     ¯.9899924966  .2836621855  .7539022543
      3○Z
¯.8714479827  3.3805150062  .1425465431  ¯1.5574077247  0  1.5574077247
     ¯.1425465431  ¯3.3805150062  .8714479827
     ¯3○Z
¯1.4288992722  ¯1.3734007669  ¯1.2490457724  ¯.7853981634  0  .7853981634
      1.2490457724  1.3734007669  1.4288992722
```

The six hyperbolic functions are:

```
    5○X = Sinh
    6○X = Cosh
    7○X = Tanh
    ¯5○X = Arcsinh
    ¯6○X = Arccosh
    ¯7○X = Arctanh
```

The functions sinh (5○X) and cosh (6○X) are the odd and even components of the exponential function. For example, 5○X is odd, 6○X is even, and the sum (5○X) + 6○X is equivalent to *X.

```
            X←8
            5○X
      1490.478826
            6○X
      1490.479161
            (5○X)+6○X
      2980.957987
           ×X
      2980.957987
```

The tanh function (7○X) is similar to the definition of the tangent, which is

$$\tanh = \frac{\sinh}{\cosh}$$

thus

```
            7○X
      .9999997749
            (5○X)÷6○X
      .9999997749
```

The three pythagorean functions are:

```
    0○X = (1-X*2)*.5
   ¯4○X = (¯1+X*2)*.5
    4○X = (1+X*2)*.5
```

The pythagorean functions are related to the properties of a right triangle as shown in figure 3-1.

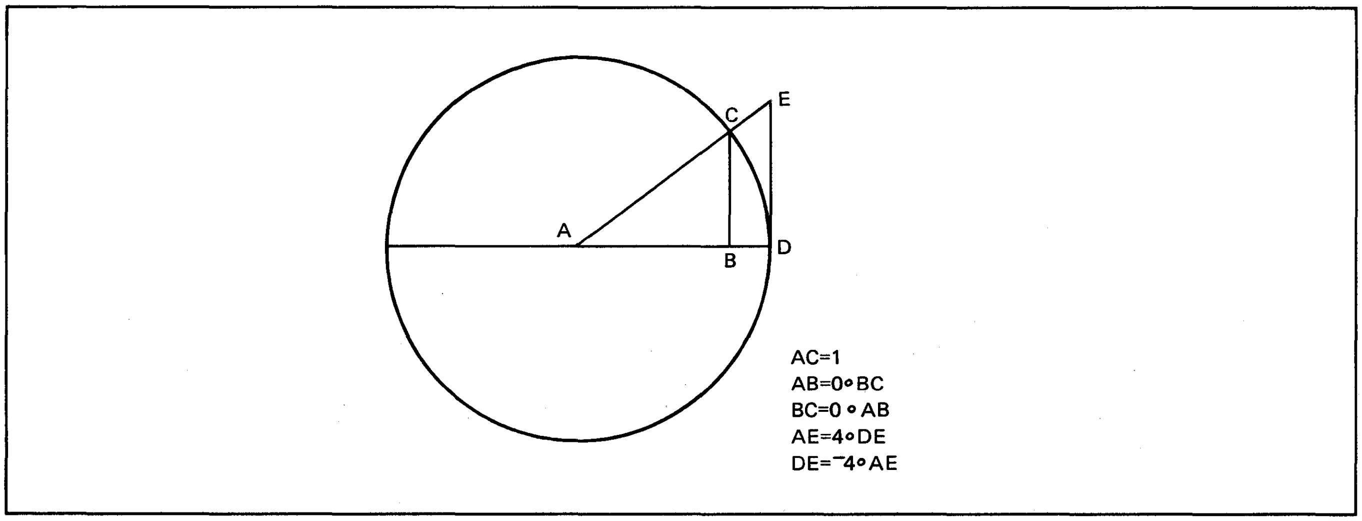


Figure 3-1. Pythagarean Functions

Each of the circular, hyperbolic, and pythagorean functions has an inverse in the same family; thus, (-I)○X is the inverse of I○X. Some of the functions are not isomorphic, however, and thus their inverses can have many values. The principal values are shown below:

```
          ARCCOSH     V←¯6○X   V≥0
                      V←¯4○X   V≥0
          ARCCOS      V←¯2○X   (V≥0)∧(X≤○1)
          ARCSIN      V←¯1○X   (|V)≤○.5
                      V←0○X    V≥0
                      V←4○X    V>0
```

Domain restrictions are as follows:

```
          ARCTANH     ¯7○Y    1>|Y
          ARCCOSH     ¯6○Y    Y≥1
                      ¯4○Y    1≤|Y
          ARCCOS      ¯2○Y
                              1≥|Y
          ARCSIN      ¯1○Y
                       0○Y     Y≤1
```

FACTORIAL FUNCTION

The factorial function (!) is a monadic primitive scalar function. For a positive integer argument X, !X is the product of all positive integers up to and including X. Thus, !X = X×!X-1, or !X-1 = (!X)÷X. This relation is used to extend the function to both positive integer and non-integer values and to negative non-integer values. Negative integer values are excluded from the domain of the factorial function because the relation described above leads to the expression (!0)÷0, or 1÷0 for !¯1.

Examples of the factorial function are:

```
              X←¯2.5 ¯1.4 ¯.5 0 1 2 3 4 5
              !X
          2.363271801  ¯3.722980622  1.772453851  1  1  2  6  24  120
```

BINOMIAL FUNCTION

The binomial function (!) is a dyadic primitive scalar function. For non-negative integer arguments X and Y, the function X!Y is defined as the number of different ways X things can be chosen from Y things. The expression (!Y)÷(!X)×(!Y-X), however, produces an equivalent definition which is used to extend the binomial function definition to all numbers.

Unlike the factorial function, which excludes negative integers from its domain, the binomial function does not. This is because any implied division by zero in the numerator !Y is accompanied by a corresponding division by zero in the denominator. Thus, the binomial function extends correctly to all numbers.

Examples of the binomial function are:

```
               X←0 1 2 3 4 5
               Y←6
               X!Y
      1  6  15  20  15  6
```

OPERATORS

Operators are combined with dyadic primitive scalar functions to produce different functions. For example, the reduction operator (/) can be combined with the dyadic primitive scalar function plus (+) to sum the elements of a vector to produce a scalar sum as follows:

```
               X
      0  1  2  3  4  5
               +/X
      15
```

The four major operators are:

* Reduction (/)

* Scan (\)

* Inner product (.)

* Outer product (°.)

Additionally, an auxiliary axis operator may be used in conjunction with the scan and reduction operators and the primitive mixed functions to specify the coordinate (axis) over which the operation is to occur.

REDUCTION OPERATOR

The reduction operator (/) applies a dyadic primitive scalar function which precedes it to elements in the right argument, producing a result whose rank is one less than that of the argument (thus reducing the rank). For example,

```
               VECTOR←2 4 6 8 10
               +/VECTOR
      30
               -/VECTOR
      6
```

+/VECTOR is the equivalent of 2+4+6+8+10

-/VECTOR is the equivalent of 2-4-6-8-10

The reduction operator performs as though the function were placed between adjacent pairs of elements of VECTOR and associating right-to-left.

The last example demonstrates the right-to-left association, which causes -/VECTOR to result in the alternating sum of the elements of VECTOR. The alternating sum is the sum obtained after multiplying alternate elements of a vector by 1 and ¯1. Thus, if ALTER←1¯1 1¯1 1, then +/VECTOR×ALTER and -/VECTOR are equal, as demonstrated below:

```
      VECTOR
2  4  6  8  10
      ALTER←1 ¯1 1 ¯1 1
      VECTOR×ALTER
2 ¯4  6 ¯8  10
      +/VECTOR×ALTER
6
      -/VECTOR
6
```

An alternating product can be obtained by ÷/VECTOR. For example,

```
      VECTOR
2  4  6  8  10
      ALTER
1 ¯1  1 ¯1  1
      ×/VECTOR*ALTER
3.75
      ÷/VECTOR
3.75
```

When the reduction operator is applied to any scalar or vector argument, the result is a scalar value. The value resulting from a scalar or unit array argument is the argument itself. The effect of applying the reduction operator to multi-dimensional arrays is discussed under the axis operator on page 3-20 .

If the reduction operator and a primitive scalar dyadic function are applied to an empty array, the identity element of the function becomes the result if an identity element exists for that function. If an identity element does not exist for the function, a domain error results. Note that an empty array may be of type character or numeric and identity elements differ depending on these types. For example, the identity elements for the times function (×) is 1 for numbers, and none exists for the nand function.

```
      E←0⍴0
      ×/E
1
      ⍲/E
DOMAIN ERROR
       ⍲/E
        ↑
      +/E
0

      E←0⍴''
      ×/E
DOMAIN ERROR
       ×/E
        ↑
      ⍲/E
DOMAIN ERROR
       ⍲/E
        ↑
      +/E
DOMAIN ERROR
       +/E
        ↑
```

The identity elements (or the domain error resulting when no identity element exists) of all functions when they are combined with the reduction operator and applied to an empty vector are shown in table 3-3.

SCAN OPERATOR

The scan operator (\) applies the dyadic primitive scalar function which precedes it to the argument. The scan operator performs a cumulative reduction over arrays. The result of this operator is an array of the same shape as the operand, in which the nth element corresponds to the result of the reduction over the first n elements.

```
      VECTOR
2  4  6  8  10
      +/VECTOR
30
      +\VECTOR
2  6  12  20  30
```

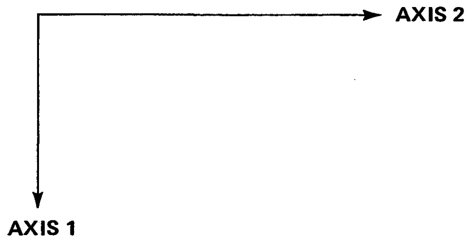
Other examples of the scan operator are:

```
          VECTOR
2   4   6   8   10
          ×\VECTOR
2   8   48   384   3840
          VEC←1  1  1  0  0  0  1
          ∧\VEC
1   1   1   0   0   0   0
          ∨\VEC
1   1   1   1   1   1   1
          ⍲\VEC
1   0   1   1   0   0   0
```

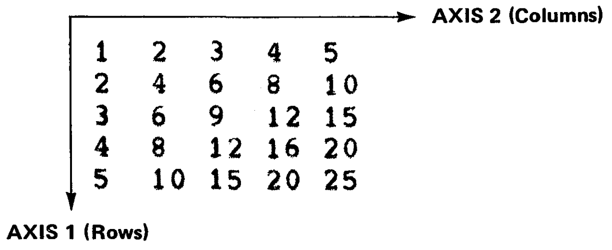
The results obtained when the scan operator is applied to arrays other than vectors is discussed under the axis operator.

AXIS OPERATOR

The discussion of the reduction and scan operators described what happens when those operators are coupled with a dyadic primitive scalar function and applied to a vector. The reduction operator, however, also can be applied to arrays, which can be thought of as collections of vectors. For example, consider an array that has two axes:



The columns extend along axis 1 and rows extend along axis 2.



Reduction of an array can be defined as the vector of results produced by reduction of each of the column vectors or the row vectors.

The axis operator is signified by brackets [ ] enclosing an expression. The expression, when evaluated, yields the index of the axis. For example,

```
      ARRAY←4 6ρ1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
      ARRAY
  1   2   3   4   5   6
  7   8   9  10  11  12
 13  14  15  16  17  18
 19  20  21  22  23  24
      +/[1]ARRAY
40  44  48  52  56  60
      +/[2]ARRAY
21  57  93  129
      +\[1]ARRAY
  1   2   3   4   5   6
  8  10  12  14  16  18
 21  24  27  30  33  36
 40  44  48  52  56  60
      +\[2]ARRAY
  1   3   6  10   15   21
  7  15  24  34   45   57
 13  27  42  58   75   93
 19  39  60  82  105  129
```

Note that the scan operator produces a result whose shape is the same as that of the argument while the reduction operator produces a result whose shape is the shape of the argument with the reduction axis removed. That is, the shape vector of the result has one fewer elements.

If no axis operator is included with reduction and scan, these operators apply along the last axis as follows:

```
      ARRAY
  1   2   3   4   5   6
  7   8   9  10  11  12
 13  14  15  16  17  18
 19  20  21  22  23  24
      +/[]ARRAY
21  57  93  129
      +\[]ARRAY
  1   3   6  10   15   21
  7  15  24  34   45   57
 13  27  42  58   75   93
 19  39  60  82  105  129
```

The symbols ⌿ and ⍀ may also signify reduction and scan (also compression and expansion), respectively; and, in the absence of the axis operator, these operators apply along the first axis, as follows:

```
      +⌿ARRAY
40  44  48  52  56  60
      +⍀ARRAY
  1   2   3   4   5   6
  8  10  12  14  16  18
 21  24  27  30  33  36
 40  44  48  52  56  60
```

If an axis operator is used with ⌿ or ⍀, it signifies the nth from last axis, as opposed to nth from first axis with / or \.

See the discussions of the mixed functions reverse, rotate, compress, and expand for additional applications of the axis operator.

INNER PRODUCT OPERATOR

Sets of data can be arranged into vectors of the same shape to perform numerous useful computations. For example, if vector A represents a list of parts and B represents a list of prices, and A and B are the same shape, then the expression +/A×B would produce the total cost of inventory.

Expressions of the same form using other functions also are useful. For example,

```
      X
2   4   6   8   10
      Y
17   4   3.95   8.96   10
      ∧/X=Y ◄──────────────── Comparison of X and Y
0
      +/X=Y ◄──────────────── Number of agreements between X and Y
2
```

The inner product operator (.) applies the two functions that enclose it to a left and a right argument to produce functions equivalent to the examples shown above.

Thus, Afn1.fn2B is equivalent to fn1/Afn2B. For example, for vectors/scalars:

```
      A
2   4   6   8   10
      B
17   4   3.95   8.96   10
      A+.×B
245.38
      +/A×B
245.38
      A×.*B
4.91058931E28
      ×/A*B
4.91058931E28
```

When applied to arrays, the inner product operator extends to the last axis of the left argument and the first axis of the right argument. The lengths of the two axes must agree. The axes operated on by the inner product operator are deleted and the shape of the result is the catenation of the remaining shapes of the operands, as for example,

```
      VEC
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
      A←3 5ρVEC
      B←5 4ρVEC
      A
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
      B
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
 17  18  19  20
      A+.×B
 175  190  205  220
 400  440  480  520
 625  690  755  820
      B←5ρVEC
      B
1  2  3  4  5
      A
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
      A+.×B
55  130  205
      A←8ρVEC
      B←8ρVEC
      A
1  2  3  4  5  6  7  8
      B
1  2  3  4  5  6  7  8
      A+.×B
204
```

The inner product A+.×B is also known as the matrix product. Examples are

```
      VEC
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
      A←4ρVEC
      A
1  2  3  4
      BOOL←1 1 1 1 1 0 0 0 1 0 1 0 0 0 0 0
      B←4 4ρBOOL
      B
  1  1  1  1
  1  0  0  0
  1  0  1  0
  0  0  0  0
      A+.×B
6  1  4  1
      B+.×A
10  1  4  0
      B+.×B
  3  1  2  1
  1  1  1  1
  2  1  2  1
  0  0  0  0
```

Examples of other inner products:

```
      A
1  2  3  4
      B
  1  1  1  1
  1  0  0  0
  1  0  1  0
  0  0  0  0
      A×.*B
6  1  3  1
      B∧.=B
  0  0  0  0
  0  1  0  1
  0  0  1  0
  0  0  0  0
      B∧.=1 1 0 0
0  0  0  0
      B-.×B
  1  1  2  1
  1  1  1  1
  2  1  2  1
  0  0  0  0
```

The preceding examples show that either argument can be of any rank, so long as the rank of the result is ≤63, and the last dimension of the left argument is compatible with the first dimension of the right argument, that is,

```
      ((¯1↑⍴A)=1↑⍴B)∨1∊(¯1↑⍴A),1↑B
```

Thus, A+.×1 is equivalent to +/A and 1+.×A is equivalent to +/A. For example,

```
      A←2 4 6 8 10
      +/A
30
      A+.×1
30
      +⌿A
30
      1+.×A
30
```

OUTER PRODUCT OPERATOR

The outer product operator is signified by the symbols °. and precedes the function to which it is applied. The outer product operator can be applied to any dyadic primitive scalar function. When the outer product is applied to a function, that function is evaluated for each element of the left argument paired with each element of the right argument. For example,

```
          A
 2   4   6
          B
 2   4   6   8   10
        A∘.+B
   4    6    8   10   12
   6    8   10   12   14
   8   10   12   14   16
        A∘.×B
    4    8   12   16   20
    8   16   24   32   40
   12   24   36   48   60
        A∘.*B
    4      16       64        256        1024
   16     256     4096      65536     1048576
   36    1296    46656    1679616    60466176
        A∘.<B
   0   1   1   1   1
   0   0   1   1   1
   0   0   0   1   1
        A∘.>B
   0   0   0   0   0
   1   0   0   0   0
   1   1   0   0   0
```

These examples show that the shape of the result of X∘.fnY is equal to (ρX),ρY. The expression (ρX),ρY produces the shape for any arguments X and Y.

MIXED FUNCTIONS

There are five classes of mixed functions, grouped according to whether they are concerned with:

* The structure of arrays.
* Selection from arrays.
* The generation of selection information.
* Numerical calculations.
* Transformations of data such as that between numbers and characters.

These five groups of mixed functions are listed in tables 3-5 through 3-9. Included in each table are the names of the mixed functions, the symbols used to denote the functions, a definition or example of each function, and restrictions on the ranks of arguments that may be used with each mixed function.

Table 3-5. Structural Mixed Functions

NAME	SYMBOL	FORM	DEFINITION
Ravel	,	,A	Produces vector whose elements are the elements of the right argument in row major order.
Shape	ρ	ρA	Produces vector whose elements are the dimensions of A.
Reshape	ρ	AρB	Reshapes the ravel of right argument to shape specified by left argument.
Reversal	⌽ or ⊖	⌽ A or ⊖ A	Reverses elements in the right argument. When ⌽ is used, elements along the last coordinate are reversed; with ⊖, elements along the first coordinate are reversed.
Rotate	⌽ or ⊖	A⌽B or A⊖B	Causes elements of the right argument to be rotated. When ⌽ is used, elements along last coordinate are rotated; with ⊖ , elements along first coordinate are rotated.
Catenate	,[]	,[A]	Joins two arrays along an existing axis.
Laminate	,[]	A,[B]	Joins two arrays along a *new* axis.
Transpose	⍉	⍉ A or A⍉B	Reverses the order of (*transposes*) the axes of an array. If used dyadically, as A⍉B, arranges axes of B to conform to argument A.

Table 3-6. Selection Mixed Functions

NAME	SPECIAL CHARACTER	FORM	DEFINITION
Take	↑	N↑A	Takes N elements from A. If N is positive, *first* N elements are taken; if negative, *last* N elements taken.
Drop	↓	N↓A	Drops N elements from A. If N is positive, *first* N elements are dropped; if negative, *last* N elements dropped.
Compress	/	N/A	Selects elements from an array as determined by boolean argument N. For each 1 in N, the corresponding element in A is selected; for each 0, it is ignored.
Expand	\	N\A	Fills array with spaces (if alphabetic) or zeros (if numeric) depending on boolean argument N.
Indexing	[]	A[]	Selects elements from A depending on expression enclosed in brackets. If A is 2 4 6 8 10, A[3] selects 6 if *1-origin* indexing is in effect.

Table 3-7. Selector Generator Mixed Functions

NAME	SYMBOL	FORM	DEFINITION
Index generator	⍳	⍳A	Produces first A integers in order, beginning with index origin in effect.
Index of	⍳	A ⍳ B	Produces the index of first occurrence of B in A.
Membership	∊	A∊B	Determines if each element of A is a member of B.
Grade up	⍋	⍋A	Sorts the elements of a vector in ascending order, returning indices.
Grade down	⍒	⍒A	Sorts the elements of a vector in descending order, returning indices.
Deal	?	A?B	Selects A random integers without replacement from ⍳B.

Table 3-8. Numerical Mixed Functions

NAME	SYMBOL	FORM	DEFINITION
Matrix inverse	⌹	⌹A	Produces the inverse of a non-singular matrix. Columns of A must be linearly independent.
Matrix divide	⌹	A⌹B	Produces a result equal to (⌹ B) + . x A.
Decode	⊥	A⊥B	Computes the sum of all the elements of B raised to a power specified by the base value of A. If A is 2 and B is 1 2 3 4 5, then A ⊥ B is 101.
Encode	⊤	A⊤B	Converts the value of A into its representation in the number system specified by the base value of B.

Table 3-9. Data Transformation Mixed Functions

NAME	SYMBOL	FORM	DEFINITION
Execute	⍎	⍎A	Executes the *character expression* A.
Format, monadic	⍕	⍕A	Monadic form A produces character representation of A to current default printing precision. For example, A←○1 'PI IS EQUAL TO ',⍕A produces PI IS EQUAL TO 3.14159265
Format, dyadic	⍕	A⍕B	Produces result based on data B displayed in accordance with control argument A. For example, 4 2⍕3.14159 3.14
Quad output	⎕	⎕←A	Generates carriage return/linefeed when displaying A.
Quote quad output	⍞	⍞←A	Outputs A with no carriage return/ linefeed.
Quote quad input	⍞	A←⍞	Reads a line of characters typed in by user and creates a character vector result.
Quad input	⎕	A←⎕	Evaluates a line of input from the terminals.

Figure 3-2 contains a list of those mixed functions for which scalar and vector arguments may be substituted.

```
1. A scalar may be used in place of a one-element vector:

   a. as left argument of

      reshape           2ρ5      ↔     (,2)ρ5
      take              4↑ 6     ↔     (,4)↑ 6
      drop              ¯4↓ 6          (,4)↓ 6
      expand            1\,6           (,2)\,6
      transpose         1ϕ,4           (,1)ϕ,4
      format            6⍕4.5          (,6)⍕4.5    ↔     0 6⍕4.5
      rotate            2ϕA            (,2)ϕA

   b. as right argument of

      execute           ⍎'X'           ⍎, 'X'

2. A scalar is extended to conform to a vector:

   a. as left argument of

      compress          1/ι4  ↔        1 1 1 1/ι4
      rotate            1ϕ2 2 ρι4   ↔    1 1 ϕ 2 2ρι4

   b. as right argument of

      compress          1 0 1 / 2   ↔  1 0 1 / 2 2 2
      expand            1 0 1 \ 2   ↔  1 0 1 \ 2 2

3. A unit array is permitted in place of a scaler:

   a. as left argument of

      deal              (,4)?5      ↔    4?5

   b. as right argument of

      index generator   ι,6         ↔    ι6
      deal              2?,6        ↔    2?6
```

Figure 3-2. Scalar-Vector Substitutions for Mixed Functions

STRUCTURAL FUNCTIONS

The structural functions consist of:

* Ravel (,)

* Shape (ρ)

* Reshape (⍴)

* Reverse (⌽ or ⊖)

* Rotate (⌽ or ⊖)

* Catenate (,[ ])

* Laminate (,[ ])

* Transpose (⍉)

For monadic structure functions, the argument may be of any type, numeric or character. For dyadic structure functions, the right argument may be of any type, but the left argument (which serves as an <u>index</u> or other <u>selection generator</u>) must be numeric integer.

SHAPE FUNCTION. The monadic shape function (⍴) applied to an array argument, yields the shape of the array as a vector whose elements are the dimensions of the array. For example,

```
          ARRAY
      1   2
      3   4
      5   6
      7   8
      9  10
          ⍴ARRAY
  5   2
          ⍴⍴ARRAY
  2
          ⍴⍴⍴ARRAY
  1
```

The result produced by ⍴ARRAY contains one component for each axis of ARRAY. For example, 5 2 (above) signifies that ARRAY is a matrix of five rows and two columns. Thus, the expression ⍴⍴ARRAY produces the <u>rank</u> of ARRAY, and ⍴⍴⍴ARRAY produces the <u>shape</u> of the array <u>resulting</u> from the expression ⍴⍴ARRAY. (Note that ⍴⍴⍴ARRAY is always 1.) Figure 3-3 illustrates arrays from rank 0 (scalar) up to rank 6. Note that the function ⍴ applied to a scalar yields the <u>empty vector</u>. Note also that a one-dimensional array is rank 1, two-dimensional is rank 2, and so forth.

RAVEL FUNCTION. The monadic ravel function (,) applied to an <u>array</u>, produces a <u>vector</u> whose elements are the elements of the array <u>in row</u> major order. For example,

```
          ARRAY
       1   2   3   4
       5   6   7   8
       9  10  11  12
      13  14  15  16
          VECTOR←,ARRAY
          VECTOR
  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
```

If the ravel function is applied to a vector argument, the result is equivalent to the argument itself. If applied to a scalar argument, the ravel function produces a vector of length 1.

```
      A0←1
      ρA0
                          RANK 0
      ρρA0
0
      A1←4ρVEC
      ρA1
4                             RANK 1
      ρρA1
1
      A2←4 4ρVEC
      ρA2
4  4                            RANK 2
      ρρA2
2
      A3←4 4 4ρVEC
      ρA3
4  4  4                           RANK 3
      ρρA3
3
      A4←4 4 4 4ρVEC
      ρA4
4  4  4  4                          RANK 4
      ρρA4
4
      A5←4 4 4 4 4ρVEC
      ρA5
4  4  4  4  4                         RANK 5
      ρρA5
5
      A6←4 4 4 4 4 4ρVEC
      ρA6
4  4  4  4  4  4                        RANK 6
      ρρA6
6
```

Figure 3-3. Rank of Arrays

RESHAPE FUNCTION. The dyadic reshape function (ρ) reshapes the ravel of its right argument to the shape specified by its left argument. For example,

```
      A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
      4 5ρA
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
```

For the reshape expression LρR, if the total number of elements in the right argument R is equal to the total number of elements required by the left argument L (as above), the ravel of LρR is equal to the ravel of R (the elements are equal). If L specifies a value that requires <u>less</u> elements than are contained in R, only the <u>first ×/L elements</u> of R are used; if L requires <u>more</u> elements than are contained in R, the elements of R are <u>repeated cyclically</u>. For example,

```
      2 3ρA
1 2 3
4 5 6
      5 6ρA
 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20  1  2  3  4
 5  6  7  8  9 10
```

Any one or more of the axes of an array <u>may</u> have zero length, thus, 0ρA, 0 3ρA, and 0 0 0ρA are all valid. Such an array is called an <u>empty array</u>. If A is a numeric <u>empty vector</u>, then AρB is a <u>scalar</u> containing the first element of ravel B.

REVERSAL FUNCTION. The monadic reversal function is denoted by the symbols ⌽ or ⊖ and is used to reverse the elements along a particular axis of the argument. For example,

```
      A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
      ⌽A
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

When ⌽A is used, the reversal occurs along the <u>last axis</u> (the <u>columns</u> are reversed) of the array. For example,

```
      ARRAY
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
      ⌽ARRAY
 4  3  2  1
 8  7  6  5
12 11 10  9
16 15 14 13
```

When the ⊖ symbol is specified, the reversal occurs along the <u>first axis</u> (the <u>rows</u> are reversed), as for example,

```
          ⊖ARRAY
     13  14  15  16
      9  10  11  12
      5   6   7   8
      1   2   3   4
```

The auxiliary axis operator can be applied to the reversal function to specify a particular axis for the reversal. For example,

```
          ⌽[1]ARRAY
     13  14  15  16
      9  10  11  12
      5   6   7   8
      1   2   3   4
          ⌽[2]ARRAY
      4   3   2   1
      8   7   6   5
     12  11  10   9
     16  15  14  13
```

The previous example shows that ⌽A is equivalent to ⌽[⍴⍴A]A or ⌽[1]A, and ⊖A is equivalent to ⊖[1]A.

ROTATE FUNCTION. The dyadic rotate function is denoted by the symbols ⌽ or ⊖ and rotates elements in the right argument by amounts specified in the left argument.

If S is a scalar or unit and V is a vector, then S⌽V results in a cyclic rotation of V, as follows:

For 1-origin indexing, `S⌽V = V[1+(⍴V)|¯1+S+⍳⍴V]`

For 0-origin indexing, `S⌽V = V[(⍴V)|S+⍳⍴V]`

General expression: `S⌽V = V[⎕IO+(⍴V)|(-⎕IO)+S⍳⍴V]`

The axis operator can be used with the rotate function to specify the axis along which the rotation is to be performed. The form is

    S⌽[n]V

For general arrays, the vector along the nth axis of V is rotated as signified by the corresponding element of S, and the shape of S must equal the remaining dimensions of V. For example,

```
      VECTOR
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
      X←4 4ρVECTOR
      X
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16
      1 2 3 4⌽[1]X
 5  10  15   4
 9  14   3   8
13   2   7  12
 1   6  11  16
      1 2 3 4⌽[2]X
 2   3   4   1
 7   8   5   6
12   9  10  11
13  14  15  16
```

The symbol ⊖ can be used to signify rotation along the first axis of an array and therefore A⊖B is equivalent to A⊖[1]B, as follows:

```
      VEC←ι16◇VEC
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
      B←4 5ρVEC◇B
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15
16   1   2   3   4
      (ι5)⊖B
 6  12   2   4  10
11   1   3   9  15
16   2   8  14   4
 1   7  13   3   5
      (ι4)⊖[1]B
 2   3   4   5   1
 8   9  10   6   7
14  15  11  12  13
 4  16   1   2   3
```

CATENATE FUNCTION. The dyadic catenate function (,) is used to join two arrays along an existing coordinate. The number of elements in the resulting array is equal to the total number of elements in the two arguments. For example,

```
      A
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15
16  17  18  19  20
      B
21  22  23  24  25
26  27  28  29  30
31  32  33  34  35
36  37  38  39  40
      A,B
 1   2   3   4   5  21  22  23  24  25
 6   7   8   9  10  26  27  28  29  30
11  12  13  14  15  31  32  33  34  35
16  17  18  19  20  36  37  38  39  40
```

A numeric vector cannot be catenated with a character vector.

The axis operator can be applied to the catenate function to signify the axis along which the arguments are to be catenated. For example,

```
      A
2 4 6 8 10 12 14 16 18
      B←3 3⍴A
      B
 2   4   6
 8  10  12
14  16  18
      B,[1]B
 2   4   6
 8  10  12
14  16  18
 2   4   6
 8  10  12
14  16  18
      B,[2]B
 2   4   6   2   4   6
 8  10  12   8  10  12
14  16  18  14  16  18
```

Arrays of different shapes can be catenated along an axis n if they have the same number of elements along that axis and they differ in rank by 1. For example,

```
      (2 3⍴⍳100),2 10⍴⌽⍳(100)
1 2 3 100 99 98 97 96 95 94 93 92 91
4 5 6  90 89 88 87 86 85 84 83 82 81
```

Some other examples:

```
      ARR1
2  4  6
      ARR2
   2   4   6   8
  10  12  14  16
  18   2   4   6
      ARR3
   2   4   6   8  10
  12  14  16  18   2
   4   6   8  10  12
  14  16  18   2   4

   6   8  10  12  14
  16  18   2   4   6
   8  10  12  14  16
  18   2   4   6   8

  10  12  14  16  18
   2   4   6   8  10
  12  14  16  18   2
   4   6   8  10  12
      ARR1,ARR2
  2   2   4   6   8
  4  10  12  14  16
  6  18   2   4   6
      ARR2,ARR3
   2   2   4   6   8  10
   4  12  14  16  18   2
   6   4   6   8  10  12
   8  14  16  18   2   4

  10   6   8  10  12  14
  12  16  18   2   4   6
  14   8  10  12  14  16
  16  18   2   4   6   8

  18  10  12  14  16  18
   2   2   4   6   8  10
   4  12  14  16  18   2
   6   4   6   8  10  12
```

A scalar or unit argument is repeated along the appropriate axis when used as an argument in the catenate function. For example,

```
        A←1
        ARR2
  2     4     6     8
 10    12    14    16
 18     2     4     6
        ARR2,[1]A
  2     4     6     8
 10    12    14    16
 18     2     4     6
  1     1     1     1
        ARR2,[2]A
  2     4     6     8    1
 10    12    14    16    1
 18     2     4     6    1
```

LAMINATE FUNCTION. The dyadic laminate function is denoted by a comma followed by the lamination coordinates enclosed in brackets ,[ ]. The lamination coordinate is a non-integral index number signifying a new coordinate between existing coordinates along which the lamination is to occur.

The laminate function joins two arrays of identical rank and shape along a new axis; this new axis is indicated by the index number. For example, if the new axis is to be inserted between existing axes 1 and 2, the index number must be between 1 and 2; for laminating between existing axes 2 and 3, the index number must be between 2 and 3, and so forth. If the new axis is to be inserted before the existing first axis, the index number must be between 0 and 1. (If 0-origin indexing is in effect, subtract 1 from the above index numbers.) If the new axis is to be added after the existing last axis, the fractional index number must exceed the last axis number by a fractional amount between 0 and 1.

Examples of lamination are:

```
      A
ABCD
EFGH
IJKL
MNOP
      B
1234
5678
9012
3456
      C←A,[.5]B◇C
ABCD
EFGH
IJKL
MNOP

1234
5678
9012
3456
      D←A,[1.5]B◇D
ABCD
1234

EFGH
5678

IJKL
9012

MNOP
3456
```

```
      E←A,[2.5]B◇E
A1
B2
C3
D4

E5
F6
G7
H8

I9
J0
K1
L2

M3
N4
O5
P6
      ρC
2 4 4
      ρD
4 2 4
      ρE
4 4 2
```

The shapes of the resulting arrays in the above examples are 2 4 4, 4 2 4, and 4 4 2. Note that the resulting array in each case is one rank greater than the rank of A and B, and has the same shape except for the insertion of the new axis. The 2 in 2 4 4, 4 2 4, and 4 4 2 shows where the new axis was inserted and also denotes the length of the new axis.

When used with the laminate function, a scalar or unit argument is extended as necessary. For example,

```
           A←3 3ρ'ABCDEFGHI'
           B←'1'
           A
     ABC
     DEF
     GHI
           B
     1
           A,[2.5]B
     A1
     B1
     C1

     D1
     E1
     F1

     G1
     H1
     I1
```

TRANSPOSE FUNCTION. The dyadic transpose function is signified by the character ⍉ and reverses the order of (transposes) the axes of A. An element [I,J] in the result is equal to [J,I] in the argument. Thus, [1;2] in the argument is equal to [2;1] in the result. For example,

```
           A
     ABC
     DEF
     GHI
           2 1⍉A
     ADG
     BEH
     CFI
           RESULT←2 1⍉A⋄RESULT
     ADG
     BEH
     CFI
           A[1;2]
     B
           RESULT[2;1]
     B
```

Monadic Transpose Examples

```
           A
   1   2   3   4
   5   6   7   8
   9  10  11  12
           ρA
 3  4
           A←⍉A
           A
   1   5   9
   2   6  10
   3   7  11
   4   8  12

           B
 HOWNOWOLDCOW
           B←4 3ρB
           B
 HOW
 NOW
 OLD
 COW
           ⍉B
 HNOC
 OOLO
 WWDW
```

The dyadic expression 2 1 ⍉A reverses the order of the axes of A. For example,

```
       A
ABC
DEF
GHI
       ⍉A
ADG
BEH
CFI
       2 1⍉A
ADG
BEH
CFI
```

SELECTION FUNCTIONS

The selection functions include:

* Take (↑)
* Drop (↓)
* Compress (/)
* Expand (\)
* Indexing ([ ])

The arguments whose elements are being selected may be any type of array, while the other argument, which specifies the selection, must be numeric <u>integer</u> or <u>bit</u>. For the expand and compress functions, the numeric values must be boolean.

TAKE FUNCTION. The take function (↑) selects elements from an array. The elements selected are dependent on the numeric left argument. If the values of N are <u>positive</u>, the <u>first N</u> elements are selected; if the values of N are <u>negative</u>, the <u>last N</u> elements are selected. If N is greater than the number of elements in the array, the result is filled with zeros if the array is numeric or spaces if the array is alphabetic.

Examples of the take function being applied to a vector are:

```
          A←2 4 6 8 10
          2↑A
2   4
          4↑A
2   4   6   8
          6↑A
2   4   6   8   10   0
          8↑A
2   4   6   8   10   0   0   0
          ¯8↑A
0   0   0   2   4   6   8   10
          B←'12345'
          2↑B
12
          4↑B
1234
          6↑B
12345
          'A',8↑B
A12345
          'A',¯8↑B
A    12345
```

Note that the zeros (or spaces) are added on the right if the left argument is positive and on the left if the left argument is negative.

If the left argument is a vector, then the expression V↑A is valid only if V has one element for each axis in array A. For example, if A is unit or if A has two axes, then V can have only two elements.

The rank of the result of the take function is the same as the rank of the right argument.

DROP FUNCTION. The drop function (↓) is the opposite of the take function, and removes specified elements from an array. If the number of elements dropped from an array equals or exceeds the number of elements along the axis, the result has zero length for that axis.

Examples of the drop function are:

```
          A
2   4   6   8   10
          2↓A
6   8   10
          4↓A
10
          ¯2↓A
2   4   6
          ¯4↓A
2
```

The rank of the result of the drop function is the same as the rank of the right argument.

COMPRESS FUNCTION. The compress function (/) selects elements from an array as determined by a boolean argument. For each 1 in the boolean argument, the corresponding portion in the array is selected; for each zero in the boolean argument, the corresponding portion in the array is not selected. For example, a boolean argument 1 0 1 0 1 selects the first, third, and fifth elements of an array as follows:

```
          A
2   4   6   8   10
          1 0 1 0 1/A
2   6   10
```

The dimensions of the arguments must agree, except that scalar arguments are extended. Thus, 1/A equals A and 0/A equals an <u>empty vector</u>, as shown below:

```
          A
2   4   6   8   10
          1/A
2   4   6   8   10
          0/A

          ρ0/A
0
```

The axis operator can be used with the compress function. For an expression A/[n]B, the shapes of A and B conform if ⍴A equals (⍴B)[n], or A is a unit. An example,

```
      A←4 4⍴⍳16

      A
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16
      1 0 1 0/[1]A
 1   2   3   4
 9  10  11  12
      1 0 1 0/[2]A
 1   3
 5   7
 9  11
13  15
      B←4 4⍴'ABCDEFGHIJKLMNOP'⋄B
ABCD
EFGH
IJKL
MNOP
      1 0 1 0/[1]B
ABCD
IJKL
      1 0 1 0/[2]B
AC
EG
IK
MO
```

The ⌿ symbol can be used to denote compression along the first axis, as follows,

```
      A
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16
      1 0 1 0⌿A
 1   2   3   4
 9  10  11  12
```

The rank of the result of the compress function equals the rank of the right argument, and ⍴result, along the axis of compression equals +/left argument.

EXPAND FUNCTION. The expand function (\) expands an array, filling identity elements as determined by a boolean argument. If the array is numeric, the identity elements are zeros where the array is expanded; if the array is alphabetic, the identity elements are spaces.

Examples of the expand function are:

```
      X←'THEQUICKBROWNFOX'
      Y←1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1
      Y\X
THE QUICK BROWN FOX
      C←5 4⍴X⋄C
THEQ
UICK
BROW
NFOX
THEQ
      1 0 1 0 1 1\C
T H EQ
U I CK
B R OW
N F OX
T H EQ
```

The axis operator can be used with the expand function. For example,

```
     A
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
     1 1 0 1 0 1\[1]A
  1   2   3   4
  5   6   7   8
  0   0   0   0
  9  10  11  12
  0   0   0   0
 13  14  15  16
     1 1 0 1 0 1\[2]A
  1   2  0   3  0   4
  5   6  0   7  0   8
  9  10  0  11  0  12
 13  14  0  15  0  16
```

The ⍀ symbol can be used to denote expansion along the first axis as follows,

```
        A
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16
        1 0 1 0 1 1⍀A
 1   2   3   4
 0   0   0   0
 5   6   7   8
 0   0   0   0
 9  10  11  12
13  14  15  16
```

The rank of the result of the expand function is equal to the rank of the right argument, and the length of the result along axis of expansion is ⍴left argument.

INDEXING FUNCTION. The indexing function is denoted by brackets and may be 1-origin or 0-origin as specified by ⎕IO. For 1-origin indexing, the function A[I] indicates the Ith element of A; for 0-origin, A[I] indicates the I+1 element of A. For example,

```
        A←1 2 3 4 5 6 7 8 9 0
        ⎕IO←1
        A[3]
3
        A[6]
6
        ⎕IO←0
        A[3]
4
        A[6]
7
```

If a vector V is used within the brackets, such as A[V], elements are selected from A as indicated by the elements of V. For example,

```
        V←1 3 5 7 9
        'ABCDEFGHIJKLMNOP'[V]
ACEGI
```

If the value specifies an element outside the range of A, an error message results. In general, the shape of A[I] is the shape of I. Thus, if I is scalar, the result of A[I] is scalar; and if I is an array of any rank, then A[I] is an array of that rank. For example,

```
      A←'ABCDEFGHIJKLMNOP'
      A[4]
D
      V←3 5⍴1 2 3 4 2 3 4 1 3 4 1 2 4 3 2 1
      A[V]
ABCDB
CDACD
ABDCB
```

If A is a matrix, it must be indexed in the form [R;C]. The first index, R, signifies the row (or rows) and the second index, C, signifies the column (or columns). Thus, A[2;1] selects the element from the second row, column 1. If either index is a vector, the rows or columns specified by all values of the vector are selected. For example,

```
      A
ABCD
EFGH
IJKL
MNOP
ABCD
      A←4 4⍴A
      A
ABCD
EFGH
IJKL
MNOP
      A[2 3;1]
EI
      A[4 3 2;2 3 4]
NOP
JKL
FGH
```

In general, the shape of the result of A[R;C] is (⍴R),⍴C. Thus, if R and C are both vectors, the result is a matrix; if R and C are both matrices (rank 2), the result is an array of rank 4. Similarly, if R and C are both scalars, the result is scalar; if R is vector and C scalar, or vice versa, the result is a vector.

Examples:

```
      A
ABCD
EFGH
IJKL
MNOP
      A[2;3] ←———————— Both scalars
G
      A[2;3 4 2] ←———— Scalar and a vector
GHF
      A[2 4;2 3 4] ←—— Both vectors
FGH
NOP
      R←2 2ρ2 3 1 4
      C←2 2ρ4 1 3 2
      R
 2  3
 1  4
      C
 4  1
 3  2
      A[R;C] ←———————— Both matrices
HE
GF

LI
KJ


DA
CB

PM
ON
```

Omitting one of the members of the index denotes all rows or columns, depending on which is omitted. Thus, A[;C] specifies all rows (the <u>row</u> index is omitted), and A[R;] specifies all columns (the <u>column</u> index is omitted). For example,

```
      A
ABCD
EFGH
IJKL
MNOP
      A[;4]
DHLP
      A[4;]
MNOP
```

The left-hand part of an assignment expression may be an indexed expression as long as it is of the correct shape and size. For example, to change elements 3 and 10 of array A to the values 4 and 2, respectively,

```
              A←'ABCDEFGHIJKLMNOP'
              A[3,10]←'42'
              A
        AB4DEFGHI2KLMNOP
```

SELECTOR GENERATOR FUNCTIONS

The selector generator functions consist of:

* Index generator (⍳)
* Index of (⍳)
* Membership (∊)
* Grade up (⍋)
* Grade down (⍒)
* Deal (?)

Each of these selector generator functions produce integer results which are useful in a variety of applications as discussed for each function following.

INDEX GENERATOR. The index generator is signified by the symbol ⍳ and can have as an argument a non-negative scalar integer N to produce a vector containing N integer values in order, beginning with the index origin in affect. For example, ⍳6 produces the vector 1 2 3 4 5 6 if the index origin is 1, and 0 1 2 3 4 5 if the index origin is 0. If zero is used as the argument, an empty vector is produced.

INDEX OF. When the ⍳ function is used dyadically with a vector and a scalar argument in the form VECTOR⍳SCALAR, the index generator function results in the index of the first occurrence of each element of VECTOR in SCALAR.

If the scalar is different from all elements of the vector, a value one greater than the index of the last element of VECTOR is returned, as for example,

```
              VECTOR←'ABCDEF'
              SCALAR←'J'
              VECTOR⍳SCALAR
        7
```

Note that the result of VECTOR⍳SCALAR is origin dependent.

MEMBERSHIP FUNCTION. The membership function is denoted by the symbol ∊. If A is an array, the expression A∊B produces an array with the same shape as A but consisting of boolean values only (B may be of any shape). The elements of the result have a value of 1 if the corresponding element of A also exists in B, and a value of 0 if the corresponding element of A does not exist in B. For example,

```
         A
ABCD EFGH IJKL MNOP QRST UVWX
         B
BAD NEWS
         A∊B
1  1  0  1  1  1  0  0  0  1  0  0  0  0  1  0  1  0  0  1  0  0  1  0  1  0  0
      1  0
```

The arguments of the membership function do not have to be of the same shape or rank. See below.

```
         A
ABCD EFGH IJKL MNOP QRST UVWX
         B
BAD NEWS
         C←5 6⍴A◇C
ABCD E
FGH IJ
KL MNO
P QRST
 UVWXA
         C∊B
  1  1  0  1  1  1
  0  0  0  1  0  0
  0  0  1  0  1  0
  0  1  0  0  1  0
  1  0  0  1  0  1
         D←2 4⍴B◇D
BAD
NEWS
         C∊D
  1  1  0  1  1  1
  0  0  0  1  0  0
  0  0  1  0  1  0
  0  1  0  0  1  0
  1  0  0  1  0  1
```

GRADE FUNCTIONS. The two grade functions, grade up (⍋) and grade down (⍒), apply only to numeric vectors and are sorting functions. The grade up function sorts the elements of a vector in ascending order and produces a vector of the same length as the argument, containing the indices of the sorted elements of the argument. For example, if A←10 6 1 3 2, ⍋A produces 3 5 4 2 1, in which the index of the lowest value of A is first, the index of the next lowest value is second, and so forth. In order to access the elements of A in ascending order, rather than the <u>indices</u> of the elements, the expression A[⍋A] is used. For example,

```
            A←10 5 3 2 1
            ⍋A
      5  4  3  2  1
            A[⍋A]
      1  2  3  5  10
```

If two or more elements of a vector are the same, the order is determined by their positions in the vector. For example,

```
            A←6 6 6 4 3 6
            ⍋A
      5  4  1  2  3  6
```

The grade down function (⍒) produces a vector of indices of the elements of a vector sorted in <u>descending</u> order. Equal elements are sorted according to their position in the vector just as they are for the grade up function.

Examples of the grade down function are:

```
            A←3 10 6 1 2
            ⍒A
      2  3  1  5  4
            A[⍒A]
      10  6  3  2  1
            A←3 10 3 3 6
            ⍒A
      2  5  1  3  4
```

Note that the results of grade up and grade down are <u>origin dependent</u>.

DEAL FUNCTION. The deal function (?) selects pseudo-random integer selections from the vector of integer values produced by the index generator function (⍳). No two of the selections are the same. Both A and B are limited to scalar or unit array arguments. Each selection from the ⍳B set of integers is in accordance with the method described for the roll function. That is, A?B produces A integers selected in random fashion without replacement from the set of ⍳B. A?B is <u>origin dependent</u>.

Examples of the deal function are:

```
            6?9
      2  4  5  8  1  3
            6?3
      DOMAIN ERROR
             6?3
              ↑
            3?5
      3  5  1
            4?6
      6  2  5  4
```

To select N elements at random from a vector V, the following form can be used:

```
    V[N?⍴V]
```

NUMERICAL FUNCTIONS

The numerical functions consist of:

* Matrix inverse (⌹)
* Matrix divide (⌹)
* Decode (⊥)
* Encode (⊤)

The numerical functions apply only to numeric arguments and produce only numeric results.

MATRIX INVERSE AND MATRIX DIVIDE FUNCTIONS. The matrix inverse and matrix divide functions are both denoted by the domino symbol (⌹).

The matrix inverse function is of the form

```
    ⌹A
```

This function produces the inverse of a non-singular matrix. (A non-singular matrix is one in which all rows and all columns are linearly independent. For example,

```
    2 2 2 2
    2 2 2 2
```

is a singular matrix.)

An example of matrix inverse is

```
      A
1  2  3  4
2  3  4  5
3  4  5  6
4  5  6  7
      ⌹A
¯4.270079647E15   3.469439713E15   5.871359514E15  ¯5.07071958E15
 8.006399338E15  ¯8.006399338E15  ¯8.006399338E15   8.006399338E15
¯3.202559735E15   5.604479536E15  ¯1.601279868E15  ¯8.006399338E14
¯5.337599558E14  ¯1.067519912E15   3.736319691E15  ¯2.135039823E15
```

The result is such that (⌹A)+.×A yields an identity matrix (that is, produces a left inverse).

The matrix divide function is of the form

```
    A⌹B
```

The matrix divide expression

```
X←A⌹B
```

can be used to solve systems of linear equations. For example,

```
           A
      1  0  1  0
      1  1  0  0
      1  1  1  0
      1  1  1  1
           ⌹A
        1  1 ¯1  0
       ¯1  0  1  0
        0 ¯1  1  0
        0  0 ¯1  1
           A+.×⌹A
      1  0  0  0
      0  1  0  0
      0  0  1  0
      0  0  0  1
           B←2 4 6 8
           X←B⌹A
           X
   0  4  2  2
           A+.×X
   2  4  6  8
           (⌹A)+.×B
   0  4  2  2
```

```
           C←4 2⍴1 2 3 4 2 4 6 8
           C
      1  2
      3  4
      2  4
      6  8
           R←C⌹A
           A+.×R
      1  2
      3  4
      2  4
      6  8
           (⌹A)+.×C
       2  2
       1  2
      ¯1  0
       4  4
```

The matrix inverse and matrix divide functions apply to <u>singular</u> and <u>non-square</u> matrices, and to vectors and scalars, but not to arrays of <u>rank greater</u> than 2 (this produces a rank error). The expression

```
⌹A
```

will produce a result only if A is a <u>non-singular array</u> and the columns of A are <u>linearly independent</u>.

Similarly, the expression

```
R←A⌹B
```

will produce a result only if:

* A and B have the same number of rows.

* The columns of B are linearly independent.

A vector argument is treated by matrix inverse and matrix divide as a one-column matrix and a scalar argument is treated as a matrix of ⍴↔1 1. For scalar arguments A and B, the expression ⌹B is equivalent to ÷B and the expression A⌹B is equivalent to A÷B, except that 0⌹0 produces a domain error (whereas 0÷0 does not).

DECODE FUNCTION. The dyadic decode (base value) function (⊥) evaluates two arguments and computes the sum of all the elements of the right argument raised to a power specified by the base value of the left argument. For example, if A←5 2 8 3 7 and B←1 2 3 4 5, then A⊥B equals 768.

If the left argument is scalar or unit, the scalar value is extended for all the elements of the right argument, as follows:

```
      A←2
      B←8 8 10 2 8 10
      A⊥B
498
```

The decode function is extended to arrays as follows: each of the vectors along the last axis of the first argument is applied to each of the vectors along the first axis of the second argument. If either of the axes is of length 1, it will be extended as necessary to match the length of the axis of the other argument.

Examples of the decode function are:

```
      A←8
      B←1 7 7 7 7 7
      A⊥B
65535
      8⊥B
65535
      A←4 4⍴8
      B←4 4⍴2
      A
 8  8  8  8
 8  8  8  8
 8  8  8  8
 8  8  8  8
      B
 2  2  2  2
 2  2  2  2
 2  2  2  2
 2  2  2  2
      A⊥B
 1170  1170  1170  1170
 1170  1170  1170  1170
 1170  1170  1170  1170
 1170  1170  1170  1170
```

ENCODE FUNCTION. The dyadic encode (representation) function (⊤) is the inverse of the decode function for some arguments. For example,

```
      A←8 8 8 8 8 8
      B←1 7 7 7 7 7
      A⊥B
65535
      A⊤65535
1  7  7  7  7  7
```

The above is not true when the left argument is scalar and the right argument is vector. For example,

```
            A←8
            B←1 7 7 7 7 7
            A⊥B
65535
            A⊤65535
7
```

The encode function applies to arrays in the same manner as the decode function. That is, each vector along the last axis of the left argument is applied to each of the vectors along the first axis of the right argument. For example,

```
          A←4 4⍴8
          B←4 4⍴2
          C←A⊥B◊C
1170   1170   1170   1170
1170   1170   1170   1170
1170   1170   1170   1170
1170   1170   1170   1170
```

DATA TRANSFORMATIONS

The two data transformation functions are format and execute. The format function transforms numeric data in its argument to a character representation of this data. In general, the execute function can be considered the inverse of format, that is, it produces a numeric result from a character argument.

EXECUTE FUNCTION. The execute function, denoted by the symbol ⍎, is both monadic and dyadic and applies to character right arguments and numeric left arguments. The character argument can be scalar, vector or unit.

The execute function considers its character argument to be an APL expression and it executes this expression. If the argument does not constitute a well-formed APL expression, an error results. Note that only valid APL expressions can be used as arguments; system commands are invalid arguments.

An empty vector or one containing only spaces can be used with execute if no assignment arrow is placed to the left of the execute character, as for example,

```
            A←⍎'  '
VALUE ERROR
             A←⍎'  '
              ↑
            ⍎'  '
```

Domain errors result if a non-character argument is used as the right argument of the function.

FORMAT FUNCTION. Format (⍕) is a monadic or dyadic function which converts numeric data to character arrays.

Monadic Format. The monadic format function is of the form:

    ⍕A

The result of the monadic format function looks identical to the result produced by the argument without the format function, however, the format function converts the data to a character representation, as follows:

```
                  'PI IS EQUAL TO ',⍕○1
          PI IS EQUAL TO 3.14159265358979З
```

The argument A may be numeric or character. Numeric values are displayed in accordance with the print precision in effect (see Section IV). The display converts to scaled form if any of the numbers in the data are such that the number of significant digits is greater than the precision in effect.

Examples of monadic format are:

```
              A←3 4⍴6
              ⍕A
          6   6   6   6
          6   6   6   6
          6   6   6   6
              A←2 4⍴23*8
              ⍕A
          7.831098528E10   7.831098528E10   7.831098528E10   7.831098528E10
          7.831098528E10   7.831098528E10   7.831098528E10   7.831098528E10
              A←4 5⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
              ⍕A
        ABCDE
        FGHIJ
        KLMNO
        PQRST
```

Dyadic Format. Dyadic format is of the form

    A⍕B

where A is the control argument and B is the data argument.

The data argument, B, may be any APL expression that produces a result.

* If B is empty (at least one element of ⍴B is zero), the result is the same shape as B except that it is always of type character.

* If B already is a character variable, the result is a copy of B.

* If B is scalar, it is treated as a one-element vector.

* If B is an array of rank 2 or greater, it is formatted according to the contents of argument A.

CONTROL PAIRS. A control pair describes how to format a number by giving the number of characters available for the result, the type of formatting, and the precision of the formatted number.

Width Control. The first number in the control-pair is called the width. This number must be an integer between 0 and 32767. The width controls how many characters the resultant formatted output will occupy. A width value of zero causes the minimum number of characters to be used such that there are two spaces in front of the number. If the width allows more characters that the formatted number requires, spaces are added on the left.

Shape and Precision Control. The second number in a control-pair is called the precision. The sign of the precision controls whether to format the number in decimal form or in scaled form. If precision is positive, the data is displayed as a sign (no sign for positive data), followed by the integer portion of the data, followed by a decimal point, followed by the fractional part of the data.

The magnitude of precision controls how many fraction digits to return. If the precision is zero, no fraction digits or decimal point are displayed. All numbers are rounded or padded with zeros to obtain the proper number of fraction digits.

If the precision of the control-pair is negative, the data is formatted as a sign (no sign for positive data), a one-digit characteristic, the mantissa digits, an 'E' followed by an exponent sign (no sign if positive), and two exponent digits.

For example,

    2.3462E02

The number of mantissa digits displayed is controlled by the absolute magnitude of precision. The result is rounded or padded with zeros to fit the precision specified. If the precision value is ¯1, the characteristic digit is returned with no decimal point (the E(sign)xx is returned). If the exponent is ≥0, a trailing blank replaces the leading sign.

Control-Pair Formation. Dyadic format requires one control-pair for each column in the data. It is possible, however, to specify the control argument as a scalar, unit, one-element vector, two-element vector, or a vector with one control-pair (two elements) for each data column. When the control variable is a scalar, unit, or one-element vector, then it is treated as a one-control-pair with a width value of zero. If the control variable has only one control-pair, the control-pair is used on all columns. Note that with dyadic format, the precision for at least one control-pair must be specified.

Dyadic Format Conditions. There are several conditions controlling dyadic format, as follows:

1. If the resulting formatted output is a vector and the width value is zero, any leading blanks are omitted. This is done by not allowing the normal column separation spaces to be placed in front of the first column.

2. The rounding process is performed on the absolute magnitude of the number, thus negative numbers round differently than positive numbers.

3. There are several conditions under which the dyadic format will generate errors:

   a. Domain Error

      1) One of the numbers in the data variable would not fit into the specified width.

      2) The width portion of one of the control-pairs was negative, or was greater than 32767, or was not an integer.

      3) The precision portion of one of the control-pairs was not in the range ¯32768 to +32767, or the value was not an integer.

   b. Length Error

      1) The number of elements in the control variable is not one, two, or the number of data columns times two.

   c. Rank Error

      1) The control variable is higher dimension than a vector, unless it is a unit.

Note: See Section XI for a further discussion of errors.

Examples of dyadic format with control-pairs are:

```
      A←6 6ρ3421.789473
      A
  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473
  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473
  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473
  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473
  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473
  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473  3421.789473
      B←10 3⍕A
      B
  3421.789  3421.789  3421.789  3421.789  3421.789  3421.789
  3421.789  3421.789  3421.789  3421.789  3421.789  3421.789
  3421.789  3421.789  3421.789  3421.789  3421.789  3421.789
  3421.789  3421.789  3421.789  3421.789  3421.789  3421.789
  3421.789  3421.789  3421.789  3421.789  3421.789  3421.789
  3421.789  3421.789  3421.789  3421.789  3421.789  3421.789
      B←9 2⍕A
      B
  3421.79  3421.79  3421.79  3421.79  3421.79  3421.79
  3421.79  3421.79  3421.79  3421.79  3421.79  3421.79
  3421.79  3421.79  3421.79  3421.79  3421.79  3421.79
  3421.79  3421.79  3421.79  3421.79  3421.79  3421.79
  3421.79  3421.79  3421.79  3421.79  3421.79  3421.79
  3421.79  3421.79  3421.79  3421.79  3421.79  3421.79
      B←9 ¯3⍕A
      B
 3.42E03  3.42E03  3.42E03  3.42E03  3.42E03  3.42E03
 3.42E03  3.42E03  3.42E03  3.42E03  3.42E03  3.42E03
 3.42E03  3.42E03  3.42E03  3.42E03  3.42E03  3.42E03
 3.42E03  3.42E03  3.42E03  3.42E03  3.42E03  3.42E03
 3.42E03  3.42E03  3.42E03  3.42E03  3.42E03  3.42E03
 3.42E03  3.42E03  3.42E03  3.42E03  3.42E03  3.42E03
      B←10 ¯4⍕A
      B
 3.422E03  3.422E03  3.422E03  3.422E03  3.422E03  3.422E03
 3.422E03  3.422E03  3.422E03  3.422E03  3.422E03  3.422E03
 3.422E03  3.422E03  3.422E03  3.422E03  3.422E03  3.422E03
 3.422E03  3.422E03  3.422E03  3.422E03  3.422E03  3.422E03
 3.422E03  3.422E03  3.422E03  3.422E03  3.422E03  3.422E03
 3.422E03  3.422E03  3.422E03  3.422E03  3.422E03  3.422E03
```

Quad Output. Quad output is of the form

```
    ⎕←A
```

where A is any APL expression which returns a result.

If A is a character variable, the data is displayed starting at the left margin. If the printing width in effect is reached before the last column is printed, a carriage return/linefeed is generated and printing resumes on the next line, indented six spaces. Arrays of rank three or higher are printed with extra linefeeds in between each dimension. Thus, a three-dimensional variable will print as several two-dimensional arrays with one blank line between each plane. Similarly, a four-dimensional array will print as several groups of three-dimensional arrays with two blank lines between each plane.

Examples of quad output are:

```
      A←4 6⍴123*8
      ⎕←A
 5.238909443E16   5.238909443E16   5.238909443E16   5.238909443E16
      5.238909443E16   5.238909443E16
 5.238909443E16   5.238909443E16   5.238909443E16   5.238909443E16
      5.238909443E16   5.238909443E16
 5.238909443E16   5.238909443E16   5.238909443E16   5.238909443E16
      5.238909443E16   5.238909443E16
 5.238909443E16   5.238909443E16   5.238909443E16   5.238909443E16
      5.238909443E16   5.238909443E16
      A←4 80⍴'ABCD'
      A
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
      ⎕←A
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
      A←4 4 4⍴'ABCD'
      ⎕←A
ABCD
ABCD
ABCD
ABCD

ABCD
ABCD
ABCD
ABCD

ABCD
ABCD
ABCD
ABCD

ABCD
ABCD
ABCD
ABCD
```

Quad Input. Quad input is of the form

```
  ⎕←A
```

The system writes the characters ⎕: and unlocks the keyboard on the next line, indented six spaces, and awaits input. At this point, any APL expression may be entered. This expression is evaluated and the result is used as the value of A.

Quad input example:

```
      A←⎕
⎕:
      34.2
      A
34.2
```

Quote Quad Output. Quote quad output is of the form

```
⍞←A
```

where A is any APL expression.

Operation of quote quad output is exactly the same as quad output except that the concluding carriage return/linefeed is not generated. This is useful in the case where either the next output results from quote quad or the next input request results from quote quad. In these two cases the carriage starts where it left off with the last quote quad output.

Quote quad output example:

```
      ⍞←'THIS IS A '⋄⍞←'TEST'
THIS IS A TEST
      'THIS'⋄'IS A TEST'
THIS
IS A TEST
      A←4 3⍴123*8
      ⍞←A
  5.238909443E16    5.238909443E16    5.238909443E16
  5.238909443E16    5.238909443E16    5.238909443E16
  5.238909443E16    5.238909443E16    5.238909443E16
  5.238909443E16    5.238909443E16    5.238909443E16
```

Quote Quad Input. Quote quad input is of the form

```
A←⍞
```

where the result is always a string of zero or more characters.

Quote quad input reads in the line of characters typed by the user and creates a character vector result to contain that input. Any characters may be entered from zero characters (carriage return) up to the maximum number of characters allowable by the system (the printing width in effect is ignored). In the case where a preceding quote quad output has left the carriage somewhere other than the left margin, the result of the quote quad input is as if the carriage had been spaced to the current carriage position before entering the characters. The system allows backspacing to a point to the left of the last output before entering data, and this is reflected in the result. Note that if characters are entered which do not cause the carriage to advance, visual fidelity (see Section I) will not be preserved in the output, because the computer treats every output character as if it caused a carriage movement of one space to the left.

Quote quad input example:

```
      A←⍞
3.14159
      A
3.14159
      A←⍞
THE VALUE OF A IS READ BY THE SYSTEM AS IT IS TYPED IN
      A
THE VALUE OF A IS READ BY THE SYSTEM AS IT IS TYPED IN
```

SYSTEM FUNCTIONS AND SYSTEM VARIABLES

SECTION IV

The set of primitive APL functions described in Section III deals only with abstract items such as numeric and character arrays. To deal with concrete items, such as system resources, a set of system variables is identified for use in communicating among the user, APL, and the system (MPE) in which APL resides.

The system variables are used for interaction between APL and its environment; however, there are situations where it is more convenient to use functions based on system variables when the system variables themselves may not be explicitly available to users. Such functions are called system functions.

System variables and system functions are denoted by distinguished names. These are formed by the quad symbol (⎕) followed by a name denoting the variable or function (for example, ⎕IO or ⎕SVQ). Such names cannot be used for user-defined objects, and cannot be copied or erased.

SYSTEM FUNCTIONS

Twenty four system functions are provided

Canonical representation ⎕CR

Capture stack environment ⎕CSE

Convert ⎕CV

Delay ⎕DL

Expunge ⎕EX

Function establishment ⎕FX

Monitor values ⎕MV

Name classification ⎕NC

Name list ⎕NL

Query monitor ⎕QM

Query stop ⎕QS

Query trace ⎕QT

Release stack environment ⎕RSE

Reset monitor ⎕RM

Reset stop ⎕RS

Reset trace ⎕RT

Set monitor ⎕SM

Set stop ⎕SS

Set trace ⎕ST

Shared variable control ⎕SVC

Shared variable offer ⎕SVO

Shared variable retract ⎕SVR

Shared variable query ⎕SVQ

Vector representation ⎕VR

Four system functions -- shared variable control (⎕SVO), shared variable offer (⎕SVO), shared variable query (⎕SVQ), and shared variable retract (⎕VR) -- are concerned with the management of the shared-variable facility and are described in Section V.

The convert (⎕CV) system function performs data conversions and is described in Section VI.

The capture stack environment (⎕CSE) and release stack environment (⎕RSE) system functions are used with the extended control facility and are described in Section X.

The following system functions are used as debugging aids and are described in Section X:

Monitor values (⎕MV)

Query monitor (⎕QM)

Query stop (⎕QS)

Query trace (⎕QT)

Reset monitor (⎕RM)

Reset stop (⎕RS)

Reset trace (⎕RT)

Set monitor (⎕SM)

Set stop (⎕SS)

Set trace (⎕ST)

The remaining seven system functions are listed in table 4-1 and are described in this section.

System functions can be referenced or executed like any other function. They are monadic or dyadic, as appropriate, and have explicit results. In most cases, they also have implicit results, in that their execution causes a change in the environment. The explicit result always indicates the status of the environment relevant to the possible implicit result.

CANONICAL REPRESENTATION FUNCTION

The canonical representation function is denoted by the name ⎕CR. When applied to a character argument representing the name of an already established user-defined function, the ⎕CR function produces the user-defined function's canonical representation. For example, if ROOTS is a user-defined function.

```
      ⎕CR 'ROOTS'
 ROOTS
 'ENTER A NUMBER'
 'AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT'
 'AND THE CUBE ROOT'
LABEL1:N←⎕
LABEL2:A←N*÷2
LABEL3:B←N*÷3
 'THE SQUARE ROOT IS ',⍕A
 'THE CUBE ROOT IS ',⍕B
 'ENTER 0 IF YOU DO NOT WISH TO CONTINUE'
LABEL4:N←⎕
 →(N≠0)/5
```

The status of the original function ROOTS is unchanged and it can be executed by entering ROOTS.

```
          ROOTS
      ENTER A NUMBER
      AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
      AND THE CUBE ROOT
      ⎕:
          45
      THE SQUARE ROOT IS 6.708203932
      THE CUBE ROOT IS 3.556893304
      ENTER 0 IF YOU DO NOT WISH TO CONTINUE
      ⎕:
          0
```

When applied to any argument which does not represent the name of an unlocked defined function, ⎕CR returns a matrix of dimensions 0 0. For example,

```
          ⍴⎕CR 'NONE'
      0  0
```

Possible error reports for ⎕CR are rank error if the argument is not a vector or scalar, or domain error if the argument is not character.

VECTOR REPRESENTATION FUNCTION

The vector representation function (⎕VR) is similar to the canonic representation function (⎕CR), the difference being that the result of ⎕VR is a vector with carriage return characters used to separate lines of the function, instead of trailing blanks. Note that there is no carriage return on the last line of the result. Note also that the result of ⎕VR usually takes considerably less storage space than does that of ⎕CR when executed with the same argument, because there are no blank characters needed to fill each row of the matrix result of ⎕CR.

An example:

```
          ⎕VR 'ROOTS'
       ROOTS
       'ENTER A NUMBER'
       'AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT'
       'AND THE CUBE ROOT'
      LABEL1:N←⎕
      LABEL2:A←N*÷2
      LABEL3:B←N*÷3
       'THE SQUARE ROOT IS ',⍕A
       'THE CUBE ROOT IS ',⍕B
       'ENTER 0 IF YOU DO NOT WISH TO CONTINUE'
      LABEL4:N←⎕
       →(N≠0)/5
```

Table 4-1. System Functions

NAME	SYMBOLS	REQUIREMENTS			EFFECT ON ENVIRONMENT	EXPLICIT RESULT
		RANK	LENGTH	DOMAIN		
Canonical representation	⎕*CR N*	1≥*ρρN*		Character array.	None.	Canonical representation of *N*. The result for anything other than an unlocked defined function has the dimensions 0 0.
Function establishment	⎕*FX N*	None		Character matrix, vector, or unit.	Fix (establish) definition of the function represented by *N*, unless its name is already in use for an object other than a function which is not halted.	A vector representing the name of the function established, or the scalar row index of the fault which prevented establishment.
Expunge	⎕*EX N*	2≥*ρρN*		Character array.	Expunge (erase) objects named by rows of *N*, except groups, labels, or halted functions.	A boolean vector whose *I*th element is 1 if the *I*th name is now free, or 0 if the *I*th name is not free.
Name list (monadic)	⎕*NL N*	1≥*ρρN*	1≥*ρ,S*	∧/*N*∊1 2 3 4	None.	A matrix of rows (in random order) representing names of designated kinds in the dynamic environment: 1, 2, 3, 4 for labels, variables, APL functions, and APLGOL functions respectively.
Name list (dyadic)	*A*⎕*NL N*	1≥*ρρN*		∧/*N*∊1 2 3 4 Elements of *A* must be alphabetic.	None.	As for the monadic form, except that only names beginning with letters in *A* will be included.
Name classification	⎕*NC A*	2≥*ρρM*		Character array.	None.	A vector giving the usage of the name in each row of *A*: 0-name is available 1-label 2-variable 3-APL function 4-APLGOL function 5-name unavailable
Delay	⎕*DL N*	1≥*ρρN*		Numeric value.	None, but requires *N* seconds to complete.	Scalar value of actual delay.
Vector representation	⎕*VR N*	1≥*ρρN*		Character vector	None.	Vector representation of *N*. The result for anything other than an unlocked defined function has the dimensions 0 0.

FUNCTION ESTABLISHMENT

A function can be created with the system function denoted by ⎕FX. The argument to the function must be a character vector or matrix, and must be a matrix or vector canonical representation. ⎕FX is executed with the character representation of the function as its argument and produces as an explicit result a character vector of the name of the function (this is the name contained in the first statement of the function). If ⎕FX cannot establish the function, it returns a scalar numeric denoting the line number (⎕IO dependent) in which the error was found.

The ⎕FX function returns the name of the function being created (BOOTS). For example,

```
              TEST←⎕CR 'ROOTS'
              TEST[1;2]←'B'
              TEST
          BOOTS
          'ENTER A NUMBER'
          'AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT '' '
          'AND THE CUBE ROOT'
         LABEL1:N←⎕
         LABEL2:A←N*÷2
         LABEL3:B←N*÷3
          'THE SQUARE ROOT IS ',⍕A
          'THE CUBE ROOT IS ',⍕B
          'ENTER 0 IF YOU DO NOT WISH TO CONTINUE'
         LABEL4:N←⎕
          →(N≠0)/5
```

EXPUNGE FUNCTION

The expunge function is denoted by the name ⎕EX and is used to eliminate an object from the active workspace.

The ⎕EX function will not expunge a label or a halted function. (A label is a name used to identify a specific statement in a defined function, and a halted function is a function that has been halted while in execution mode.)

The ⎕EX function returns a logical vector result of 1 if the name is presently available, or a result of 0 if it is not. A 0 also is returned if the argument used with ⎕EX is not a well-formed name. A rank error is reported if the argument is of higher rank than a matrix, or a domain error if the argument is not character.

An example of the ⎕EX function is as follows:

```
              ⎕EX 'ROOTS'
         1
```

NAME LIST FUNCTION

The name list function is denoted by the name ⎕NL and can be used monadically or dyadically. ⎕NL returns a character matrix, each row of which represents the name of a label, variable, or function currently in the dynamic environment.

When used dyadically, the left argument is a scalar or vector which restricts the names produced to those whose initial letter is the same as a letter occurring in the argument. For example, if the left argument is A, then only names beginning with A will be produced by the ⎕NL function. The right argument of ⎕NL is a scalar or vector whose values may be the integers 1, 2, 3, or 4. The values 1, 2, 3, and 4 respectively produce the names of labels, variables, APL functions, and APLGOL functions.

If the vector value 1 2 3 4 is used as the right argument of ⎕NL, the names from all categories are produced. The results produced are in the order in which the names first appeared in the workspace.

Examples of the ⎕NL function used dyadically are:

```
      'BERT' ⎕NL 2 3
E
ROOTS
B
EDIT1
RESHAPE1
RESHAPE2
      'B' ⎕NL 2 3
B
      'R' ⎕NL 3
ROOTS
```

When used monadically, there is no restriction on initial letters. The right argument performs the same as when used dyadically. An example of monadic use is:

```
          ⎕NL 2 3
      CIRCLEAREA
      E
      ROOTS
      N
      A
      B
      C
      Y
      EDIT1
      APLGOL1
      APLGOL2
      APLGOL3
      APLGOL4
      APLGOL5
      APLGOL6
      APLGOL7
      APLGOL8
      APLGOL9
      APL11
      APL31
      APL32
      APL33
      APL34
      APL35
      APL51
      APL52
      APL61
      APL62
      APL101
      APL102
      APL103
      APL104
      APLSET
      YIELD
      INCOME
      VEC
      XQR
      CHAR
      SHAPE
      RESHAPE1
      RESHAPE2
      D
      X
      Z
      VECTOR
      ALTER
      ARRAY
```

Further uses of the ⎕NL function include the following:

* In conjunction with the expunge function (⎕EX), all the objects of a certain class can be dynamically erased; or a function can be defined that will clear a workspace of all but a preselected set of objects.

* In conjunction with the canonical representation function (⎕CR), functions can be written to display automatically the definitions of all or certain functions in the workspace, or to analyze the interactions among functions and variables.

* The dyadic form of ⎕NL can be used as a convenient guide in the choice of names while designing or experimenting with a workspace.

NAME CLASSIFICATION FUNCTION

The monadic name classification function is denoted by the name ⎕NC. This function accepts scalar, vector, or matrix arguments and returns a numerical indication of the class of the name (or names) represented by the argument. For example,

```
      ⎕NC 'ROOTS'
3
      ⎕NC'ABN'
0
      ⎕NC 'C'
2
      ⎕NC 'A'
2
```

The result of the ⎕NL function can be used as an argument for ⎕NC, but other character arrays may also be used. The results are integer values from 0 to 5. The integers 1, 2, 3, and 4 have the same meanings as for ⎕NL; a result of 0 signifies that the corresponding name is available for any use; and a result of 5 signifies that the name is not valid because it is a distinguished name, or is incorrectly formed.

DELAY FUNCTION

The delay function is denoted by the name ⎕DL and causes a pause in the execution of the statement in which it appears. The duration of the pause, in seconds, is determined by the argument of the ⎕DL function; the accuracy, however, is limited by possible contending demands on the system when the statement is executed. Additionally, the delay can be overridden by a hard interrupt.

The result of the ⎕DL function is a scalar value equal to the actual delay. If the argument used with ⎕DL is not a numeric scalar value, a rank or domain error is reported.

Because the delay function uses only a small amount of computer time compared to connect time, it can be used repeatedly in situations where it is desirable to determine if an expected event has occurred. This is useful in interactions between a program and the user, and in work with shared variables as discussed in Section V.

Example:

```
              TIME←⎕DL 3◇TIME
        3.032000065
```

SYSTEM VARIABLES

System variables are shared between a workspace and the APL system, thus they are instances of shared variables which are discussed in Section V. Sharing occurs automatically when a workspace is activated, or, when a system variable is used in a function, each time that function is used.

The characteristics of shared variables that are significant here are:

* When a variable is shared by two processors, the value of the variable may be different for each processor.

* Each processor is free to use or not use the value specified by the other processor for a variable.

System variables are shown in table 4-2. Included is the name of each variable, the name used to denote the variable, its purpose, its value in a clear workspace (where appropriate), and its meaningful range. Note that there are two classes of system variables, as follows:

1. The value specified by the user (or available in a clear workspace) for a system variable is used by the processor in operations relating to this variable. If the value is inappropriate, a domain error occurs at assignment execution.

   Included in this class are:

   Assert level ⎕AL

   Comparison tolerance ⎕CT.

   Horizontal tabs ⎕HT.

   Index origin ⎕IO.

   Language ⎕LA

   Latent expression ⎕LX.

Table 4-2. System Variables

NAME	SYMBOLS	PURPOSE	INITIAL VALUE	MEANINGFUL RANGE
Comparison tolerance	⎕CT	Contains the comparison tolerance. Used in monadic , dyadic < ≤ = ≥ > ≠ ϵ	1E¯13	0 to 1
Index origin	⎕IO	Contains the index origin. Used in indexing and in ? ⍳ ⍋ ⍒ ⍉ ⎕FX	1	0,1
Latent expression	⎕LX	Executed on activation of workspace	'' (empty vector)	characters
Printing precision	⎕PP	Contains the print precision. Affects numeric output and monadic format	10	1 to 16
Printing width	⎕PW	Contains printing width	80	20 to 255
Random link	⎕RL	Contains the random link. Used in *roll* and *deal* primitive functions	0	0 to 1
Account information	⎕AI	Contains connect time this session and CPU time this session, in milliseconds	—	Cannot be set
Atomic vector	⎕AV	Contains all available characters in APL	See page 4-17	Cannot be set
Line counter	⎕LC	Contains statement numbers of functions in execution or halted, most recently activated first	0	Cannot be set
Time stamp	⎕TS	Contains year, month, day of month, hour (24-hour clock), minute, second, millisecond.	—	Cannot be set
Assertion level	⎕AL	Contains the APLGOL assertion used in APLGOL assertion-checking	0	−32768 to 32767 (integer)
Execution trace	⎕XT	Contains trace information. Prints value in TRACE format		Any value
Branch trace	⎕BT	Prints value in TRACE format as if value were argument to branch (→)		Any value
Virtual memory	⎕VM	Contains virtual memory paging scheme parameters	256 ¯24	N[1] : 2*X 7≤X≤12 N[2] : X>0 - 2 × Y 2≤Y≤L X<0 : 2≤X≤L L stack size dependent
Language	⎕LA	Contains language setting	'APL'	'APL' 'APLGOL'
Terminal type	⎕TT	Contains internal terminal type	Same as previous workspace	See page 4-22

Table 4-2. System Variables (Continued)

NAME	SYMBOL	PURPOSE	INITIAL VALUE	MEANINGFUL RANGE
Horizontal tab setting	⎕HT	Contains tab positions	10	Non-negative integer vector
Work area available	⎕WA	Contains amount of space still unused in workspace (in bytes)	1610474 bytes	Cannot be set
Stack names	⎕SN	Contains character matrix of names of suspended functions	0 0ρ "	Characters
Workspace identification	⎕WI	Contains workspace identification	"	Characters
Backspace	⎕B	Backspace character	ASCII 8 0-origin ⎕AV [148]	Cannot be set
Linefeed	⎕L	Linefeed character	ASCII 10 0-origin ⎕AV [140]	Cannot be set
Return	⎕R	Carriage return (new line) character	ASCII 13 0-origin ⎕AV [152]	Cannot be set
Tab	⎕T	Tab character	ASCII 9 0-origin ⎕AV [141]	Cannot be set
Null	⎕N	Null character	ASCII 0 0-origin ⎕AV [138]	Cannot be set
Escape	⎕E	Escape character	ASCII 27 0-origin ⎕AV [166]	Cannot be set
Alphabet	⎕A	Alphabet	ABCDEFGHIJKLM NOPQRSTUVWXY	Cannot be set
Digits	⎕D	Digits	0123456789	Cannot be set

```
Printing precision ⎕PP.

Printing width ⎕PW.

Random link ⎕RL.

Terminal type ⎕TT

Virtual memory ⎕VM

Workspace identification ⎕WI
```

2. The value specified by the user is not used. The APL processor always resets the variable before it is used.

   Included in this class are:

   Account information ⎕AI

   Alphabet ⎕A

   Atomic vector ⎕AV

   Backspace ⎕B

   Branch trace ⎕BT

   Digits ⎕D

   Escape ⎕E

   Execution trace ⎕XT

   Line counter ⎕LC

   Linefeed ⎕L

   Null ⎕N

   Return ⎕R

   Stack names ⎕SN

   Tab ⎕T

   Time stamp ⎕TS

   Working area ⎕WA

COMPARISON TOLERANCE

The comparison tolerance system variable is denoted by the name ⎕CT and is used to establish the tolerance for the monadic functions less (<), not greater (≤), equal (=), not less (≥), greater (>), not equal (≠), floor (⌊), and ceiling (⌈); and the mixed functions index of (⍳) and membership (∊).

In APL\3000, as with all languages, floating-point numbers are represented in a finite number of bits. This makes some floating-point numbers difficult to represent exactly. For example, the question "is A equal to B" is straightforward unless floating-point numbers represented in a finite number of bits (64 bits for APL\3000) are involved. The A=B question then becomes harder to answer because many floating- point numbers cannot be represented exactly in 64 bits. Thus, problems arise if the equals test is defined to be "exact." The following example illustrates this point.

```
      A←÷97◊A
1.0309278355¯02
      ⎕CT←0          ⍝ THIS MAKES '=' AN EXACT TEST
      1=A×97
0
      ⍝ BECAUSE 1/97 CANNOT BE STORED EXACTLY
      ⍝ THEN 'A' IS NOT A NUMBER THAT CAN
      ⍝ BE MULTIPLIED BY 97 TO RETURN 1
```

This particular way to define = is then not very consistent with the way = would be expected to act. Thus the definition of = (and some related functions) is not an "exact" definition, but is relative to the magnitude of the operands and the value of ⎕CT. The definition is

```
X←|A-B                 [1]
Y←⌈/(|A),|B            [2]
IF (Y×⎕CT)≥X THEN      [3]
A IS EQUAL TO B
```

Notice that the above set of equations, while concise and correct, is difficult to understand. Paraphrasing them as follows may help:

Equation [1] sets the variable X to the absolute value of the difference of the two arguments A and B.

Equation [2] sets Y to the absolute value of the larger of the two arguments A and B.

The third (and crucial) equation [3] states that the arguments are defined to be equal if ⎕CT times the larger of the arguments (Y) is larger that the difference between the arguments.

Note that ⎕CT does not specify the absolute difference between the arguments but the difference relative to the size of the arguments. Thus two big numbers need not be as close, in an absolute sense, as two small numbers. Note that under this definition, if ⎕CT is 0, the equals test is exact in that the difference between the arguments A and B must be 0, exactly, for equation [3] to be true.

The functions (less, not greater, equal, not less, greater, not equal, floor, ceiling, index of, and membership) for which ⎕CT establishes the tolerance result in an error unless the operand(s) are considered "integers." In APL\3000, this test for integer is done in the following way:

1) First, the integer closest to the argument is obtained.

2) Second, the integer obtained in 1) is compared in a relative sense to the argument.

3) If the integer from 1) is relatively equal to the argument, that integer is used as the argument.

A comparison tolerance example:

```
      A←34*÷5◇A
2.024397458
      B←33*÷5◇B
2.012346617
      A=B
0
      ⎕CT←1E¯4
      A=B
0
      ⎕CT←1E¯2
      A=B
1
```

INDEX ORIGIN

The index origin system variable is denoted by the name ⎕IO and is used to establish the index origin (1-origin or 0-origin) for the monadic function roll (?); the mixed functions deal (?), index generator (⍳), index of (⍳), grade up (⍋), grade down (⍒), and transpose (⍉); and the system function fix (⎕FX). For example,

```
      A←1 2 3 4 5 6 7 8 9 0
      ⎕IO←1
      A[4]
4
      ⎕IO←0
      A[4]
5
```

LATENT EXPRESSION

The latent expression system variable is denoted by the name □LX. The APL statement represented by a latent expression is executed automatically whenever a workspace is activated. For example, if the expression

    □LX '''THIS IS WORKSPACE 3'''

is entered and workspace WS3 is saved, the phrase THIS IS WORKSPACE 3 will be displayed when WS3 is activated. See below.

```
              □LX←'''THIS IS WORKSPACE 2'''
              )SAVE WS2
        SAVED 11:12 10/14/76  WS2
              )LOAD WS2
        SAVED 11:12 10/14/76
        THIS IS WORKSPACE 2
```

The form □LX←'→□LC' can be used to restart a suspended function automatically and the form □LX←'TEST' also may be used to activate the function TEST when a workspace is activated. For example,

```
              □LX←'TEST'
              )EDIT
        [0]             TEST;□LX
        [1]             □LX←'□C,ρ□←''LATENT EXPRESSION DEMONSTRATION'''
        [2]             'FUNCTION TEST WILL BE CALLED AUTOMATICALLY'
        [3]             return
        >END
              )SAVE WS1
        SAVED 11:14 10/14/76  WS1
              )LOAD WS1
        SAVED 11:14 10/14/76
        FUNCTION TEST WILL BE CALLED AUTOMATICALLY
```

Note that system commands may be used with □LX. For example, □LX←')FNS' is valid.

RANDOM LINK

The random link system variable is denoted by the name ⎕RL. The random link is a value used by APL to generate random numbers for the roll (?) and deal (?) functions. The random link variable has a value of 0 when a workspace in first activated. After a roll or deal function is executed, the random link is changed, so that when the roll or deal function is executed again the same set of random numbers is not repeated. For example,

```
      ⎕RL
0
      7?9
1  8  6  2  9  5  4
      ⎕RL
9.928070009E¯02
      7?9
4  5  3  8  2  9  6
      ⎕RL
.5041744709
```

If the random link is set by the user before executing a roll or deal function, this value is used by APL as the link value. For example,

```
      ⎕RL←0
      7?9
1  8  6  2  9  5  4
      ⎕RL←0
      7?9
1  8  6  2  9  5  4
      ⎕RL←.5576
      7?9
4  3  7  9  1  8  6
      ⎕RL←.5576
      7?9
4  3  7  9  1  8  6
```

PRINTING PRECISION

The printing precision system variable (⎕PP) contains the precision of values displayed, Examples are:

```
      ⎕PP
10
      A←34*12
      A
2.386420684E18
      ⎕PP←8
      A
2.3864207E18
      ⎕PP←6
      A
2.38642E18
      ⎕PP←4
      A
2.386E18
```

PRINTING WIDTH

The printing width system variable (⎕PW) contains the printing width for values displayed by APL.

An example:

```
      ⎕PW
80
      A←⍳80◊A
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22
      23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41
      42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
      61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
      80
      ⎕PW←40◊A
1  2  3  4  5  6  7  8  9  10  11  12
      13  14  15  16  17  18  19  20  21
      22  23  24  25  26  27  28  29  30
      31  32  33  34  35  36  37  38  39
      40  41  42  43  44  45  46  47  48
      49  50  51  52  53  54  55  56  57
      58  59  60  61  62  63  64  65  66
      67  68  69  70  71  72  73  74  75
      76  77  78  79  80
```

ACCOUNT INFORMATION

The account information system variable is denoted by the name ⎕AI. Its result is the CPU time and the connect time used so far in the session, in milliseconds.

An example of the ⎕AI system variable is:

```
          ⎕AI
39525  ¯1610229
          ⎕AI
39871  ¯1619113
```

ATOMIC VECTOR

The atomic vector system variable is denoted by the name ⎕AV. Its value is a 256-element character vector containing all possible APL characters.

Indices of known characters, such as A, B, =, <, and so forth, can be determined by an expression such as ⎕AV⍳'AB=<'.

Examples of the ⎕AV variable are:

```
      ⎕AV
0123456789 AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ∆⍙<≤>≥=≠∨∧~∊↑↓⊥⊤/
      \()[]¯→←∇⎕'∘.;:⋄-+÷×?⍴⍳○⋆⌈⌊|,●⊖⌽!⌹⍱⍲⌽⌶⍒⍋⍞⍝⍀⌿⍕⍎⍫$¨⊢⊣$⍵⍺⊂⊃∪∩
```

Note that printing of ⎕AV may result in erratic terminal behavior due to the output of control characters.

LINE COUNTER

The line counter system variable, denoted by the name ⎕LC, produces a vector of the statement numbers of functions in execution or halted. The most recently activated statement numbers are displayed first. For example,

```
      ROOTS
ENTER A NUMBER
AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
AND THE CUBE ROOT
⎕:
      34
ROOTS[5]*
      ⎕LC
5
      →6
ROOTS[7]*
      ⎕LC
7
      →10
⎕:
      0
```

STACK NAMES

The stack names system variable (⎕SN) returns the names of all user-defined functions on the stack. For example,

```
        ROOTS
ENTER A NUMBER
AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
ROOTS[3]*
        ⎕SN
ROOTS
        →10
⎕:
        0
```

WORKSPACE IDENTIFICATION

The workspace identification system variable (⎕WI) contains the name of the active workspace. If the workspace is unnamed, an empty vector is returned. For example,

```
        ⎕WI

        )LOAD WS2
SAVED 11:12 10/14/76
THIS IS WORKSPACE 2
        ⎕WI
WS2
```

TIME STAMP

The time stamp system variable is denoted by the name ⎕TS and returns the year, month, day of the month, hour (24-hour clock), minute, second, and millisecond. For example,

```
        ⎕TS
1976    10   14   11   33   1   700
```

ASSERTION LEVEL

The assertion level system variable (⎕AL) establishes APLGOL assertion checking level. The ⎕AL system variable indicates the lower bounds of assertions to be checked. Each time an ASSERT statement is encountered in an APLGOL user-defined program, the assertion level is checked against the first expression in the ASSERT statement. If the assertion level is smaller than the level set by ⎕AL, the entire statement is regarded as a comment and is not executed. See Section IX for a further discussion of the APLGOL ASSERT statement.

EXECUTION TRACE

The execution trace (⎕XT) system variable is used to trace the execution of a statement, or to determine the type (character or numeric), shape, and value, of the result of an APL expression. When read, ⎕XT always has the value '' (empty character vector). Upon assigning a value to ⎕XT, however, the type, shape, and value are displayed on the terminal in the same format as when tracing a function with the ⎕ST system variable. See Section IX for a discussion of ⎕ST and trace format.

BRANCH TRACE

The branch trace system variable (⎕BT) causes APL to display values in trace format as if the value is an argument of a branch arrow (→). See Section IX for a discussion of trace format.

VIRTUAL MEMORY

The virtual memory system variable (⎕VM) allows a user to control the paging scheme used by APL in managing the active workspace. When read, ⎕VM yields a four-element integer vector whose elements are the page size (in bytes), the number of pages to be used, the number of page faults which have occurred since the last assignment of ⎕VM or the last)SAVE,)LOAD, or)CLEAR (whichever occurred last), and the stack size of the HP 3000 stack used (in words). When assigning a value to ⎕VM, an integer vector is used, the first two elements of which replace the first two elements of ⎕VM, and the rest is ignored.

The first element of the value assigned to ⎕VM must be a power of two between 2*7 and 2*12. The second element can either be positive or negative. If positive, it implies a congruent set paging scheme, and must be a power of two between 2*2 and a number dependent on the stack size. If the second element is negative, it implies a linked list paging scheme, and can be any integer between ¯2 and a negative number again dependent on the stack size.

If either of the first two elements of the vector being assigned to ⎕VM is out of range, the assignment has no effect.

LANGUAGE

The language system variable (⎕LA) contains the default language of the translator. When the APL\3000 editor or the ⎕FX function is used to create a user-defined function, the function is assumed to be in either APL or APLGOL. The argument of ⎕LA is a character vector 'APL' or 'APLGOL' to specify the translator to be used.

TERMINAL TYPE

The terminal type system variable (⎕TT) contains the terminal type. The terminal type is specified by a character vector argument as follows:

'AJ' - Anderson Jacobson

'ASCII' - ASCII

'CDI' - Computer Devices, Inc.

'GSI' - GenCom Systems, Inc.

'DM' - DataMedia

'BP' - Bit Pairing

'CP' - Character Pairing

'HP' - Hewlett-Packard HP 2641A

HORIZONTAL TABS

The horizontal tabs system variable (⎕HT) is used to set internal tab stops and the interpretation of the tab character on input. ⎕HT can be assigned an integer vector, each element of which denotes the number of character positions between a tab stop and the left margin. The vector need not be in any particular order. Upon reading ⎕HT, the tab stop positions, in ascending order, are returned. Assigning an empty vector to ⎕HT causes operation to be as though there were no tab stops.

The ⎕HT system variable has no effect if the terminal type (⎕TT) is ASCII. If ⎕TT = 'ASCII', an implicit ⎕HT is preserved but ignored. Upon subsequent resetting of the terminal type to non-ASCII, an implicit ⎕HT←⎕HT is performed and the stored value becomes effective.

WORK AREA AVAILABLE

The work area available system varaible (⎕WA) has as its value an integer representing, in bytes, the approximate amount of storage still available in the active workspace. This system variable is not explicitly changeable, but changes every time storage in the workspace is used or released.

CHARACTER SYSTEM VARIABLES

Six control character system variables, and three character sequence variables are available. These are scalar (in the case of the control characters) or vector (in the case of the character sequences) variables, whose values are constant from one read to the next. These variables are:

NAME	CHARACTER	ASCII VALUE DECIMAL	ASCII VALUE OCTAL	ATOMIC VECTOR (AV) INDEX (0-ORIGIN)
⎕B	Backspace	8	10	148
⎕L	Linefeed	10	12	140
⎕R	Carriage Return	13	15	152
⎕T	Raw Tab	9	11	141
⎕N	Null	0	0	138
⎕E	Escape	27	33	166
⎕A	Alphabet	'ABCDEFGHIJKLMNOPQRSTUVWXYZ'		
⎕D	Digits	'0123456789'		
⎕AV	Atomic Vector	(See page 4-19)		

SHARED VARIABLES

SECTION V

Shared variables are used to communicate between two processes. This allows two independent concurrently operating processes to cooperate with one another by sharing information which each process can use for its own purposes. Currently, variables may be shared between the active workspace, the APL system, and the file system.

Shared variables may either be global or local, and are similar to ordinary variables except that shared variables may not be used with indexed assignments. A shared variable may appear on the left of an assignment statement, in which case its value is said to be set, or written; or it may be used elsewhere in a statement, in which case its value is said to be used, or read. Either form is defined as an access.

A shared variable can have only one value at any given instant; however, either process can change the value. Thus a process using a shared variable may find its value different from that which it assigned previously, or from one read to the next.

Although a process can share variables with any number of other processes simultaneously, each sharing is bilateral; that is, each shared variable has only two owners. This does not detract from the efficiency of the system because one process can share variables bilaterally with several other processes, controlling their access to these variables as required.

Four system functions are provided to establish the sharing of variables. Two of the functions are used for the actual management of the shared variables, and the other two are used to provide related information. The functions are listed in table 5-1.

OFFERS

An offer to share a variable is performed by the system function ⎕SVO. This function can be used monadically or dyadically. The monadic form is ⎕SVO PN, where PN is a character vector representing a shared-variable identifier. The dyadic form is PI ⎕SVO PN, where PI is a character vector identifying the other process with which sharing is to be accomplished, and PN is as noted above.

The shared-variable identifier generally consists of two names. The first name indicates the variable to be shared, and the second name is a substitute, or surrogate, name which is offered to match a name offered by the other process. The surrogate name is not necessary, only one name need be used. (In this case, the name of the variable is its own surrogate.)

Table 5-1. System Functions for the Management of Sharing

		REQUIREMENTS*				
SYMBOL	NAME	RANK	LENGTH	DOMAIN	EFFECT ON ENVIRONMENT	EXPLICIT RESULT
PI ⎕*SVO PN*	Dyadic offer	2≥ρρ*PN*	(×/¯1↓P)∊1,¯1↓ρ*N*	Characters	Tenders offer to process *P* if first (or only) name of a pair is not previously offered and not already in use as the name of an object other than a variable.	Degree of coupling now in effect for the name pair. Dimension: ×/¯1↓ρ*N*.
⎕*SVO PN*	Monadic offer	2≥ρρ*PN*	None	**	None	Degree of coupling now in effect for the name pair. Dimension:×/¯1↓ρ*N*.
C⎕*SVC PN*	Access control	2≥ρρ*PN* 2≥ρρ*C*	(1≥ρρ*C*)∧1=×/ρ*C* or (ρ*C*)=(¯1↓ρ*N*),4	∧/*C*∊0 1 **	Sets access control.	New setting of access control. Dimension: (¯1↓ρ*N*),4.
⎕*SVC PN*	Access control.	2≥ρρ*PN*	None	**	None	Existing access control.
⎕*SVR PN*	Retraction	2≥ρρ*PN*	None	**	Retracts offer (ends sharing).	Degree of coupling before this retraction. Dimension: ×/¯1↓ρ*N*.
⎕*SVQ P*	Inquiry	1≥ρρ*P*	Vector	Characters	None	If *P* is empty: Vector of identification of processers making offers to this user. If *P*= vector: Matrix of names offered by process *P* but not yet shared.

*If a requirement is not met the function is not executed and a corresponding error report is printed.
**Each row of *N* (or *N* itself if 2≥ρρ*N*) must represent a name or pair of names. If a pair of names is used for an offer (dyadic⎕*SVO*), either the pair, or the first name only, can be used for the other functions.

The surrogate name has no effect other than controlling the matching of the shared variables, thus making it possible for one process to operate with no direct knowledge of, or concern with, the variable name used by the other process. In addition, the same surrogate name may be used for offers to several processes at the same time. When this is done, however, each use of a particular surrogate name must be associated with a different variable name because a variable may be shared with only one other process at any given time.

The explicit result of the expression PI ⎕SVO PN is the degree of coupling of the name or name pair in PN, as follows:

0 - Sharing is not completed.

2 - Sharing is completed.

An offer of a name to any other process increases the coupling if no other offer has been made (0 coupling), and the name is not the name of a label, function, group, or previously shared variable. An offer never decreases the coupling.

An example of the dyadic use of the offer function is as follows:

```
            'FILE' ⎕SVO 'ABC CTRLO'
2
```

The monadic form of the offer function (⎕SVO PN) does not affect the coupling of the variable contained in PN; however, the degree of coupling is reported as the explicit result. If the degree of coupling is 2, a repeated offer to share this variable has no further implicit result. In this case, the monadic or dyadic form may be used for inquiry to determine the degree of coupling.

An example of the monadic use of the offer function is as follows:

```
            ⎕SVO 'ABC'
2
```

The offer function will not produce the proper result unless all the requirements listed in table 5-1 are met. An appropriate error report is generated when the requirements are not met.

A set of offers can be made with one dyadic offer function by using a character matrix left argument, or a scalar, vector, or unit argument which is (automatically) extended, with a character matrix right argument. Each of the rows of the right argument represent a unique name or name pair. The offers are treated in sequence; the explicit result is a vector of the resulting degrees of coupling.

ACCESS CONTROL

As mentioned previously, the value of a shared variable may be changed by either of the processes sharing it. For most applications, it is important to be able to determine whether a new value has been assigned, or whether use has been made of a current value before a new value is assigned. An access control mechanism is incorporated in the APL shared variable facility for this purpose.

The access control uses the dyadic form of the system function SVC to inhibit the setting or use of a shared variable by either of its owners, depending on the access state of the variable, and the value of an access control matrix (ACM) which is set jointly by the two owners.

A delay is caused by an inhibition of an access, resulting in a negligible amount of computer time. The keyboard is locked during this period. A hard interrupt during the delay will abort the access and unlock the keyboard.

The three possible access states for a shared variable, the possible transitions between states, and the potential inhibitions imposed by the access control matrix, ACM, are shown in figure 5-1. ASM in the figure refers to the access state matrix. The codes for the access state matrix are as follows:

0 0 1 1 - Initial ASM (can be used by process A or B).

0 1 0 1 - Can be set by process A.

1 0 1 0 - Can be set by process B.

The operations permissible for any state are indicated by the zeros in the expression ACM∧ASM. Thus, referring to figure 5-1, each of the following statements can be validated.

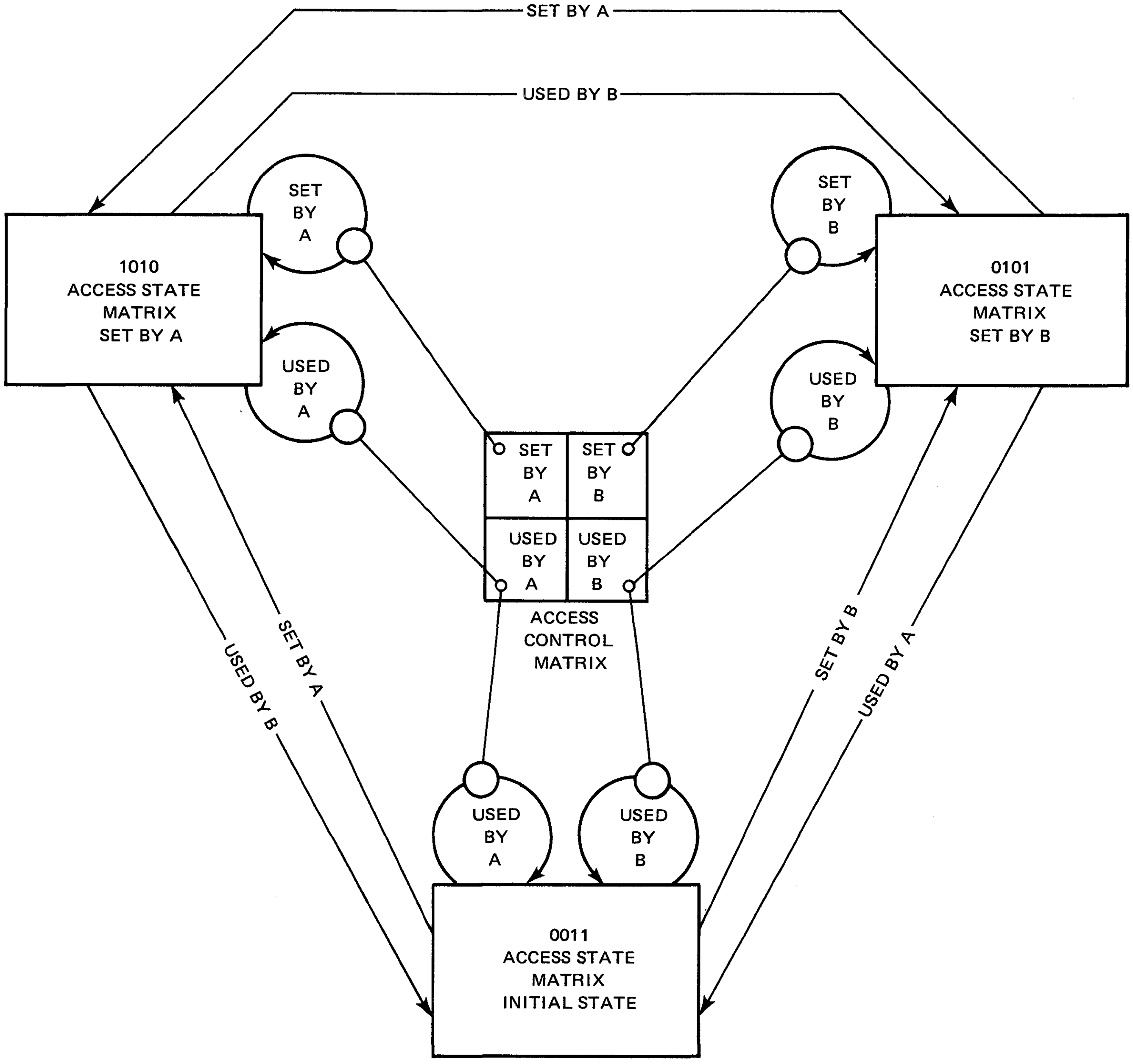
If ACM[1;1]=1	-	Two successive sets by A require an intervening access (set or use) by B.
If ACM[1;2]=1	-	Two successive sets by B require an intervening access (set or use) by A.
If ACM[2;1]=1	-	Two successive uses by A require an intervening set by B.
If ACM[2;2]=1	-	Two successive uses by B require an intervening set by A

The value of the access state matrix (ASM) is not directly available to a user, but the value of the access control matrix (ACM) is. The ACM can be obtained from the monadic function ⎕SVC 'N', where N is the name of the shared variable of interest.

Note that if two owners use the function ⎕SVC 'N', the results are reversed. In other words, if user A enters ⎕SVC 'N', the result is the access control vector 1 4⍴ACM. User B, however, on using the same expression, will obtain the reverse of the access control vector, or ⌽ACM. The reason for the reversal is that sharing is symmetric; that is, neither process has precedence over the other, and each sees a control vector in which the first one of each pair of control settings applies to that process' accesses. This can be seen from figure 5-1; if the rows of A and B are reversed, the access control matrix will be the row reversal of the matrix shown.

The access control matrix setting for a shared variable is determined in a manner that retains the functional symmetry. An expression such as L ⎕SVC 'N' executed by user A assigns the value of the left argument L to a four-element vector. A similar action by user B also results in a four-element vector. If these vectors are called VA (for user A) and VB (for user B), then the value of the access control matrix can be determined as follows:

```
ACM←(2 2⍴VA)∧⌽2 2⍴VB
```



A one in an element of ACM inhibits the associated access. Allowable accesses are given by the zeroes in ACM∧ASM. Access control vectors as seen by A and B, respectively, are ,ACM and ,⌽ACM.

The access state matrix represents the last access: ones occur in the last row if it is not a set, and in a column if it is, the first column if set by A and the last if set by B.

Figure 5-1. Access Control of a Shared Variable

Because the ones in the access control matrix inhibit the corresponding actions, it can only be the case that a user can increase, and not decrease, the degree of control imposed by the other user. A user can, however, restore the control to the minimum level available to him by using the ⎕SVC function with a left argument of all zeros.

The initial values of VA and VB when sharing is first offered are zero. Access control can be imposed only after a variable is offered, however, after once being offered, access control can be imposed whether or not the sharing is completed. In other words, access control can be imposed either before or after the degree of coupling reaches two.

The access state when a variable is first offered (the degree of coupling is one) is always the initial state as shown in figure 5-1. Completion of sharing does not change this access state, however, if the variable is set or used before the offer is accepted, the access state changes accordingly.

Table 5-2 lists several settings of the access control vector. These settings also could be represented by omitting the control matrix from figure 5-1 and deleting the lines representing accesses which are inhibited for each particular case. For example, all inner paths in figure 5-1 would be deleted when maximum restraint (all ones) is imposed.

Table 5-2. Access Control Vector Settings

ACCESS CONTROL VECTOR AS SEEN BY		COMMENTS
A	B	
0 0 0 0	0 0 0 0	No constraints.
0 0 1 1	0 0 1 1	Half-duplex. Ensures that each use is preceded by a set by partner.
1 1 0 0	1 1 0 0	Half-duplex. Ensures that each set is preceded by an access by partner.
1 1 1 1	1 1 1 1	Reversing half-duplex. Maximum constraint.
0 1 1 0	1 0 0 1	Simplex. Controlled communication from B to A. (For card reader, etc.)

Several access control matrices can be set by using matrix arguments in the ⎕SVC functions as follows:

> To set N access control matrices, use an N by 4 matrix left argument for ⎕SVC and an N-rowed right argument of variable names.

The explicit result produced is an N by 4 matrix of the current values (the 1 4⍴) of the control matrices. If control is being set for all inhibits, the left argument can be a single 1; for no inhibits, the left argument can be a single 0.

RETRACTION

The system function denoted by the name ⎕SVR is used to retract sharing offers. The argument of the ⎕SVR function can be a single name to retract a single offer or a matrix of names to retract several offers.

The explicit result of the ⎕SVR function is the degree of coupling for each name specified in the argument prior to retraction. The implicit result is to reduce the degree of coupling for all specified names to zero.

The APL system retracts sharing automatically if the connection to the computer is interrupted, if the user logs off, or if a new workspace is loaded (including clearing the active workspace). Sharing of a variable also is retracted automatically if the variable is erased by either user or, if it is a local variable, upon completion of the function in which it appeared.

The value of a shared variable set by one process often will not be represented in the partner process' workspace until it is actually required to be there. Conditions requiring the value to be represented are when the variable is to be used or when sharing is terminated.

Under any of the above conditions, it is possible that a WS FULL error message will be reported. The prior value of the variable remains in effect in this case, and, after corrective action, the particular action that caused the error message can be repeated and the current value of the variable will be brought into the workspace.

INQUIRIES

The monadic system functions ⎕SVO and ⎕SVC (already discussed), and ⎕SVQ produce information concerning the shared variable environment but do not alter it.

If the ⎕SVQ function is executed with an empty vector argument, the result is a vector containing the identification of each process making any sharing offers.

If the argument to the ⎕SVQ function specifies a particular process, the result is a matrix of variable names offered by the identified process. This matrix does not contain the names of variables which have been accepted by counter offers.

To produce a character matrix showing the names of shared variables in a dynamic environment, the expression shown below can be used:

```
            M←⎕NL 2◇M (0≠⎕SVO M)⌿M
```

The names now will be in variable M.

APL\3000 FILE SYSTEM

SECTION VI

Interface between APL\3000 and MPE is provided by the shared variable facility. A process named 'FILE' shares certain variables when they are offered by an APL user.

The variables which can be shared by the APL workspace and the file process must be offered with the specific surrogate names 'CTRL' or 'DATA' followed by the single digit 0 through 9. For example,

```
    CTRL0                DATA0

    CTRL7                DATA7
```

The digit refers to the file being offered, thus CTRL0 and DATA0 refer to the same file. A maximum of ten files can be shared at the same time.

A third variable ('CMNDS') can be shared between APL and MPE in order to issue certain MPE commands from APL. See page 6-11 for a discussion of the CMNDS variable.

CONTROL VARIABLE

Before a file can be used, it must be opened. The control variable, issued with the surrogate name CTRLn, is used for this purpose. The APL system then invokes the MPE FOPEN intrinsic to open the file. The file name is converted from internal APL characters to ASCII. The foptions parameter of FOPEN is specified as %2003, aoptions as %4, and default values are taken for all other FOPEN parameters. This means that the file is opened as an old binary file, with fixed- length records and no carriage control. These options can be overridden by the file label or the specification of a :FILE command (see the MPE Commands Reference Manual). Additionally, the name is that of a file (as opposed to that of a file equation), and the file is opened for read/write, single record access, buffering, and exclusive access.

Note: See the MPE Intrinsics Reference Manual for a complete discussion of the FOPEN intrinsic.

The shared variable offer system function (⎕SVO) is used to offer to share the control variable with the file system. As described in Section V, the left argument of the ⎕SVO function specifies the process to which the offer is being made. The process name in this case is 'FILE', thus the character vector 'FILE' must be specified as the left argument of ⎕SVO.

The right argument of ⎕SVO is a character vector which consists of two names: the control variable and the surrogate name CTRLn, where n is a digit from 0 through 9. The form of the complete statement is

```
'FILE' ⎕SVO '[controlvariable] CTRLn'
```

For example,

```
'FILE' ⎕SVO 'ABC CTRL0'
```

When the above statement is executed, it returns the degree of coupling, as follows:

0 - The offer is not accepted (usually because of an error, for example, misspelling, or name already shared, etc.).

2 - The offer is accepted.

For example,

```
      'FILE' ⎕SVO 'ABC CTRL0'
2
```

If a 2 is returned, the attempt at establishing communication with the file system was successful. If a 0 is returned, the attempt was unsuccessful.

The control variable (ABC in the above example) must be assigned the name of the file (the "formal file designator") being accessed. This is accomplished as follows:

```
      ABC←'FILE1'
      ABC
1
```

The resulting condition code from the FOPEN attempt can be obtained by accessing the control variable (ABC). The file system will signify the condition code returned by FOPEN by returning one of the following values in the control variable:

CONDITION CODE	APL DISPLAYS
CCE	1
CCG	0
CCL	Negative of the error number returned by the FCHECK intrinsic.

Note: The numbers ¯1000, ¯1001, and ¯1002 are not returned by MPE. These are APL error numbers which have the following meanings:

¯1000	- File already opened. Remains as previously opened.
¯1001	- File not yet opened.
¯1002	- An attempt was made to write to a file with a record size which would cause a stack overflow.

A condition code example,

```
          ABC←'FILE2'
    FILE ERROR
           ABC←'FILE2'
              ↑
          ABC
   ¯52
```

Referenced file does not exist

Note: Only an existing file can be accessed (a new file will not be created if none exists under the name assigned to the control variable). An MPE :FILE command can be entered and the file can be back-referenced as follows:

```
    'FILE' ⎕SVO 3 5⍴'DATAOCTRLOCMNDS'
2 2 2
    CMNDS ← 'FILE L;DEV=LP' ◊ CMNDS
1
    CTRLO  '*L'
    CTRLO
1
```

The * means turn off the "no file equation" bit in FOPEN.

If FILE1 exists:

```
              ABC←'FILE1'
              ABC
      1
```

The control variable also may be assigned numeric vector values which direct the file system to perform certain actions (through MPE intrinsics). The first element of the vector value must be:

0 - Issues an FCLOSE. Elements 2 and 3 of the vector specify the disposition and seccode parameters of FCLOSE. For example,

```
ABC←0 4 0
```

The above statement closes the file identified by the control variable ABC and deletes the file from the system. If element 3 is omitted, it is assumed to be 0. Subsequent reading of the control variable causes the file system to return a scalar value signifying the condition code returned by FCLOSE as follows:

Condition codes:

CCE (1)	- Successful
CCL (<0)	- Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

An example,

```
              ABC←0 4 0
              ABC
      1
```

Note: Issuing a shared variable retract (⎕SVR) on the control variable will close the file with FCLOSE disposition of 0 (no change - if the file is NEW, it is deleted; otherwise, it is returned to its previous disposition domain).

1 - Calls the FCONTROL intrinsic. Elements 2 and 3 of the vector specify the controlcode and param parameters of FCONTROL. For example,

```
ABC←1 6 0
```

The above statement writes an end-of-file mark on the file associated with ABC.

The following actions are available through the FCONTROL intrinsic. Some of these actions only apply to certain types of files (for example, terminals, tapes, and so forth). See the MPE Intrinsics Reference Manual for details.

VECTOR[2]	OPERATION
0	General device control.
1	Line control.
2	Complete input/output.
4	Set time-out interval.
5	Rewind tape.
6	Write end-of-file.
7	Space forward to tape mark.
8	Space backward to tape mark.
9	Rewind and unload tape.
10	Change terminal input speed.
11	Change terminal output speed.
12	Turn ECHO on or off.
14	Disable BREAK.
15	Enable BREAK.
16	Disable CONTROL-Y.
17	Enable CONTROL-Y.
18	Disable tape mode.
19	Enable tape mode.
20	Disable input timer.
21	Enable input timer.
23	Disable parity checking.
24	Enable parity checking.
25	Define line-termination character.
26	Disable binary transfers.

27	Enable binary transfers.
28	Disable user block mode transfers.
29	Enable user block mode transfers.
34	Disable line deletion echo suppresion.
35	Enable line deletion echo suppression.
36	Set parity.
37	Allocate a terminal.
38	Set terminal type.
39	Obtain terminal type information.
40	Obtain terminal output speed.
41	Set unedited terminal mode.

Condition codes:

CCE (1) - Successful

CCL (<0) - Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

2 - Calls the FSPACE intrinsic. The second element specifies the number of records to skip (forward if positive, backward if negative). For example,

```
ABC←2 6
```

skips forward 6 records on the file associated with ABC.

Condition codes:

CCE (1) - Successful

CCG (0) - End-of-file

CCL (<0) - Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

3 - Calls the FPOINT intrinsic. The second element specifies the number of the record at which the file is to be positioned. For example,

```
ABC←3 4
```

points to record 4 in the file associated with ABC.

Condition codes:

CCE (1) - Successful.

CCG (0) - End-of-file.

CCL (<0) - Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

4 - Calls the FSETMODE intrinsic. The second element specifies the modeflags parameter of FSETMODE. For example,

```
    ABC←4 0
```

calls FSETMODE and sets modeflags to 0.

Condition codes:

CCE (1) - Successful

CCL (<0) - Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

5 - Calls the FGETINFO intrinsic and requests "full status" concerning the file. For example,

```
    ABC←5
```

requests a full status report (from FGETINFO) for the file associated with ABC. Reading ABC returns a 25 by 20 character array containing the file information, as follows:

```
      ABC←5◇ABC
MPEFILEINFO
FILENAME←'FILE1    '
GRPNAME ←'GOODWIN '
ACCTNAME←'TEST     '
FOPTIONS←1025
AOPTIONS←4
RECSIZE ←128
DEVTYPE ←0
DEVSUBTP←3
LDEV    ←4
DRT     ←5
UNIT    ←1
FILECODE←0
RECPTR  ←0
EOF     ←0
FLIMIT  ←1023
LOGCOUNT←0
PHYCOUNT←0
BLKSIZE ←128
EXTSIZE ←128
NUMEXTS ←8
USERLAB ←0
CREATOR ←'GOODWIN '
LABADDR ←67110318
```

6 - Calls the FLOCK intrinsic to lock the file. The second element specifies the _lockcond_ parameter of FLOCK (1 for TRUE, 0 for FALSE lock). For example,

```
ABC←6 1
```

locks the file associated with ABC unconditionally (_lockcond_ = TRUE).

Condition codes

The condition codes possible if _lockcond_ = TRUE are

CCE (1) - Successful

CCG (0) - Not returned when _lockcond_ = TRUE.

CCL (<0) - Request denied because this file was not opened with the dynamic locking _aoption_ specified in the FOPEN intrinsic, or the request was to lock more than one file and the calling process does not possess the Multiple RIN Capability (see the _MPE Intrinsics Reference Manual_).

The condition codes possible when _lockcond_ = FALSE are

CCE (1) - Successful

CCG (0) - Request denied because the file was locked by another process.

CCL (<0) - Request denied because this file was not opened with the dynamic locking _aoption_ specified in the FOPEN intrinsic, or the request was to lock more than one file and the calling process did not possess the Multiple RIN Capability (see the _MPE Intrinsics Reference Manual_).

7 - Calls the FUNLOCK intrinsic to unlock the file. For example,

```
ABC←7
```

unlocks the file associated with ABC.

Condition codes:

CCE (1) - Successful

CCG (0) - Request denied because the file had not been locked by the calling process.

CCL (<0) - Request denied because the file was not opened with the dynamic locking _aoption_ of the FOPEN intrinsic, or the _filenum_ parameter is invalid.

8 - Controls auto-ASCII conversion. The second element is a 0 to turn auto-convert OFF, or a 1 to turn auto-convert ON. When executing with auto-convert ON, APL-to-ASCII conversion is performed implicitly; when auto-convert is OFF, no such implicit conversion is performed. All files are initially opened with auto-convert OFF. Once set -- either by the user or by the open -- auto-convert does not change for the duration of the open unless explicitly set by the use of CTRLn.

Note: See the MPE Intrinsics Reference Manual for a discussion of the FCLOSE, FCONTROL, FSPACE, FPOINT, FSETMODE, FGETINFO, FLOCK, and FUNLOCK intrinsics.

When a file is first opened, a 'FILE ERROR MODE' flag is set to zero. When this flag is 0, any attempt to perform an operation on the file system which causes a non-1 return into the control variable will cause APL to suspend execution. An error report is printed, consisting of the line on which the error occurred and the words 'FILE SYSTEM ERROR.'

The control variable may then be read to determine which error occurred.

The 'FILE ERROR MODE' flag may be altered through the use of the control variable. Setting control with the vector 9 0 will set the flag to zero, thus causing APL to report errors. Setting control with the vector 9 1 will cause APL to ignore file system errors (which may still be checked by the return from the control variable).

DATA VARIABLE

Once a file has been opened, data can be written or read from this file using the data variable.

The data variable is offered for sharing with the shared variable offer system function (⎕SVO) and the surrogate name DATAn, where n is a value from 0 through 9. The process named 'FILE' must be used as the left argument of the ⎕SVO function. The form of the complete statement is as follows:

    'FILE' ⎕SVO '[datavariable] DATAn'

For example,

    'FILE' ⎕SVO 'DID DATA0'

When the above statement is executed, it returns the degree of coupling, as follows:

0 - Sharing is not completed.

2 - The offer is accepted.

For example,

```
          'FILE' ⎕SVO 'DID DATAO'
   2
   ↗———————————————————————————— Offer accepted
```

WRITING TO A FILE

If a character vector is assigned to the data variable, the file system will perform an FWRITE to the file -- using the actual byte values of the characters as the data. The length (⍴) of the character vector being written is used as the length parameter in the FWRITE intrinsic. The FWRITE control parameter is always 1, thus allowing embedded carriage control. The condition code status returned by FWRITE can be obtained by reading the control variable.

Condition codes:

CCE (1) - Successful

CCG (0) - End-of-file while attempting a write.

CCL (<0) - Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

An example of writing to a file is as follows:

```
          DID←'THIS IS RECORD 0'
          DID←'THIS IS RECORD 1'
          ABC
   1
          DID←'THIS IS RECORD 2'
          DID←'THIS IS RECORD 3'
          DID←'THIS IS RECORD 4'
          ABC
   1
```

Writing is performed sequentially; thus record 0 is written first, then record 1, record 2, and so forth. To write data to a specific record in the file, a numeric scalar representing the record number is assigned to the data variable before assigning the character data. The FWRITEDIR intrinsic is invoked in this case. For example,

```
          DID←12
          DID←'DIRECT WRITING'
          ABC
   1
```

Again, the status of the FWRITE is returned in the control variable, as follows:

CCE (1) - Successful

CCG (0) - End-of-file

CCL (<0) - Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

READING A FILE

Reading the data variable directs the file system to use the FREAD intrinsic. A character vector representing the contents of a record in the file will be returned. The FREAD is performed <u>sequentially</u>, and successive records are read each time the data variable is read. The number of words per record in the file as opened is used as the length parameter of FREAD. The condition code status returned by FREAD can be obtained by reading the control variable.

Examples of reading a file are

```
      DID
THIS IS RECORD 0
      DID
THIS IS RECORD 1
      ABC
1
      DID
THIS IS RECORD 2
      DID
THIS IS RECORD 3
      DID
THIS IS RECORD 4
      ABC
1
```

Reading the control variable returns the status of the condition code:

CCE (1) - Successful

CCG (0) - End-of-file

CCL (<0) - Unsuccessful. The value returned is the negative of the error number returned by the FCHECK intrinsic.

To read a specific record in the file, a scalar value representing the record number is assigned to the data variable. This positions the file to that record, and the next time the data variable is read, the record is read. The FREADDIR intrinsic is used in this case. For example,

```
      DID←2
      DID
THIS IS RECORD 2
```

CMNDS VARIABLE

The CMNDS variable allows MPE commands to be used from APL by using the MPE COMMAND intrinsic.

The CMNDS variable is offered for sharing with the shared variable offer system function (⎕SVO), as follows:

```
            'FILE' ⎕SVO 'CMNDS'
      2
```

Note that the surrogate name can be reserved.

The MPE command to be issued then is assigned to CMNDS as a character vector.

```
      CMNDS←'FILE LIST;DEV=LP'
```

The condition code status returned by the COMMAND intrinsic can be obtained by reading the CMNDS variable.

```
            CMNDS
      1
```

The negative of the error number is returned if an error occurred.

```
            CMNDS←'LISTF TEST1'
            CMNDS
     ¯108
      ↑________________________Non-existent file
```

DATA CONVERSION

All data read or written by the file system is represented by APL characters. The internal value of any character may be obtained with the atomic vector system variable (⎕AV) by executing ⎕AV⍳C, where C is a vector of characters for which the internal values are desired. APL will return a vector representing the indices of these characters in the 256-element atomic vector (see Section IV). For example,

```
      ⎕AV⍳'1A≥.'
      2  12  69  93
```

The system function ⎕CV can be used to convert data from internal APL format to external formats compatible with other MPE subsystems, and from external formats to the internal APL format. The left argument of ⎕CV is a scalar value used as a control to specify the type of conversion, or a 256-element vector which is indexed by the right argument of ⎕AV to obtain a result. The right argument is the data to be converted; the result is the converted data.

EXTERNAL TO INTERNAL APL CONVERSION

The following values of the left argument (control) of ⎕CV produce the following external to internal APL conversions:

<u>control</u>

1 The right argument must be a character vector or unit or scalar character. The result is a character vector which is formed by treating the right argument characters as ASCII and performing an input conversion from external ASCII to internal APL. (See Appendix A for a conversion table.)

2 Converts every two characters in the right argument to a numeric value in the result using integer conversion. If the input vector is of odd length, the last byte is ignored.

3 Converts every four characters in the right argument to a numeric value in the result using double integer conversion. If between one and three bytes are left over at the end of the right argument, they are ignored.

4 Converts every four characters in the right argument to a numeric value in the result using real conversion. If between one and three bytes are left over at the end of the right argument, they are ignored.

5 Converts every eight characters in the right argument to a numeric value in the result using real conversion. If bytes are left over at the end of the right argument, they are ignored.

Note: An APL statement equivalent to 2 ⎕CV VEC is:

```
256⊥⍉((⌊0.5×⍴VEC),2)⍴((-⎕IO)+⎕AV⍳VEC)
```

INTERNAL APL TO EXTERNAL CONVERSION

The following values of the left argument (control) of ⎕CV produce the following internal APL to external conversions:

<u>control</u>

1 Converts the right argument, which must be characters, to external ASCII and returns a character vector result.

2 Converts each right argument element to two characters in the result. The right argument must be a numeric scalar, vector, or unit in which each element is an integer between ¯32768 and 32767, or a domain error will occur.

3 Converts each data value in the right argument to a four-character result. The right argument must be a scalar, unit, or vector numeric value in which each element is an integer between ¯2,147,483,648 and +2,147,483,647, or a domain error will occur.

4 Each data value in the right argument is converted to four characters in the result. The right argument must be numeric scalar, unit, or vector.

5 Each data value in the right argument is converted to eight bytes in the result. The right argument must be numeric scalar, unit, or vector.

In the last four of the above conversions, each result character is obtained by dividing the right-argument element into bytes and using these bytes as an index into ⎕AV. This is simulated in APL, for ¯2 ⎕CV, by

```
        ⎕AV [⎕IO+,⍉256 256⊤VEC]
```

where VEC is the right argument of ⎕CV.

If the right argument of ⎕CV is a 256-element vector (of any type), a translation is performed whereby each character in the right argument is used, essentially, as an index into the left argument. The result has the same shape as the right argument (which must be a character vector), and is the same type as the left argument.

In this mode, ⎕CV is equivalent to the APL expression

```
    leftarg[⎕AV⍳rightarg]
```

FUNCTION DEFINITION

SECTION VII

A user-defined function is a function written by a user to perform a specific computation. A user-defined function (or, more simply, a defined function) can be established in a workspace in one of four ways:

1. An existing defined function can be obtained from a stored workspace using the)LOAD,)COPY, or)PCOPY commands (see Section XI).

2. A defined function can be established with the ⎕FX system command.

3. A defined function can be created and saved using the APL\3000 editor.

4. A new defined function can be created by modifying an existing defined function with the APL\3000 editor.

Once established in a workspace, a defined function can be displayed or executed, modified using the APL\3000 editor (see Section VIII), stored in a saved workspace, or deleted (destroyed).

CANONICAL REPRESENTATION AND FUNCTION ESTABLISHMENT

A canonical representation is a character matrix which must satisfy the following requirements:

1. The first row of the matrix is the function header and must be in one of the forms described under the heading FUNCTION HEADER, below.

2. The remaining rows, if any, of the matrix constitute the function body, and may consist of any combinations of characters, except that there may be no blank rows.

The canonical representation of a defined function can be obtained by executing the ⎕CR system function, and the vector representation of a defined function can be obtained by executing the ⎕VR system function. A character vector argument containing the name of the function must be specified as the argument of ⎕CR and ⎕VR.

An example of ⎕CR is:

```
              TEST←⎕CR 'ROOTS'
              TEST
         ROOTS
         'ENTER A NUMBER'
         'AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT'
         'AND THE CUBE ROOT'
        LABEL1:N←⎕
        LABEL2:A←N*÷2
        LABEL3:B←N*÷3
         'THE SQUARE ROOT IS ',⍕A
         'THE CUBE ROOT IS ',⍕B
         'ENTER 0 IF YOU DO NOT WISH TO CONTINUE'
        LABEL4:N←⎕
         →(N≠0)/5
```

See Section IV for complete discussions of the ⎕CR and ⎕VR system functions.

If ROOTS is expunged with the ⎕EX system function (see Section IV), it is no longer available for use:

```
              ⎕EX 'ROOTS'
        1


              ROOTS
        VALUE ERROR
               ROOTS
               ↑
```

The function can be re-established by executing the ⎕FX system function with TEST (the variable to which the canonical representation of ROOTS had been assigned) as its argument:

```
              ⎕FX TEST
        ROOTS
```

The function ⎕FX produces as an explicit result a character vector representing the name of the function being fixed, while replacing any existing definition of the function with the same name. The function ROOTS now can be used again:

```
              ROOTS
        ENTER A NUMBER
        AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
        AND THE CUBE ROOT
        ⎕:
              125
        THE SQUARE ROOT IS 11.18033989
        THE CUBE ROOT IS 5
        ENTER 0 IF YOU DO NOT WISH TO CONTINUE
        ⎕:
              0
```

The expression ⎕FX n will establish a function if the following conditions are met:

* The argument n is a valid representation of a function. Any character vector or matrix which differs from a vector or canonical representation only in the addition of non-significant spaces (other than rows consisting of spaces only) is a valid representation.

* The name of the function to be established does not conflict with an existing use of the name for an executing or halted function, or for a label or variable.

If the expression ⎕FX n fails to establish a function, no change occurs in the workspace and the expression returns a scalar index of the row in the argument where the fault was found. See Section IV for a complete discussion of ⎕FX.

FUNCTION HEADER

A defined function may or may not return a result, and it may have one argument (monadic), two arguments (dyadic), or no arguments (niladic).

If the function header contains a specification (left) arrow, the function returns a result, and the name to the left of the arrow is the name used within the function to identify the result.

The valence of a defined function is defined as the number of arguments it takes. Thus, a defined function may have a valence of zero (no argument), one (one argument), or two (two arguments). This allows six possible header forms as follows:

TYPE	VALENCE	EXPLICIT RESULT	NO EXPLICIT RESULT
Dyadic	2	R←A FUNCTIONNAME B	A FUNCTIONNAME B
Monadic	1	R←FUNCTIONNAME B	FUNCTIONNAME B
Niladic	0	R←FUNCTIONNAME	FUNCTIONNAME

The name of a defined function is global (see LOCAL AND GLOBAL NAMES, below). The names used for arguments of a function are local to the function. Additional local names may be designated by listing them in the function header after the function name and argument name(s). These additional names must be separated from the function name and argument(s), and from one another, by semicolons. For example,

```
AREA←RADIUS CIRCLEAREA DEGREES;LOCAL1;LOCAL2
```

A name, except the function name itself, may not be repeated in the function header. Argument names used in the function header do not need to be used within the body of the function.

LOCAL AND GLOBAL NAMES

When a function is executed, it often is necessary to use intermediate results or temporary functions which have no significance outside the function. The use of names local to the function, so designated by their appearance in the function header, or by being used as labels, relieves the programmer of the requirement of keeping track of such transient names, and allows greater freedom in the choice of names (the same name can be used independently in several functions as long as it is local to its function).

The name of the function itself, and names used in the function body that are not designated as local by being included in the function header, are defined as global names. Global names have significance both inside and outside the function and may be referenced in the workspace (assuming that the function is established in the workspace). For example, the following function computes the areas of sectors of circles.

```
      ⎕CR 'CIRCLEAREA'
AREA←RADIUS CIRCLEAREA DEGREES;LOCAL1;LOCAL2
AREA←(○RADIUS*2)×DEGREES÷360
DIAMETER←RADIUS×2
⍝THIS IS A COMMENT
```

The names RADIUS and DEGREES are argument names defined as local by being included in the function header. The name CIRCLEAREA is the function name and is global. In addition, the name DIAMETER is global because it is included in the body of the function but not in the function header. The names CIRCLEAREA and DIAMETER, being global, can be referenced from the workspace outside the function. Note that names global to one function may be local to another calling function. Therefore local/global distinction is on a function-by-function basis.

```
      348 CIRCLEAREA 12.852
13582.40189
      DIAMETER
696
```

A local name may be the same as a global name, and any number of names local to different functions may be the same. During the execution of a defined function, a local name will temporarily exclude from use a global object of the same name. If the execution of a function is interrupted (leaving it either suspended or pendent; see Section X), the local objects retain their dominant position during the execution of subsequent APL operations, until such time as the halted function is completed.

The localization of names is dynamic, that is, a local name has no effect except when the defined function is being executed. When a defined function uses another defined function during its execution, a name local to the first (or outer) function continues to exclude global objects of the same names from the second (or inner) function. This means that a name localized in an outer function has the significance assigned to it in that function, but has no further localization in an inner function. The same name localized in a sequence of nested functions has the significance assigned to it at the innermost level of execution.

The shadowing of a name by localization is complete, in that once a name has been localized its global values are inaccessible, even if nothing is assigned to it during execution of the function in which it is localized.

BRANCHING AND LINE NUMBERS

Lines in a function are normally executed sequentially, from line 1 through the highest numbered line, and execution terminates at the end of the last line in the function. This normal order can be modified by branching. Branching is used in iterative procedures, in choosing one out of a number of possible lines, and in other situations where the normal order of line execution is not desired.

Lines in a function have reference numbers associated with them, starting with the number one for the first line in the function body (the function header is number zero), and continuing with successive integers. Thus, the statement →11 specifies a branch to the eleventh line in the function body. When the expression is executed, branching occurs and line number 11 is executed next, regardless of where the branch statement itself occurs. (The branch statement →11 may be in line 11, in which case an infinite loop may result until interrupted by an action from the terminal.)

A branch statement always starts with the branch (or right) arrow on the left, followed by any expression. For the statement to be effective, however, the expression must evaluate to an integer, to a vector whose first element is an integer, or to an empty vector. Any other value results in a DOMAIN or RANK error. If the expression evaluates to a valid result, then the following rules apply:

* If the result is an empty vector, the branch has no effect and the next statement in the function is executed. If there is no next statement (the branch is the last statement), the function terminates normally.

* If the expression evaluates to the number of a line in the function, that line is the next to be executed.

* If the result of the evaluation is a number out of the range of line numbers in the function, the function terminates. (The number 0 and all negative numbers are outside the range of line numbers for any function.)

Because zero is often a convenient result to compute, and it is never the number of a line in the body of a function, it is often used as a standard value for a branch intended to end the execution of a function.

An example of branching:

```
        ⎕CR 'ROOTS'
  ROOTS
  'ENTER A NUMBER'
  'AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT'
  'AND THE CUBE ROOT'
 LABEL1:N←⎕
 LABEL2:A←N*÷2
 LABEL3:B←N*÷3
  'THE SQUARE ROOT IS ',⍕A
  'THE CUBE ROOT IS ',⍕B
  'ENTER 0 IF YOU DO NOT WISH TO CONTINUE'
 LABEL4:N←⎕
  →(N≠0)/5←──────────────── Branch statement


        ROOTS
 ENTER A NUMBER
 AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
 AND THE CUBE ROOT
 ⎕:
       574
 THE SQUARE ROOT IS 23.9582971
 THE CUBE ROOT IS 8.310694107
 ENTER 0 IF YOU DO NOT WISH TO CONTINUE
 ⎕:
       0←──────────────── Terminates execution when 0 entered (does not branch)
```

The compression function in the form U/V (the statement →(N≠0)/5 above) gives V if U is equal to one (true), and an empty vector if U is equal to 0 (false). Thus, the statement →(N≠0)/5 in ROOTS is a branch statement which causes a branch to line 5 if the condition N≠0 is true, and a branch to an empty vector (normal sequence) when the condition is false. In this case, there is no next statement and the function terminates.

LABELS

If a line occurring in the body of a function is prefaced by a name and a colon, the name is assigned a value equal to the line number automatically upon function execution. A name used in this way is called a label. Labels are advantageous when it is expected that a function may be changed, because a label automatically assumes the new line number of its associated line as other lines are inserted or deleted.

The name of a label is local to the function in which it appears, and must be distinct from other label names and from local names in the function header.

A label name may not appear immediately to the left of a specification arrow. In effect a label acts like a local constant.

Examples of labels are:

```
          ⎕CR 'ROOTS'
      ROOTS
      'ENTER A NUMBER'
      'AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT'
      'AND THE CUBE ROOT'
     LABEL1:N←⎕
     LABEL2:A←N*÷2
     LABEL3:B←N*÷3
      'THE SQUARE ROOT IS ',⍕A
      'THE CUBE ROOT IS ',⍕B
      'ENTER 0 IF YOU DO NOT WISH TO CONTINUE'
     LABEL4:N←⎕
      →(N≠0)/5
```

Labels

COMMENTS

The symbol ⍝ signifies a comment. A comment is inserted in a function for informative purposes only, and is not executed. The symbol may occur anywhere within a line; however, everything to the right of the comment symbol in the line is ignored at execution. A comment may not be placed in the header line.

A comment example:

```
          ⎕CR 'CIRCLEAREA'
      AREA←RADIUS CIRCLEAREA DEGREES
      AREA←(○RADIUS*2)×DEGREES÷360
      DIAMETER←RADIUS×2
      ⍝THIS IS A COMMENT
```

APL\3000 EDITOR

SECTION VIII

The APL\3000 editor is used to create and modify APL or APLGOL functions and to create and modify one- or two-dimensional character data. The editor recognizes lines of input and operates on lines of text and on characters within these lines. Within the editor, both line numbers and a cursor to the line currently being edited are maintained, so that editing may specify line numbers or a line position relative to the cursor.

EDITOR FEATURES

* The editor retains instruction parameters from one edit instruction to the next, so that in successive applications of an edit instruction, the parameters often need not be respecified.

* Most edit instructions may be abbreviated.

* In the absence of specified parameters, default parameters are assumed.

* In all instructions which require that a line be specified (except ADD), the position of the cursor is assumed if an explicit line number is absent. If a line number is specified, it will be used and the cursor is adjusted to reflect the new current line. The instructions which are used to set patterns (DELTA, CURSOR, and so forth), may be used without parameters to determine the current parameter setting.

* In some instructions, a character string may be specified instead of a line number. In this case, the next line starting with the line in which the string is located is the selected line.

* Line numbers may range between 0.000 to 99999.999 for a maximum of 100,000 lines.

To access the editor, the system command)EDIT is entered, optionally followed by the name of an existing function or character variable to be edited. If a name is not specified, the editor immediately enters ADD mode, and new lines may be entered. If the name of an existing function or character variable is specified, the editor prompts with a "greater than" (>) symbol for an edit instruction.

An example of accessing the editor is as follows:

```
            )EDIT ROOTS ←———— Existing function specified
       APL FUNCTION
       >ADD
       [12]           ⍝ THIS IS A COMMENT
       [13]           return
       >END


            )EDIT ←———— Editor enters ADD mode when no existing function specified
       [0]            THIS IS LINE ZERO
       [1]            THIS IS LINE ONE
       [2]            LINE TWO
       [3]            THREE
       [4]            4
       [5]            5
       [6]            6
       [7]            return
       >
```

EDIT INSTRUCTION SYNTAX

Table 8-1 lists all edit instructions and shows the syntax and the abbreviation (where applicable) for each instruction.

Table 8-1. Edit Instructions

```
A[DD] ⎡linespec⎤  [delta]
      ⎣string  ⎦

B[RIEF]

C[HANGE]  [character [patternstring] character [changestring]
          character [rangelist]]

CO[PY]  lineblock

lineblock = linerange ⎧ :   ⎫linespec  [delta]
                      ⎨ ,   ⎬
                      ⎩blank⎭

⎧CU[RSOR]⎫ ⎡linespec ⎤
⎩*       ⎭ ⎢+ integer⎥
           ⎢- integer⎥
           ⎣string   ⎦

D[ELETE]  ⎡string   ⎤
          ⎣rangelist⎦

delta = [,] linenumber

⎧DELT[A]⎫ ⎡=⎤  [decimalnumber]
⎩   Δ   ⎭ ⎣←⎦
```

Table 8-1. Edit Instructions (continued)

```
END  [APL   ]
     [APLGOL]

FIND [string] [rangelist]

{H[ELP]  } [instruction]
{EXPLAIN }
{?       }

linerange = [linespec                                 ]
            [<linespec> <separator> <linespec>        ]
            [<linespec> <separator>                   ]
            [<separator> <linespec>                   ]
            [separator                                ]
            [ALL                                      ]

linespec = [line number]
           [FIRST      ]
           [LAST       ]
           [CURSOR     ]
           [*          ]

L[IST] [rangelist]
       [string   ]
       [ALL      ]
       [FIRST    ]
       [LAST     ]

LOCK  [APL   ]
      [APLGOL]

MAT[RIX]  [variablename]

M[ODIFY] [string   ]
         [rangelist]

QUIT

rangelist = [linerange  [,linerange]. . . [,linerange]]
            [rnge [,rlist]                            ]

R[EPLACE] [string   ]   [delta]
          [rangelist]

RES[EQUENCE]  lineblock

separator = [/]
            [|]
```

Table 8-1. Edit Instructions (continued)

```
string = <character> <text not containing character> <character>

UNDO  [integer] [grainspec]

grainspec = { ,     } {L[INES]    }
            { ;     } {C[OMMANDS] }
            { blank }

VEC[TOR]  [variablename]

VER[BOSE]
```

EDIT INSTRUCTIONS

ADD INSTRUCTION

The form of the ADD instruction is

```
A[DD] [linespec]  [delta]
      [string  ]
```

The ADD instruction places the editor into a mode to accept new lines of input. If parameters are not specified, the text is added to the end of the edit file using the present value of delta to increment the line numbers. If linespec is specified, the text is added starting with the specified line and thereafter increasing the line number by the delta specified, or by the default delta supplied by the system (the initial default value is one). If the line number specified already exists, the text is added following that line by applying the proper delta. If this is not possible, an error is reported. A null line, that is, a line with just a carriage return, terminates the ADD instruction. The system retains a delta value, initially set to one, which is updated by any edit instruction specifying a delta parameter. The delta value can be specified once, therefore, and retained as long as necessary without further respecification. When there is no more room to add lines using the present delta, the system divides the delta by 10 so that more lines can be added. This is repeated until delta becomes .001.

BRIEF INSTRUCTION

The form of the BRIEF instruction is

```
B[RIEF]
```

The BRIEF instruction is used to set the editor response mode to brief, in which case messages are either shortened or are omitted. The opposite setting of the instruction response mode is VERBOSE (the default mode).

CHANGE INSTRUCTION

The form of the CHANGE instruction is

```
    C[HANGE] [character [patternstring] character [changestring]
             character [rangelist]
```

The CHANGE instruction is used to change one pattern within a range of lines to another pattern (which may be null). If rangelist is not specified, the current line is assumed. If both patterns are omitted, whatever patterns were most recently associated with a CHANGE instruction are used again. If a single pattern is specified, it becomes the new change pattern and the former search pattern is retained. If both patterns are specified, the first string (patternstring) is a search pattern and the second string (changestring) is the change pattern.

An example of the CHANGE instruction is shown below:

```
          >LIST 0
          [0]          THIS IS LINE ZERO
          >CHANGE 'ZERO'0' 0
          [0]          THIS IS LINE 0
          >LIST 0
          [0]          THIS IS LINE 0
          >
```

COPY INSTRUCTION

The form of the COPY instruction is

```
    CO[PY]  lineblock
```

where

$$\text{lineblock} = \text{linerange} \begin{Bmatrix} : \\ ; \\ \text{blank} \end{Bmatrix} \text{linespec} \quad [\text{delta}]$$

The COPY instruction is used to duplicate one or more lines of text elsewhere in the text. This instruction requires the specification of a linerange to be copied and a linespec to specify the target point for copying. It is not possible to delete existing lines within the COPY instruction by overlaying a copied line number on top of an existing line.

An example of the COPY instruction is shown below:

```
          >COPY 2 7
          [2] => [7]
          >LIST 7
          [7] LINE TWO
```

CURSOR INSTRUCTION

The form of the CURSOR instruction is

```
{CU[RSOR]} [linespec ]
{*       } [+ integer]
           [- integer]
           [string   ]
```

The CURSOR instruction is used wither to indicate the current position of the cursor or to reposition it. To find the current cursor position, either the word CURSOR or the symbol * may be entered without other parameters. The addition of parameters to the CURSOR instruction causes the cursor to be relocated according to the parameters. If linespec is specified, the cursor moves to the specified line; if string is specified, the cursor moves to the next line beyond the present cursor position in which the string is located. The + integer and - integer parameters move the cursor forward or backward in the text relative to the present cursor location.

An example of the CURSOR instruction is shown below:

```
>CURSOR
CURSOR = [7]
>*
CURSOR = [7]
>CURSOR 0
WAS [7]
>
```

DELETE INSTRUCTION

The form of the DELETE instruction is

```
D[ELETE] [string   ]
         [rangelist]
```

The DELETE instruction is used to delete lines from the text. If no parameters are specified, the line currently indicated by the cursor position is deleted. If string is specified, the next line in which the string occurs is deleted. If rangelist is specified, the lines in the list are deleted.

Note: In VERBOSE mode each line is printed as it is deleted.

An example of the DELETE instruction is shown below:

```
>DELETE 7
[7]          LINE TWO
>
```

DELTA INSTRUCTION

The form of the DELTA instruction is

$$\left\{\begin{matrix}\text{DELT[A]}\\ \Delta\end{matrix}\right\} \left\{\begin{matrix}=\\ \leftarrow\end{matrix}\right\} \text{[}\underline{\text{decimalnumber}}\text{]}$$

The DELTA instruction is used to set the increment value for adding, replacing, copying, and resequencing lines. The default value of delta is one. The optional specification of a delta in the COPY, RESEQUENCE, ADD, and REPLACE instructions automatically changes the value of the default delta.

An example of the DELTA instruction is shown below:

```
>DELTA=.1
WAS [1]
>ADD
[6.1]          THE INCREMENT IS .1
[6.2]
>
```

END INSTRUCTION

The form of the END instruction is

$$\text{END}\ \begin{bmatrix}\text{APL}\\ \text{APLGOL}\end{bmatrix}$$

The purpose of the END instruction is to terminate editing and to translate the text into internal APL or APLGOL form suitable as a function for execution by APL. If a former version of the function existed, the new version now replaces the former. If, in translation to the internal form, errors are discovered which make it impossible to create a new internal form, an indication of the error and a listing of the line in which the error was found are displayed, and the system retains the internal form as well as the text in the editor for further editing. If the error is hard to correct, either the MATRIX or VECTOR instruction may be used to save the text as a character matrix or vector for later editing.

The optional APL and APLGOL parameters specify the particular translator to be used (that is, the kind of function being edited).

EXPLAIN INSTRUCTION

See the HELP instruction below.

FIND INSTRUCTION

The form of the FIND instruction is

F[IND] [<u>string</u>] [<u>rangelist</u>]

The FIND instruction is used to locate the line containing the next occurrence of the string starting with the cursor position. If the string is not specified, the search string from the last FIND instruction is used. The rangelist parameter may be used to limit the search.

An example of the FIND instruction is shown below:

```
>FIND 'LINE TWO'
[2]          LINE TWO
```

HELP INSTRUCTION

The form of the HELP instruction is

```
(H[ELP] )     [instruction]
{EXPLAIN}
(?      )
```

The HELP instruction lists permissible edit instructions. If followed by an instruction parameter, a brief explanation of that particular instruction is provided.

Examples:

```
>HELP
THE EDIT COMMANDS ARE: ADD, BRIEF, CHANGE, COPY, CURSOR, DELETE,
  DELTA, END, FIND, HELP, LIST, LOCK, MATRIX, MODIFY, QUIT,
  REPLACE, RESEQUENCE, UNDO, VECTOR, AND VERBOSE.
  TO OBTAIN FURTHER DATA ON ANY OF THESE COMMANDS, ENTER
  'HELP' FOLLOWED BY THE COMMAND NAME.
>HELP MATRIX
  THE MAT [RIX] COMMAND IS USED TO CREATE A CHARACTER MATRIX FROM
  THE TEXT IN THE EDIT BUFFER. THE CHARACTER VARIABLE MAY THEN BE
  USED AS DATA WITHIN THE SYSTEM OR LATER TURNED INTO A PROCEDURE
  MATRIX WITHOUT A NAME WILL STORE THE DATA IN THE VARIABLE WHICH
  WAS EDITED (IF ANY), MATRIX <VARIABLE NAME> WILL STORE IN THE
  SPECIFIED NAME.  (SEE VECTOR)
>
```

LIST INSTRUCTION

The form of the LIST instruction is

```
L[IST]  [rangelist]
        |string   |
        |ALL      |
        |FIRST    |
        [LAST     ]
```

The LIST instruction is used to print lines. If a parameter is not specified, the line currently indicated by the cursor is listed. If string is specified, the next line starting with the line in which the string occurs is listed. If rangelist is specified, lines in the list are listed.

An example of the LIST instruction is shown below:

```
        >LIST ALL
        [0]          THIS IS LINE 0
        [1]          THIS IS LINE ONE
        [2]          LINE TWO
        [3]          THREE
        [4]          4
        [5]          5
        [6]          6
        [6.1]        THE INCREMENT IS .1
```

LOCK INSTRUCTION

The form of the LOCK instruction is

```
    LOCK [APL   ]
         [APLGOL]
```

The LOCK instruction is similar to the END instruction, in that it is used to terminate the editing of a function and have the function translated into internal APL or APLGOL form for execution. If the translation is successful, however, the function then is marked as locked, and it is not possible thereafter for the function to be unlocked, edited, or read.

MATRIX INSTRUCTION

The form of the MATRIX instruction is

```
    MAT[RIX]  [variablename]
```

The MATRIX instruction stores the edit text as a character matrix with rows sufficiently long to contain the longest text line. The variable may be edited later and a function or other variable produced. If <u>variablename</u> is omitted, the name of the function or variable used in the)EDIT command is replaced by the character matrix.

MODIFY INSTRUCTION

The form of the MODIFY instruction is

```
    M[ODIFY]  [string   ]
              [rangelist]
```

The MODIFY instruction is used to modify the contents of a line or range of lines, depending on the parameter specified. If no parameter is specified, the line currently indicated by the position of the cursor is modified; if <u>string</u> is specified, the next line starting with the line in which the string is located is modified; if <u>rangelist</u> is specified, lines in the list are modified, one at a time.

When a line is to be modified, the line number is printed, followed by the line, after which special modification characters may be used as sub-editing instructions to alter the contents of the line. A sub-editing template line is created by spacing out under the line to the point where the sub-editing is to be done and then entering the appropriate single character instruction, possibly followed by replacement or insertion text. When this is done, the edited line is printed again to reflect the modifications, and further modifications can be entered. A null line (signified by just a carriage return) terminates the modification process.

MODIFICATION INSTRUCTION	MEANING
D	Delete the above character.
R	Starting at the above position, replace the following text.
I	Starting immediately before the above position, insert the following text.
/	Delete entire line.

Note: A string of delete (D) characters may be followed by a single insertion (I) or replacement (R) character, followed by the insertion/deletion text; otherwise only one action may be specified per modification template line.

An example of the MODIFY instruction is shown below:

```
>MODIFY 1
[1]
THIS IS LINE ONE
             DDDI1
THIS IS LINE 1
```

QUIT INSTRUCTION

The form of the QUIT instruction is

QUIT

The QUIT instruction terminates all editing, deletes any text being edited, and returns to immediate execution mode in the APL system. Note that a function is not changed if the QUIT instruction is performed.

REPLACE INSTRUCTION

The form of the REPLACE instruction is

```
R[EPLACE] [string   ]   [delta]
          [rangelist]
```

The REPLACE instruction is used to replace one or more lines, depending on the parameters specified. If no parameters are specified, the line currently indicated by the cursor is replaced. If string is specified, the next line containing the string is replaced. If rangelist is specified, each line in the list is replaced. In replacing a line, the current line is listed, and the replacement line may then be entered. Once the rangelist is exhausted, the editor switches to the ADD mode, so that lines may be replaced and immediately followed with new lines without having to use multiple instructions. The optional delta specification is used for the ADD mode incrementing. Entering a null line (carriage return) terminates the process.

An example of the REPLACE instruction is shown below:

```
>REPLACE 1,5
[1]         THIS IS LINE 1
[1]         THIS IS A NEW LINE 1
[5]         5
[5]         THIS IS THE NEW LINE 5
[5.1]
>LIST ALL
[0]         THIS IS LINE 0
[1]         THIS IS A NEW LINE 1
[2]         LINE TWO
[3]         THREE
[4]         4
[5]         THIS IS THE NEW LINE 5
[6]         6
[7]         LINE TWO
[7.1]       THE INCREMENT IS .1
>
```

RESEQUENCE INSTRUCTION

The form of the RESEQUENCE instruction is

    RES[EQUENCE] lineblock

The RESEQUENCE instruction is used either to resequence portions or all of a function or data, or to rearrange lines of the function or data to appear elsewhere, thus in effect acting as a move instruction. It is not possible to overlay existing lines with resequenced line using the RESEQUENCE instruction.

An example of the RESEQEUNCE instruction is shown below:

```
          >RESEQUENCE 0,.5
          [0]=>[0.5]
          >LIST ALL
          [0.5]         THIS IS LINE 0
          [1]           THIS IS A NEW LINE 1
          [2]           LINE TWO
          [3]           THREE
          [4]           4
          [5]           THIS IS THE NEW LINE 5
          [6]           6
          [7]           LINE TWO
          [7.1]         THE INCREMENT IS .1
```

UNDO INSTRUCTION

The form of the UNDO instruction is

UNDO [integer] [grainspec]

The UNDO instruction negates the effect of the last command, that is, it "undoes" a command. UNDO affects ADD, CHANGE, DELETE, COPY, MODIFY, REPLACE, and RESEQUENCE (note that this does not include UNDO itself).

The grainspec parameter specifies whether to UNDO on a line-by- line [LINES] basis, or on a command-by-command [COMMANDS] basis. The default is LINES. The integer parameter specifies how many "grains" to UNDO, that is, how many LINES or COMMANDS. The default is one.

VECTOR INSTRUCTION

The form of the VECTOR instruction is

VEC[TOR] [variablename]

The VECTOR instruction stores the edited text as a character vector with carriage return characters used to separate the lines. The variable may be edited later and a function or other variable produced. If variablename is omitted, the name of the function or variable used in the)EDIT command is replaced by the character vector.

VERBOSE INSTRUCTION

The form of the VERBOSE instruction is

VER[BOSE]

The VERBOSE instruction is used to set the editor response mode to verbose, in which case messages regarding the effect of instructions are fully printed. The opposite setting of the instruction response mode is BRIEF. The default mode is VERBOSE.

APLGOL

APLGOL is a language which is a superset of APL, adding additional statement-sequence control structures. A workspace may contain any mixture of APL and APLGOL functions, which can be used in any combination. A single function, however, must be all APL or all APLGOL; the two languages may not be mixed within the same function.

In APLGOL, keywords are used in conjunction with APL expressions (except APL branch expressions, which cannot be used in APLGOL) to describe the control flow within a given procedure. For example, the APL procedure

```
Z←FACT N
→(~N≤1)/L
Z←1
→0
L:Z←N×FACT N-1
```

is comparable to the APLGOL procedure

```
PROCEDURE Z←FACT N
  IF N≤1 THEN
    Z←1
  ELSE
    Z←N×FACT N-1;
END PROCEDURE
```

APLGOL keywords are formed from an alphabetic string.

The external attributes of an APLGOL function are the same as those of an APL function; it is named according to the same rules as APL functions and has an optional result; zero, one, or two arguments; and zero or more local variables.

The <u>header</u> line of an APLGOL function is similar to an APL header except it is preceded by the keyword PROCEDURE, and terminated with a semicolon. The list of local variables, if any, is separated by commas instead of semicolons. For example,

```
PROCEDURE Z←L FUNC R,L1,L2,⎕IO;
```

defines an APLGOL function header equivalent to the APL function header

```
Z←L FUNC R;L1;L2;⎕IO
```

GENERAL APLGOL FUNCTION FORMAT

In addition to the header, an APLGOL function is composed of one or more statements followed by END PROCEDURE. Statements are written in free-field format and are terminated by semicolons.

APLGOL comments are placed between paired comment symbols (⍝), while in APL a comment is defined as anything on a line to the right of the leftmost comment symbol.

APLGOL functions are written in a free-field format, while APL functions are line-oriented. APLGOL statements may be entered in any convenient format. When the function is subsequently edited, the listing will be formatted to show a canonic form with indenting used to depict the depth and shape of the nested control structures.

For example, an APLGOL procedure could be entered as:

```
PROCEDURE SAMPL; IF A≠B THEN BEGIN A←C; WHILE J≥⌊(N-1+I)÷2
DO BEGIN L2←L3←L4L-J-1;L4←-L-1↓ρY←77777 DYADF L; END; END; ELSE;
A←D; IF 2=ρρZ DO EXIT C[2]←(1↓ρZ)-N; Z←N,C,N,P,Q,R; END
PROCEDURE
```

while subsequent editing would show it as:

```
[0]          PROCEDURE SAMPL
[1]            IF A≠B THEN
[2]              BEGIN
[3]                A←C;
[4]                WHILE J≥⌊(N-1-I)÷2 DO
[5]                  BEGIN
[6]                    L2←L3←L4L-J-1;
[7]                    L4←-L-1↓ρY←7 DYADF L;
[8]                  END;
[9]              END
[10]           ELSE
[11]             A←D;
[12]           IF 2=ρρZ DO
[13]             EXIT C[2]←(1↓ρZ)÷N;
[14]           Z←N,C,P,Q,R;
[15]         END PROCEDURE
```

Table 9-1 lists the syntax for all APLGOL statements.

Table 9-1. APLGOL Syntax

```
aplgol function = PROCEDURE header ; statement list
                       END PROCEDURE

header = [identifier  ] identifier [identifier]
            [identifier]. . . [identifier]

statement list = [statement]  [statement list]

statement = expression

              NULL

              EXIT  [expression]

              BEGIN  statement list  END

              HALT  [expression]

              FOREVER DO statement

              ASSERT expression : expression

              IF expression DO statement

              IF expression THEN statement ELSE
              statement

              WHILE expression DO statement

              REPEAT statement list UNTIL expression

              CASE expression OF integer constant
              BEGIN subcase list + END CASE

branch =   ┌BRANCH ┐
           │LEAVE  │
           │ITERATE│
           └RESTART┘

control   =┌PROCEDURE┐
           │FOREVER  │
           │IF       │
           │WHILE    │
           │REPEAT   │
           └CASE     ┘

subcase ::= subcase label : statement

subcase label    =┌integer scalar constant┐
                  └integer vector constant┘
```

Table 9-1. APLGOL Syntax (continued)

```
     subcase list = subcase [subcase list]
                        DEFAULT

      comment    =  lamp symbol  [text not containing a lamp symbol]
                   lamp symbol

 Note:  Comments may appear anywhere except in the middle of a
        vector constant, within a keyword, or within an identifier.
```

APLGOL STATEMENTS

NULL STATEMENT

The form of the NULL statement is

    NULL

NULL is a no-operation statement. It is used when a dummy statement is needed to complete a control structure but when no other action is necessary.

EXIT STATEMENT

The form of the EXIT statement is

    EXIT [expression]

The EXIT statement is used to return from the current procedure. If the optional expression is specified, the expression is executed just prior to returning.

BEGIN STATEMENT

The form of the BEGIN statement is

    BEGIN statement list END

The BEGIN statement is the usual compound statement which is used to group multiple statements, so that they can be treated as a single statement within the control structure. Note that a BEGIN/END pair does not constitute a block as in ALGOL (permitting a new name scope); local variables may only be specified in a function header line.

An example:

```
[0]        IF KLARN ≤ 6 DO
[1]          BEGIN
[2]            'ARGGH: KLARN IS BELOW SEVEN, NAMELY, ',⍕ KLARN;
[3]            EXIT;
[4]          END;
```

HALT STATEMENT

The form of the HALT statement is

HALT [expression]

When a HALT statement is encountered, execution is suspended and the system enters immediate execution mode. If the optional expression is specified, it is evaluated just prior to the suspension. If a HALT statement is used in place of a call to an unwritten module, the expression can be used to print a message that the particular procedure has reached this point before suspending. At this point, it is possible to simulate the effect of the missing module before continuing further execution.

For example, a compiler system control routine might be started as:

```
[0]        PROCEDURE COMPILE
[1]          FOREVER DO
[2]            BEGIN                    ⍝ LOOP TO PROCESS EACH INPUT ⍝
[3]              SCANNER;               ⍝ INVOKE THE SCANNER MODULE ⍝
[4]              PARSER;                ⍝ INVOKE THE PARSER MODULE ⍝
[5]              HALT 'INTERPRETER';    ⍝ NO INTERPRETER YET ⍝
[6]            END;
[7]        END PROCEDURE
```

When line [5] is executed, the text INTERPRETER is printed and execution is suspended.

ASSERT STATEMENT

The form of the ASSERT statement is

ASSERT expression : expression

The ASSERT statement is intended as an aid in the proof-of-correctness programming approach. The ASSERT statement allows the programmer to make assertions regarding the program which the system may optionally test. The second expression in the statement is a boolean expression giving a scalar (unit) truth value for the assertion. For example, if the variable I must lie between 0 and 9 inclusively, the assertion would be:

```
ASSERT 10: (I≥0)∧I≤9;
```

which would evaluate to a 1 if true and a 0 if false.

The first expression is used to give the relative importance of the assertion and must evaluate to an integer between ¯32768 and 32767. For example, a value of 1 would indicate a trivial assertion, while a value of 10 would indicate a less trivial one and a value of 100 would indicate a major assertion.

The actual mechanics of executing ASSERT statements depends on the system variable ⎕AL, which contains the current assertion checking level. This variable indicates the lower bound of assertions to be checked and has an integer range between ¯32768 and 32767. Each time an ASSERT statement is encountered, the assertion level is checked against the first expression in the statement. If the assertion level is smaller than the system variable the statement is regarded as a comment and not executed.

If the first <u>expression</u> is larger than or equal to the assertion level, however, the second <u>expression</u> is evaluated. If the result of the evaluation is true, the program continues; otherwise execution is suspended, and an ASSERTION FAILED message is printed together with information to locate the assertion in the procedure. At this point the system suspends execution to allow the user to correct the situation.

If the assertion level is lower than the lowest specified level, all assertions are checked. An example of assertion usage might be: a program may be debugged initially with the assertion level set low to check all assertions. When the assertions no longer fail, the assertion level may be raised to the highest-valued assertion in the program, so that only the most major assertions are checked. Should a malfunction subsequently occur in a program assumed to be checked out, the assertion level can again be lowered to check all of the original assertions again. Assertion statements remain as comments in a completed program and are intended to be useful documentation and debugging aids.

IF STATEMENT

APLGOL has two separate forms of IF statements. The single-arm <u>conditional</u> evaluates the <u>expression</u> after the IF, and if it is true, executes the <u>statement</u> following the DO. The form of the single-arm conditional IF statement is

IF <u>expression</u> DO <u>statement</u>

For example,

```
        IF A>5 DO
          B←A|5;
```

The <u>double-arm conditional</u> evaluates the <u>expression</u> after the IF and executes the <u>statement</u> following the THEN if it is true; otherwise it executes the <u>statement</u> following the ELSE. The form of the double-arm conditional IF statement is

IF <u>expression</u> THEN <u>statement</u> ELSE <u>statement</u>

For example,

```
        IF A>5 THEN
         A←C÷5
        ELSE
          A←A+1;
```

Note that the expression must evaluate to a boolean (0 or 1) scalar, unit, or vector result. If the expression evaluates to a multi-element vector, an implicit 1W expression is performed to select the first element.

WHILE STATEMENT

The form of the WHILE statement is

    WHILE expression DO statement

The WHILE statement first evaluates the expression which must evaluate to a boolean scalar, vector, or unit result. If the first element of expression is true, the statement is performed and the process is started over with the re-computation of the expression. Otherwise, control proceeds to the next statement.

REPEAT STATEMENT

The form of the REPEAT statement is

    REPEAT statement list UNTIL expression

The WHILE statement is termed a pre-checked loop; the REPEAT statement is referred to as a post-checked loop. A post-checked loop means that the statement list is performed at least once, after which the expression following the UNTIL is evaluated and checked. If the first element of this expression, which must evaluate to a boolean scalar, vector, or unit, is false, control will continue with the next statement; otherwise control returns to the first statement in the statement list following the REPEAT. Note that several statements may be contained between the REPEAT and the UNTIL, since this keyword pair forms a natural block, whereas in the WHILE statement a BEGIN/END must be used to specify the statement list.

FOREVER DO STATEMENT

The form of the FOREVER DO statement is

    FOREVER DO statement

The FOREVER DO statement causes statement to execute endlessly. In order to exit the scope of the FOREVER statement a special EXIT or branch statement is required. A FOREVER DO may be interrupted by generating a hard or soft terminal interrupt.

BRANCH STATEMENTS

APLGOL branch statements are of the form

    branch : [control]+

The only branch statements permitted in APLGOL are those directed to a key point in a control structure which encloses the point in which the branch is located. Three key points, termed LEAVE, ITERATE, and RESTART, are associated with each of the following control structures: PROCEDURE, FOREVER, IF, WHILE, REPEAT, and CASE.

Each branch statement consists of a keyword specifying the type of branch, followed by a colon and a list of control structure keywords which is processed left-to-right. Each element in the list specifies a control structure in which the branch statement is located, and each successive control structure is exited until the last one in the list. Control is then transferred to the appropriate point in the outermost control structure shown in the list. The nesting is defined by the lexical structure of the function, not the run-time execution structure. For example, LEAVE: WHILE will effect a branch to the leave point in the innermost WHILE statement relative to the location of the LEAVE statement.

Examples:

```
    RESTART: FOREVER FOREVER;
```

results in leaving the innermost FOREVER statement and branching to the restart point of the next innermost FOREVER statement.

```
     ITERATE: WHILE REPEAT;
```

exits the current inner WHILE statement and branch to the iterate point in the next innermost REPEAT statement.

The LEAVE, ITERATE, and RESTART points are defined on the flowcharts at the end of this section.

CASE STATEMENT

The form of the CASE statement is

CASE <u>expression</u> OF <u>integer constant</u> BEGIN
    <u>subcase list</u> + END CASE

The CASE statement uses the value of the <u>expression</u> following CASE to select one of the <u>subcases</u> and execute it. The expression must evaluate to a non-negative integer. If the value is non-single, the value of the first element is used. The value must be between 0 and the value of the <u>integer constant</u> following OF. The <u>integer constant</u> indicates the largest number for a <u>subcase</u> in the statement, although not all subcases need be specified. A single subcase may be associated with more than one value of the expression.

Note that no more than 1024 subcases (numbered 0 through 1023) are permitted.

The case body is delimited by BEGIN and END CASE. Inside it are the subcases, in any order. The syntax of a subcase is as follows:

    <u>subcase</u> = <u>subcase label</u> : <u>statement</u>
    <u>subcase list</u> = <u>subcase</u> [<u>subcase list</u>]

The <u>subcase label</u> can be a constant integer scalar, or a constant integer vector, in which case the associated <u>statement</u> will be executed if the value of <u>expression</u> following CASE is an element of the subcase label. The <u>subcase label</u> can also contain the keyword DEFAULT, in which case the accompanying statement will be executed if the value of the selector expression is in range but does not match any of the specified values in the other subcase labels. Only one DEFAULT subcase may be permitted in a case statement.

For example:

```
CASE I|J OF 15
  BEGIN
0:  I←J←13÷K;
2:  NULL;
1:  HALT 'CASE 1 IS SYSTEM ERROR';
10 12 14:
    BEGIN
      I←I-1;
      J←J-1;
    END;
5:  EXIT J←J-1;
DEFAULT:
    HALT 'UNKNOWN CASE POSSIBILITIES';
  END CASE;
```

The flow diagrams contained in figures 9-1 through 9-7 show the flow of control for each of the APLGOL statements. The key branch points of each statement structure associated with the three types of branches are indicated by IT, the <u>iterate</u> point, RS, the <u>restart</u> point, and LV, the <u>leave</u> point.

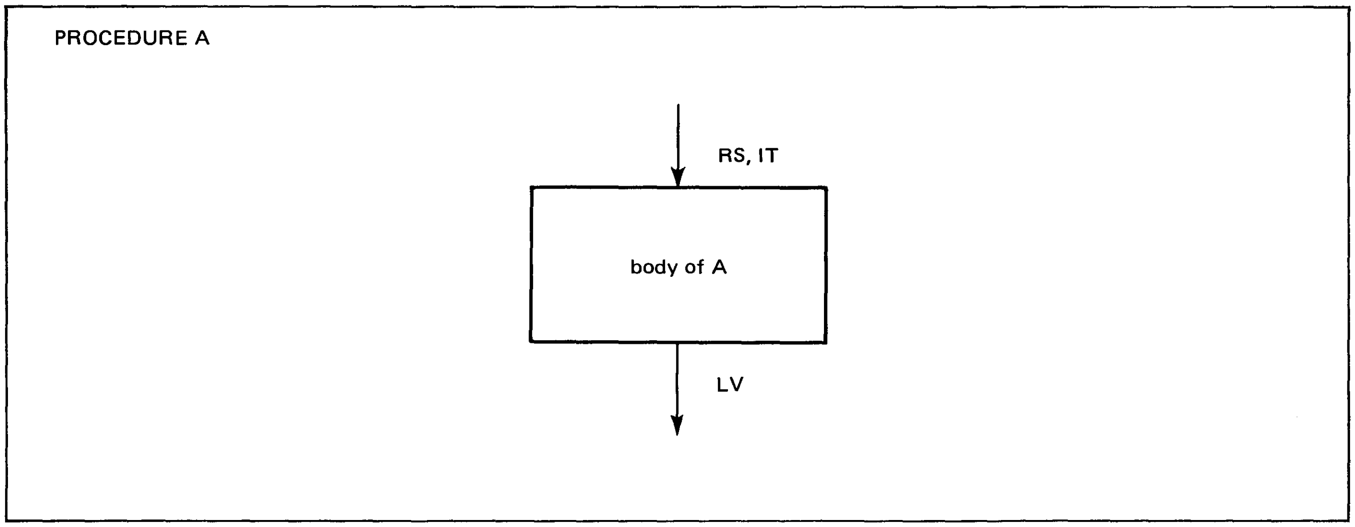


Figure 9-1. Procedure Statement Flow Chart

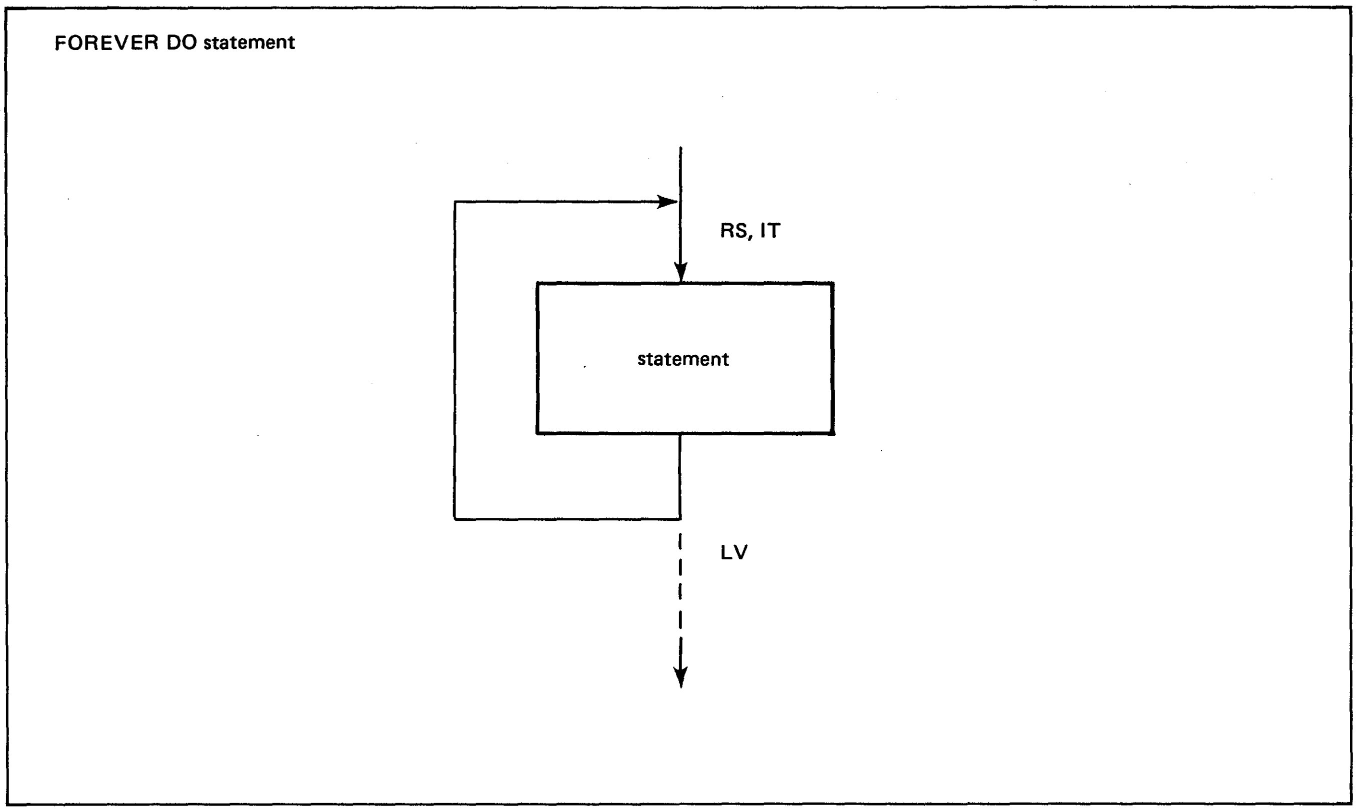


Figure 9-2. Forever Do Statement Flow Chart

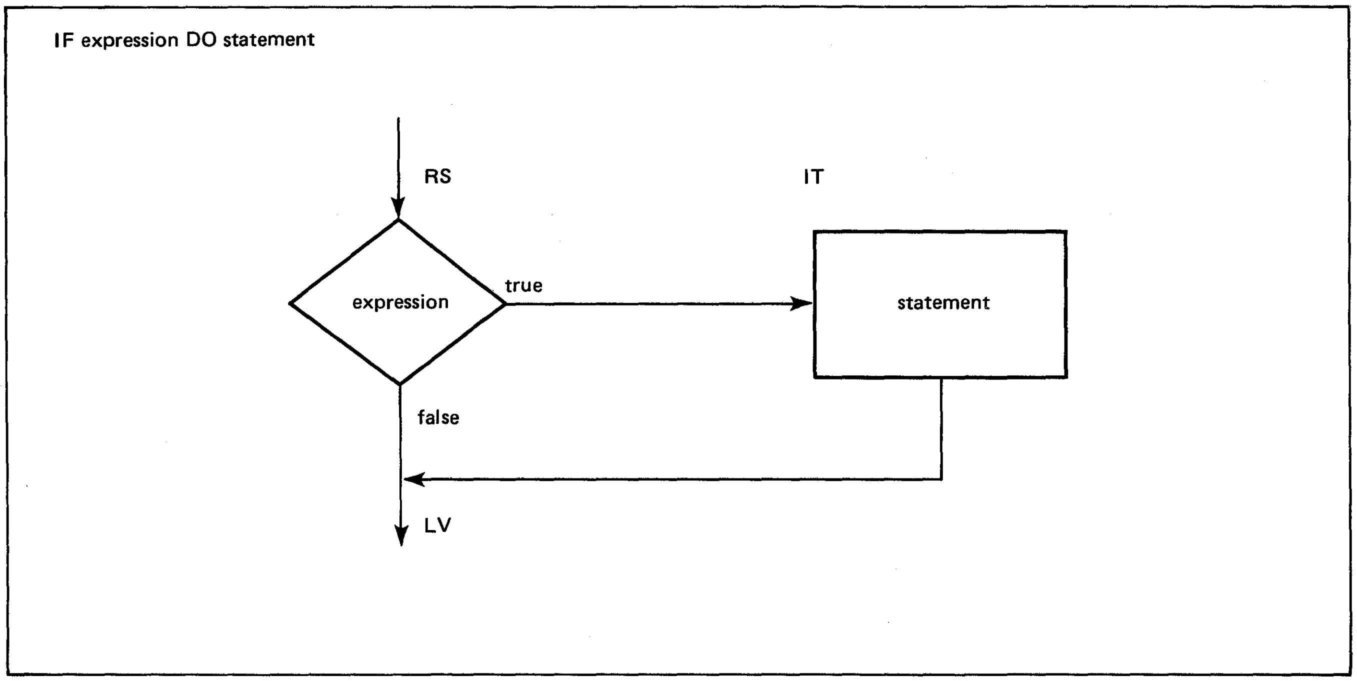


Figure 9-3. Single-Arm Conditional If Statement Flow Chart

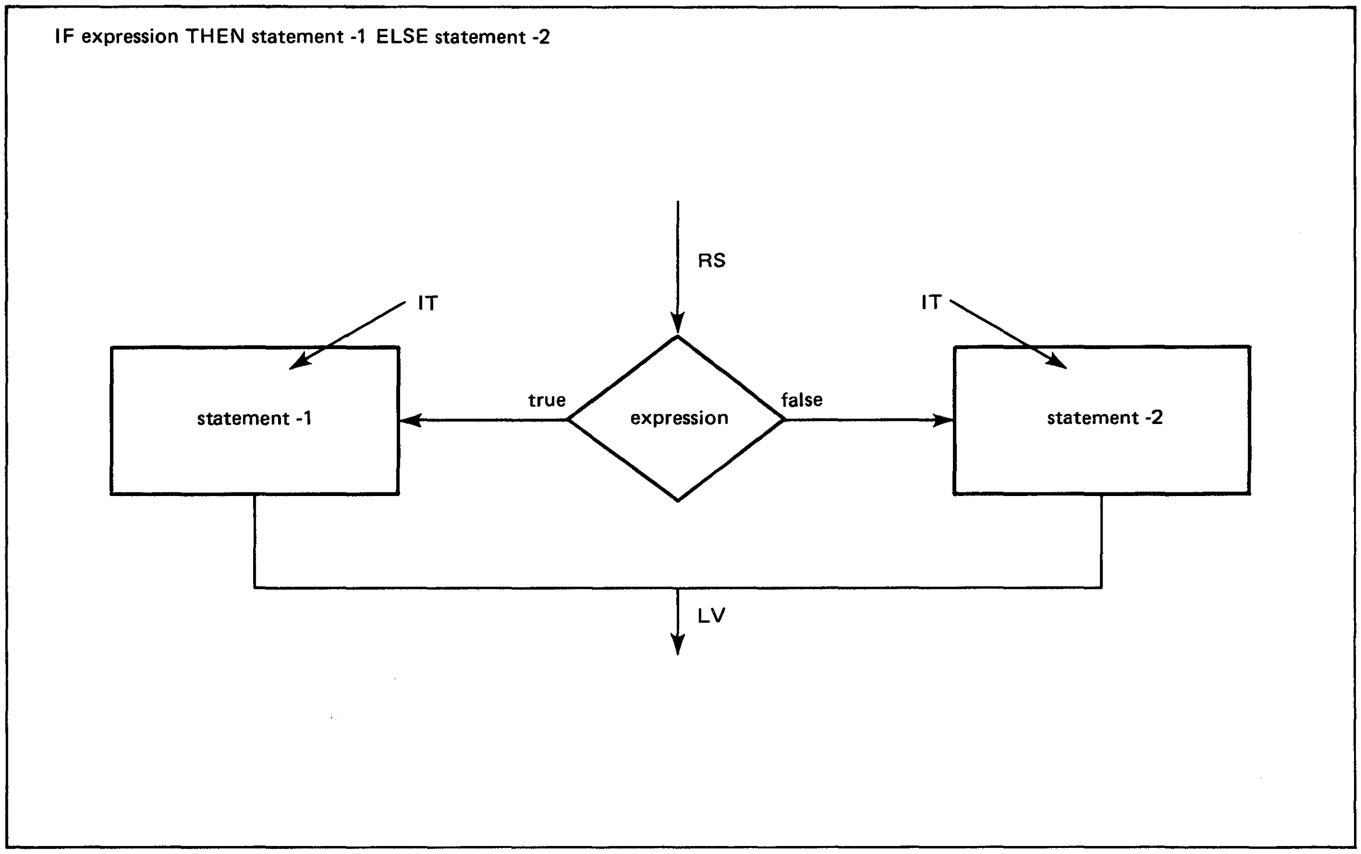


Figure 9-4. Double-Arm Conditional If Statement Flow Chart

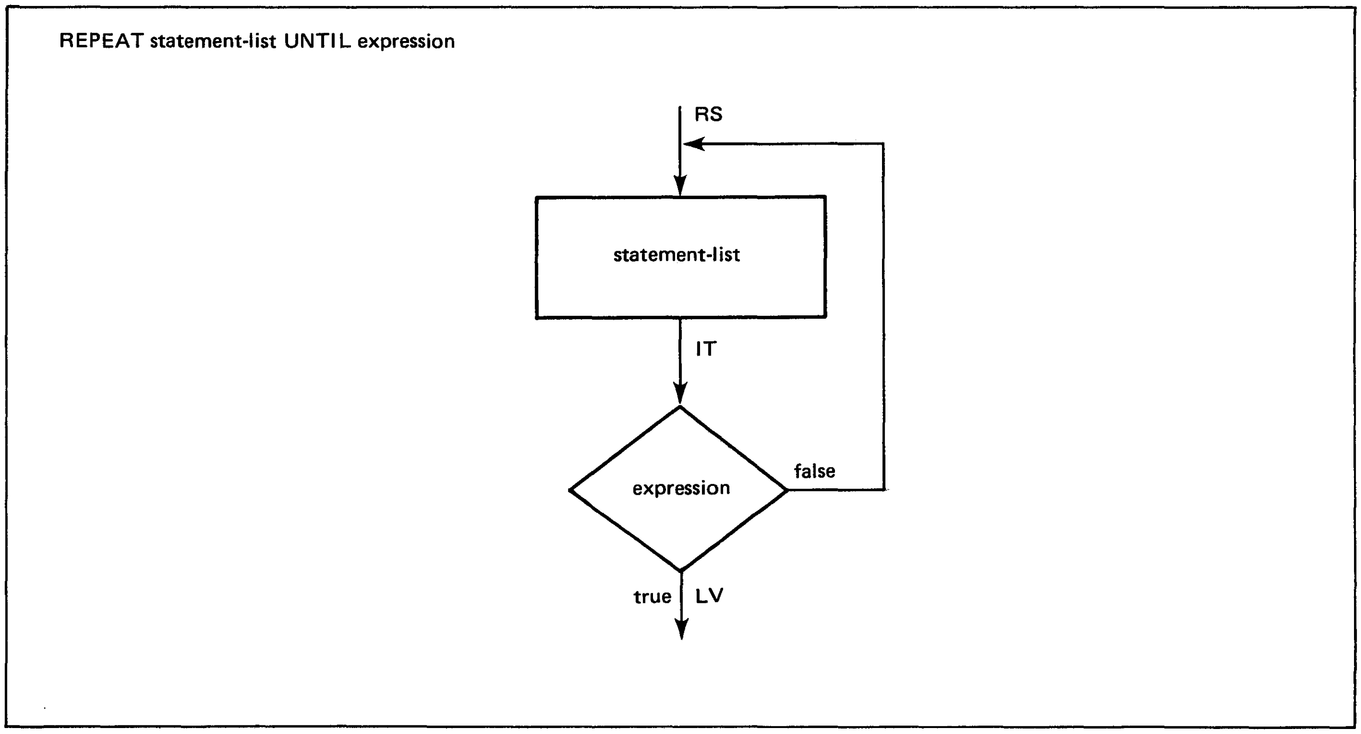


Figure 9-5. Repeat Statement Flow Chart

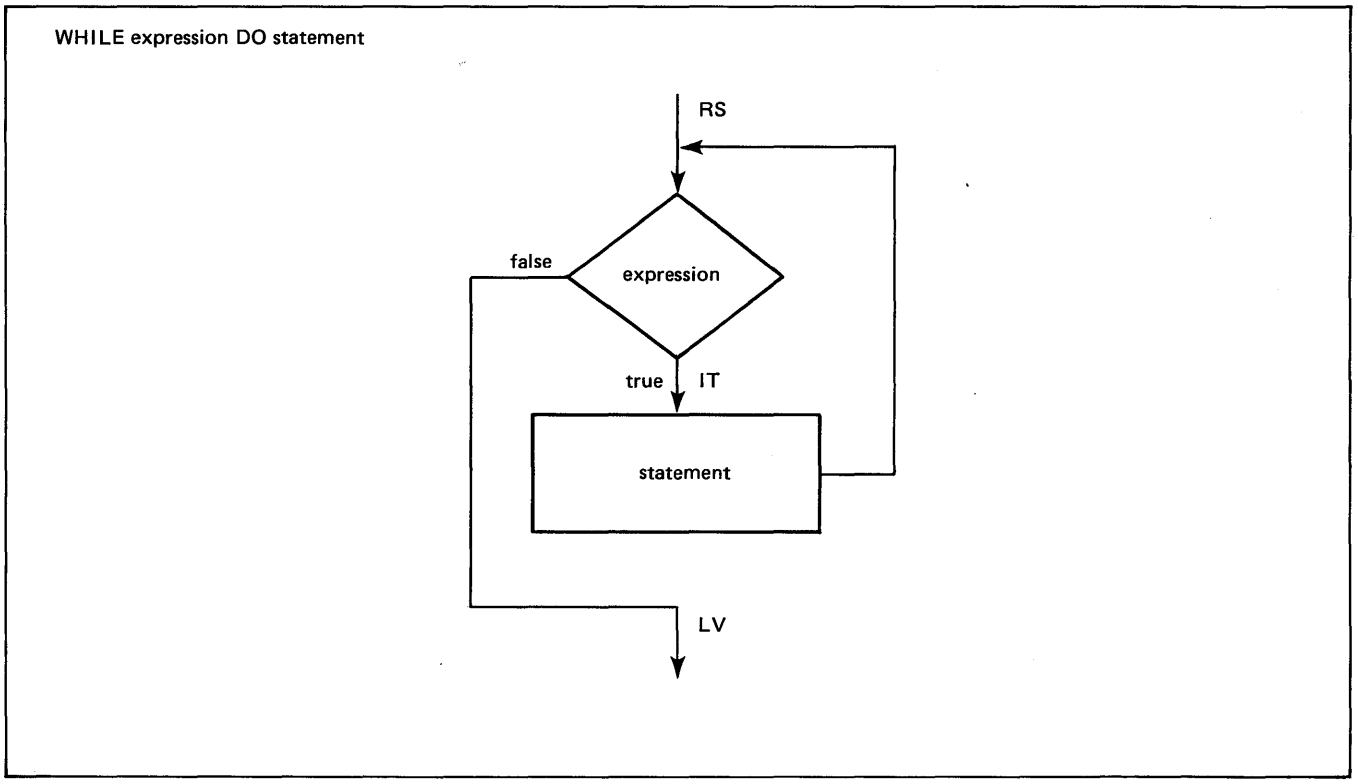


Figure 9-6. While Statement Flow Chart

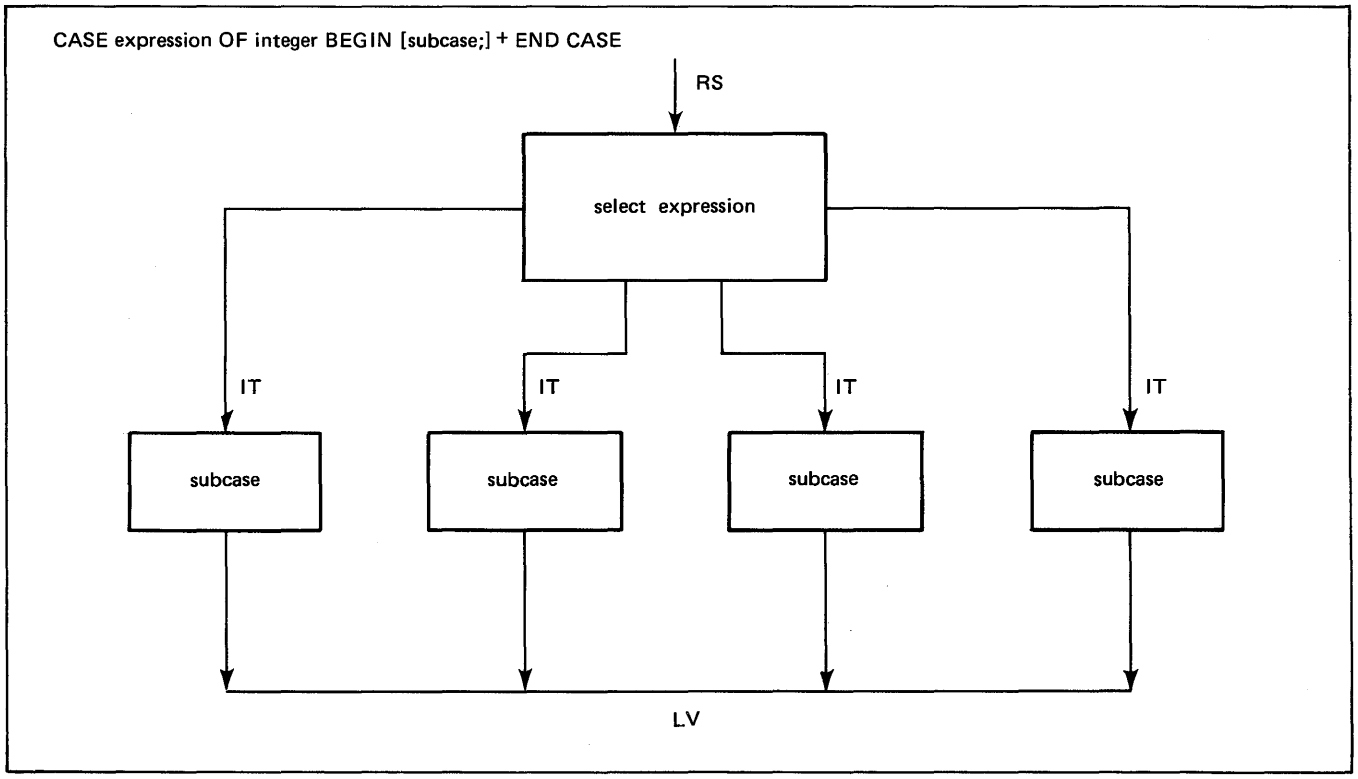


Figure 9-7. Case Statement Flow Chart

SECTION X

FUNCTION EXECUTION

User-defined functions (or simply, defined functions) may be used in the same manner as primitive functions, except that they may not be used as arguments of primitive operators. A defined function may be used in calculator mode or it may be called from within another defined function.

When a defined function is invoked, its execution begins with the first statement, then successive statements are executed in order, except as this order is changed by branch instructions.

For example, consider the function CIRCLEAREA:

```
      ⎕CR 'CIRCLEAREA'
  AREA←RADIUS CIRCLEAREA DEGREES;LOCAL1;LOCAL2
  AREA←(○RADIUS*2)×DEGREES÷360
```

When this function is executed with the statement

```
    265.3 CIRCLEAREA 16.67
```

the value 265.3 is assigned to the local name RADIUS and the value 16.67 is assigned to the local name DEGREES. The body of the function then is executed and the statement

```
    AREA (oRADIUS*2)xDEGREES 360
```

computes a value for the result variable AREA.

A function like CIRCLEAREA, which produces an explicit result, may be used in compound expressions. For example,

```
      PRICE←12×36000×12.4 CIRCLEAREA 36.2
      PRICE
20983747.88
```

The value computed for the result variable AREA in the function CIRCLEAREA is used to compute PRICE. The result variable, AREA, is treated the same as any local variable and therefore has no significance after the function is executed:

```
      AREA
VALUE ERROR
      AREA
      ^
```

HALTED EXECUTION

Execution of a function may be stopped before completion in the following ways:

* By an error report.
* By an interrupt from the terminal.
* By use of the stop control system function ⎕SS (see page 10-10).
* By execution of the HALT statement.

When a function is stopped before its execution is complete, the function is suspended. The name of the function is displayed, with a line number beside it. If the suspension is because of an error or interrupt from the terminal, the line is displayed with an appropriate message and an indication of the point of interruption. Unless multiple specification arrows or other used-defined functions appear in the line, the state of computation was restored to the condition existing before the line started to execute.

The displayed number generally is the number of the line that would be executed next if the function were to continue normally. Execution of the suspended function can be resumed by entering a branch arrow to the line counter system function (⎕LC), or by entering)RESUME (see page 10-8 for a discussion of the)RESUME command).

Entering →0, or a branch to a number outside the range of statement numbers in the function causes an immediate exit from the function.

All normal activities are possible when a function is in the suspended state. Statements or system commands may be executed, or execution of the function may be resumed at any point, or the editor may be invoked to edit any function which is not pendent (see below).

STATE INDICATOR SYSTEM COMMAND

The state indicator system command)SI displays the state indicator. A typical display has the form

```
          )SI
    A[4]  *
    B[6]
    D[4]
    C[2]  *
    D[1]
```

and indicates that execution was halted before completing execution of line 4 of function A, the current use of function A was called in line 6 of function B, function B was called in D[4], the use of function C was halted at line 2, and that function C was called in D[1]. The asterisks appearing to the right of A[4] and C[2] indicate that functions A and C are suspended. The functions B and D are defined as being pendent, because their execution can be resumed only as a result of function A resuming its execution. The term halted is used to define a function which is either pendent or suspended.

Additional functions can be invoked when in the suspended state. For example, if C were called now and a further suspension occurred in statement 3 of function D, itself invoked in statement 7 of C, the state indicator display would be:

```
          )SI
    D[3]  *
    C[7]
    A[4]  *
    B[6]
    D[4]
    C[2]  *
    D[1]
```

Because the line counter, ⎕LC, holds the current statement numbers of functions that are in execution, its value at this point would be the vector 3 7 4 6 4 2 1. The sequence from the last suspension to the preceding suspension can be cleared by entering a single branch arrow:

```
          →
          )SI
    A[4]  *
    B[6]
    D[4]
    C[2]  *
    D[1]
          ⎕LC
    4 6 4 2 1
```

Repeated use of the branch arrow will clear the state indicator and restore ⎕LC to an empty vector. (The)RESET system command (see page 10-7) has the same effect.) The cleared state indicator is displayed as a blank line. See page 11-6 for further applications of the state indicator system command.

STATE INDICATOR DAMAGE

If a function name occurs in the state indicator list, erasure of that function or replacement of that function by copying an object with the same name (even another example of the same function) makes it impossible for the original execution to be resumed. In this case, an SI DAMAGE message is reported.

If an SI DAMAGE message is reported for a suspended function, it will be impossible to resume its execution, but the function can be invoked again, with or without prior clearance of the state indicator.

APL\3000 EXTENDED CONTROL FUNCTIONS

The state indicator)SI displays a list of pendent and suspended functions in the order in which they were called. It also displays the line number on which each function is suspended and optionally, if)SIV is used, a list of all variables shadowed by each function call. Each of the user-defined function names which appear on the state indicator is termed a control point and the collection of all control points displayed by the state indicator is termed an environment. The current control point is the function which is currently executing or suspended, and the current environment is the set of function calls which would be displayed by the state indicator if it were called at the current control point.

In order to facilitate the execution of APL statements in environments other than the current environment, two system functions are available in APL\3000 which allow the saving of new environments for later use. An arbitrary APL expression can then be executed in one of these saved environments through the use of the extended execute function.

CAPTURE STACK ENVIRONMENT SYSTEM FUNCTION

The form of the capture stack environment system function is

```
    A←F ⎕CSE C S D
```

where

A = assigned environment number
F = function name
C = count (scalar, unit, or 1 to 3 element vector)
S = starting environment
D = desired environment number

The ⎕CSE function searches down the list of control points beginning with the starting environment for the control point specified by count and with the designated function name. If the required control point is found, it is assigned, along with its environment, to the assigned environment number (a number between 1 and 15 which can be used to access the captured environment at a later time). Environment 0 is always defined as the current environment.

If function name is not specified, the control point specified by count (regardless of name) will be captured. Although the execute and evaluated input functions (⍎ and ⎕) appear in the status indicator, they are not considered as control points. They cannot be captured by ⎕CSE and do not participate in the count. If the function name is not specified and the count exceeds the number of user-defined functions in the starting environment, the global environment is captured.

If a desired environment number is not specified in the right argument, the next available environment number is chosen. If the environment limit is exhausted, an error message is returned.

If a desired environment number is specified in the right argument, any environment previously assigned to that number is released before the new environment is captured.

If a starting environment is not specified, the current environment (environment 0) is assumed. If a starting environment is specified, the search starts in that environment but control always returns to the current environment.

RELEASE STACK ENVIRONMENT SYSTEM FUNCTION

The form of the release stack environement system function is

```
    RL  ⎕RSE EL
```

where

```
    RL = released environment list
    EL = environment list
```

The ⎕RSE function releases a list of environments previously captured by ⎕CSE. The released environment list contains a list of environments actually released, this may be different from the environment list because some of the environments in environment list may be empty or non-existent. ⎕RSE may be used with the current environment (number zero) which will cause the current environment to be reduced to the empty environment.

EXTENDED DYADIC EXECUTE PRIMITIVE FUNCTION

The form of the extended dyadic execute primitive function is

```
    N ⍎ E
```

where

```
    N = environment number
    E = character scalar, vector, or unit representing the APL
        expression to be evaluated
```

The dyadic form of execute evaluates an APL expression in the same way that the monadic form evaluates these expressions, except that the dyadic form evaluates the expression in the environment specified by environment number, which may be different from the current environment. If E does not contain a branch, the resulting value (that is, the result of the expression evaluated in the specified environment) is returned to the current environment as the value of the execute function.

If E results in a branch, the branch is executed as if it had occurred in the environment specified by environment number, and the environment from which execute was called is released unless it has been explicitly captured using ⎕CSE.

The following examples illustrate possible uses of the extended stack control functions:

Example 1.

Suppose APL is being used to simulate machine code for a hypothetical machine, and one of the instructions simulated is a relative branch. This can be simulated as follows:

```
[0]  CODE
[1]  LD A                  This simulated machine code program
[2]  LDI 1                 will continuously add 1 to the contents
[3]  ADD                   of memory location A.
[4]  STO A
[5]  BR ¯4
```

The BR program can be written using the extended control functions as follows:

```
[0]      BR OFFSET; ENVIRONMENT; NEXTLINE
[1]      ⍝ CAPTURE THE ENVIRONMENT OF THE FUNCTION WHICH CALLED BR
[2]      ENVIRONMENT←⎕CSE 2 0 1
[3]      ⍝ CALCULATE THE LINE TO BRANCH TO
[4]      NEXTLINE←⎕LC [2] + OFFSET
[5]      ⍝ EXECUTE THE BRANCH IN THE FUNCTION WHICH CALLED BR
[6]      ENVIRONMENT⍎'→',⍕NEXTLINE
```

A shorter version of this program is shown below:

```
(⎕CSE 2 0 1) ⍎'→⎕LC+',⍕OFFSET
```

Example 2.

Suppose that function TEST has local variable A, and the system is suspended in TEST. The following sequence will return the global (unshadowed) value of A.

```
      A←'GLOBAL A'
      ⎕CR 'TEST'
TEST; A
      A←'LOCAL A'
A
      2 ⎕SS 'TEST'   ⍝ STOP BEFORE EXECUTION OF LINE 2
2
      TEST
TEST[2] *
      A
LOCAL A
      ⎕CSE 2   ⍝ CAPTURE GLOBAL ENVIRONMENT
1
      1⍎'A'   ⍝ GLOBAL ENVIRONMENT CAPTURED AS ENVIRONMENT 1
GLOBAL A
```

The following system variables can be used to facilitate the use of the extended stack control system functions.

STACK NAMES SYSTEM FUNCTION

The stack names system function (⎕SN) returns a character matrix containing the names of the user-defined functions halted in the environment in which ⎕SN is evaluated. For example, 1⍎'⎕SN' will return a matrix of the function names halted in environment 1.

STATE INDICATOR AND STATE INDICATOR WITH VARIABLES

The state indicator and state indicator with variables system commands are entered as

```
)SI n
)SIV n
```

where n is an integer between 0 and 15 (default is 0). The environment displayed will be environment n. If environment n is not the current environment (environment 0), some of the function names may appear with a o (shift letter o in the APL character set) following the name. A o following the function name indicates that the function is not halted in the current environment.

For example, suppose that the state indicator displays a suspended and a pendent function as follows:

```
      )SI
TEST[2] *
TEST1[3]
```

If this environment is captured and the stack is then cleared, the new state indicator is shown below:

```
      ⎕CSE 1 0 2   ⍝CAPTURE ENVIRONMENT 2
2
                   ⍝CLEAR CURRENT ENVIRONMENT
      )SI
      )SI 2
TEST[2] ⊛
TEST1[3] o
```

This indicates that the functions TEST and TEST1 are no longer in the current environment, although they are contained in environment 1.

RESET SYSTEM COMMAND

The form of the RESET system command is

```
)RESET n
```

where n is an integer between 0 and 15 (default is 0). The RESET system command releases the environment specified by _n_. If _n_ is omitted, the current environment is released.)RESET _n_ is equivalent to executing ⎕RSE _n_.

DEPTH SYSTEM COMMAND

The form of the DEPTH system command is

```
    )DEPTH n
```

where _n_ is an integer specifying the size of the execution stack. The execution stack controls the number of nested functions allowed. For example, if _n_ is set to 64, up to 64 functions can be nested at any one time. A DEPTH ERROR will be returned if the number of nested functions exceeds the size of the execution stack.

RESUME SYSTEM COMMAND

The)RESUME system command resumes execution of a suspended function. Examples of the)RESUME command are shown starting on page 10-13 .

DEBUGGING AIDS

The system functions shown in table 10-1 are used to debug lines of unlocked user-defined functions.

Table 10-1. System Functions used for Debugging

MONADIC (All lines)	NAME	DYADIC (Specified lines)	RESULT
⎕ST F	Set Trace	N ⎕ST F	L
⎕SS F	Set Stop	N ⎕SS F	L
⎕SM F	Set Monitor	N ⎕SM F	L
⎕RT F	Reset Trace	N ⎕RT F	L
⎕RS F	Reset Stop	N ⎕RS F	L
⎕RM F	Reset Monitor	N ⎕RM F	L
⎕QT F	Query Trace		B
⎕QS F	Query Stop		B
⎕QM F	Query Monitor		B
⎕MV F	Monitor Values	N ⎕MV F	M

Notes:

F is a character vector denoting the name of an unlocked user-defined function.

N is a numeric vector of line numbers.

L is a numeric vector of lines with property (set, reset).

B is a boolean vector, 1 if the property is set, 0 if not set. (One element per line including header.)

M is a matrix of monitor values. The first column contains the number of executions, and the second column contains the execution or compute time for each line for which values are requested. First row corresponds th header, second row to line 1, and so forth. Values for header signify number of times function executed and CPU time for function.

The monadic forms of the debugging system functions apply to all lines including the header line (line 0). The dyadic forms apply only to the lines specified in the left argument.

During function execution, the effects of the aids are as follows on encountering a line:

	HEADER LINE	BODY LINE
Trace	Result returned by function	Result
Stop	Suspend prior to return from function	Suspend prior to execution of line
Monitor	Increase number of calls to function and total cpu time in function	Increase number of times line has been executed and increase cpu time in line execution

The trace result forms are

<u>Function name</u> [<u>line number</u>]

<u>Function name</u> [<u>line number</u>] <u>type</u> (<u>shape</u>) <u>value</u>

<u>Function name</u> [<u>line number</u>] (<u>shape</u>) <u>value</u>

The first form above occurs if no result is possible; otherwise, the second form occurs. The third form occurs when a line results in a branch.

The type is C for character or N for numeric. The shape is a numeric vector representing the result of monadic ⍴ , and value is the normal displayed value (printed beginning on next line if ⍴⍴>1).

The stop result form is

<u>Function name</u> [<u>line number</u>] *

SET TRACE, SET STOP, AND SET MONITOR FUNCTIONS

The set trace, set stop, and set monitor functions (⎕ST, ⎕SS, and ⎕SM, respectively) set the trace, stop, and monitor states of lines of a user-defined function. These set functions can be used either monadically or dyadically. If these functions are used monadically, the appropriate state is set for all the lines of the function specified by the character scalar, vector, or unit right argument. If used dyadically, the state is set for only those lines specified in the numeric scalar, vector, or unit left argument. Both forms return as their results numeric vectors denoting those lines for which the state is now set.

Note that these functions do not reset the states each time they are called; lines which are not (implicitly or explicitly) referenced are not affected.

RESET TRACE, RESET STOP, AND RESET MONITOR FUNCTIONS

The reset trace (⎕RT), reset stop (⎕RS), and reset monitor (⎕RM) functions are analogous to the set functions (described above), except that they reset the designated state. Their arguments are the same as those for the set functions; their results are analogous.

MONITOR VALUES FUNCTION

The monitor values system function (⎕MV) is dyadic or monadic. This function returns an array of execution count and execution time for lines of the function specified by its character scalar, vector, or unit right argument. If the function is used monadically, the monitor values for all the lines of the function are returned. If used dyadically, only values for those lines specified by the numeric scalar, vector, or unit left argument are returned.

The accumulated number of milliseconds is contained in ⎕MV. A time of 0 indicates unmonitored lines or monitored lines that have not been executed. Thus, monitoring all lines over a period of execution is an effective way to determine if some program path has reached each line, and also the time spent in each line.

If a line contains a call on another function, any time spent in that called function is accumulated there, instead of in the calling line.

The result of ⎕MV is a matrix of shape $\underline{n}\times 2$, where $\underline{n}$ is the number of lines in the function (including the header) if used monadically, or the length of the left argument if used dyadically. The first column contains the number of times the line has been executed since the last set monitor of the line; the second column is the compute time used by that line (excluding that used by user-defined functions called by that line) in milliseconds. The values for line number zero indicate the number of times the function has been called and the amount of computer time it has used.

QUERY TRACE, QUERY STOP, AND QUERY MONITOR FUNCTIONS

The query trace (⎕QT), query stop (⎕QS), and query monitor (⎕QM) functions take as their only argument a character scalar, vector, or unit specifying the name of a function whose trace, stop, or monitor states are to be queried.

The results of these functions are boolean vectors, with a one denoting that the state (trace, stop, or monitor) is set for that line, and a zero denoting that the state is not set. The elements of the result correspond to the lines of the function, with the first element corresponding to line zero, the second to line one, and so forth.

Examples of the debugging aid system functions are provided at the end of this section.

LOCKED FUNCTIONS

If LOCK is used instead of END in the editor to save a defined function, the function becomes locked. A locked function cannot be edited or displayed. Any associated stop control or trace control function is nullified after the function is locked.

A locked function is treated in the same manner as a primitive, and its statements are concealed as much as possible. Execution of a locked function is terminated by any error occurring within it, or by a strong interrupt from the terminal. If execution stops, the function is never suspended but is immediately abandoned. The message displayed for a stop is a DOMAIN error if an error of any kind occurred, WS FULL if the stop resulted from a system limitation, or INTERRUPT if it was stopped from the terminal.

A locked function is never pendent, and if an error occurs in any function called either directly or indirectly by a locked function, the execution of the entire sequence of nested functions is abandoned. If the outermost locked function was called by an unlocked function, the outermost function is suspended; if it was called by an entry from the terminal, an error message is displayed with a copy of the statement that called the function.

When a soft interrupt from the terminal is encountered in a locked function, or in any function that was called by a locked function, execution continues normally up to the first interruptable point, which is either the next statement in an unlocked function that called the outermost locked function, or the completion of the terminal entry that used this locked function. In the latter case, the soft interrupt has no net effect on function execution, only on display of output if the explicit result of the function is not directly used.

Locked functions may be used to keep a function definition proprietary, or as part of a security scheme for protecting other proprietary information.

```
        ⎕QS 'ROOTS'
1   0   0   0   0   0   0   0   0   0   0   1
        ⎕RS 'ROOTS'
0   1   2   3   4   5   6   7   8   9   10   11

        ⎕QS 'ROOTS'
0   0   0   0   0   0   0   0   0   0   0   0
        ⎕SS 'ROOTS'
0   1   2   3   4   5   6   7   8   9   10   11
        ⎕QS 'ROOTS'
1   1   1   1   1   1   1   1   1   1   1   1
        ROOTS
ROOTS[1]*
        )RESUME
ENTER A NUMBER
ROOTS[2]*
        )VARS
A       B       LABEL1  LABEL2  LABEL3  LABEL4  N
        )RESUME
AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
ROOTS[3]*
        )RESUME
AND THE CUBE ROOT
ROOTS[4]*
        )SI
ROOTS[4]*
        )SIV
ROOTS[4]*       LABEL1  LABEL2  LABEL3  LABEL4
        )RESUME
⎕:
        64
ROOTS[5]*
        (⍳11) ⎕RS 'ROOTS'
1   2   3   4   5   6   7   8   9   10   11
        )RESUME
THE SQUARE ROOT IS 8
THE CUBE ROOT IS 4
ENTER 0 IF YOU DO NOT WISH TO CONTINUE
⎕:
        90
THE SQUARE ROOT IS 9.486832981
THE CUBE ROOT IS 4.481404747
ENTER 0 IF YOU DO NOT WISH TO CONTINUE
⎕:
        0
ROOTS[0]*
        )RESUME
```

```
      )VARS
A       B       N
      )RESUME
      ⎕QS 'ROOTS'
1  0  0  0  0  0  0  0  0  0  0  0
      0 1 5 8 ⎕SM 'ROOTS'
0  1  5  8
      ⎕SS 'ROOTS'
0  1  2  3  4  5  6  7  8  9  10  11
      ⎕QS 'ROOTS'
1  1  1  1  1  1  1  1  1  1  1  1
      ⎕QM 'ROOTS'
1  1  0  0  0  1  0  0  1  0  0  0
      ROOTS
ROOTS[1]*
      ⎕RS 'ROOTS'
0  1  2  3  4  5  6  7  8  9  10  11
      )RESUME
ENTER A NUMBER
AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
AND THE CUBE ROOT
⎕:
      42
THE SQUARE ROOT IS 6.480740698
THE CUBE ROOT IS 3.476026645
ENTER 0 IF YOU DO NOT WISH TO CONTINUE
⎕:
      0
      ⎕QS 'ROOTS'
0  0  0  0  0  0  0  0  0  0  0  0
      ⎕QT 'ROOTS'
0  0  0  0  0  0  0  0  0  0  0  0
      6 ⎕ST 'ROOTS'
6
      ROOTS
ENTER A NUMBER
AND THE COMPUTER WILL COMPUTE THE SQUARE ROOT
AND THE CUBE ROOT
⎕:
      9
ROOTS[6]  N () 2.080083823
THE SQUARE ROOT IS 3
THE CUBE ROOT IS 2.080083823
ENTER 0 IF YOU DO NOT WISH TO CONTINUE
⎕:
      0
      ⎕QT 'ROOTS'
0   0   0   0   0   0   1   0   0   0   0   0
      ⎕RT 'ROOTS'
0   1   2   3   4   5   6   7   8   9   10   11
      ⎕MV 'ROOTS'
   2   1583
   2     87
   0      0
   0      0
   0      0
   2     18
   0      0
   0      0
   2    170
   0      0
   0      0
   0      0
```

SYSTEM COMMANDS

SECTION XI

System commands are used for sucn things as monitoring and modifying the workspace environment, saving and then reactivating copies of a workspace, accessing the APL\3000 editor, resuming suspended functions, and terminating an APL session.

System commands are prefixed by a right parentheses and can only be entered in immediate execution mode; they cannot be used as part of a defined function. The complete set of system commands is shown in table 11-1.

INITIAL VALUES IN A WORKSPACE

Some items in a workspace are set to certain standard values when the workspace is first accessed. In particular, the workspace contains the settings of the state indicator and several system variables. These settings are shown in table 11-2.

)CLEAR COMMAND

The form of the)CLEAR command is

```
    )CLEAR
```

The)CLEAR command is used to clear (and discard) the contents of the active workspace and reset the workspace to the standard initial values (see table 11-2).

An example of the)CLEAR command is shown below:

```
          )CLEAR
    CLEAR WS
```

Table 11-1. System Commands

NAME	SYNTAX	PURPOSE
Bind	)BIND	Sets the BIND flag ON or OFF
Clear	)CLEAR	Clears the active workspace
Continue	)CONTINUE	Saves CONTINUE file and terminates APL session
Copy	)COPY [*namelist*]	Obtains objects from saved workspace
Depth	)DEPTH num	See Section X
Drop	)DROP *wsname*	Purges workspace
Edit	)EDIT [*name*]	Accesses APL \ 3000 editor
Erase	)ERASE [*namelist*]	Deletes objects from workspace
Files	)FILES [*groupname.acctname*]	Lists all files in user's library or, optionally, all files in specified group and account.
Functions	)FNS [*letter*]	Lists user-defined functions in the active workspace.
Help	)HELP [*cmdname*]	Displays information on system commands
Library	)LIB [*groupname* [*acctname*]]	Lists workspaces in specified library
Load	)LOAD *wsname*	Replaces active workspace with duplicate of saved workspace
MPE	)MPE	Exits APL and enters MPE
Off	)OFF	Terminates APL session
Protected copy	)PCOPY *wsname* [*namelist*]	Obtains objects from named workspace. Does not replace named objects in active workspace.
Reset	)RESET	See Section X
Resume	)RESUME	See Section X
Save	)SAVE *wsname*	Saves duplicate of active workspace
State indicator	)SI	Lists state indicator in the active workspace
State indicator with variables	)SIV	Lists state indicator in the active workspace with names local to user-defined functions
Terminal type	)TERM [*termtype*]	Sets terminal type
Terse	)TERSE	Sets error messages to "terse"
Time	)TIME	Returns elapsed wall time and elapsed CPU time

Table 11-1. System Commands (Continued)

NAME	SYNTAX	PURPOSE
Variables	)VARS [*letter*]	Lists variables in the active workspace
Verbose	)VERBOSE	Sets error messages to "verbose"
Workspace identification	)WSID [*wsname*]	Displays the active workspace name, or, when wsname is included, renames workspace.

namelist = *name* [*name*] [*name*] . . . [*name*]

wsname = *workspace identification* [*/lockword*] [*.groupname* [*.accountname*]]

Note: All workspaces are saved with MPE lockwords. If the *lockword* parameter is not supplied by the user, APL\ 3000 supplies APL00000.

The reason is that if an attempt is made to open a file containing a lockword, and the lockword parameter is omitted, MPE prints

LOCKWORD: *fileid*

on the output device.

If the output device is an APL character=set device, it prints

⎕∘ ∩'⍵○⍴⌊.

To change the lockword of a saved workspace, enter)DROP, then)SAVE with new lockword.

Table 11-2. Initial Values in a Workspace

Latent expression, ⎕LX	Empty
Depth,)DEPTH	66
Line counter, ⎕LC	Empty
Stack names, ⎕SN	Empty
State indicator,)SI	Cleared
Workspace identification,)WI	Empty (UNNAMED WS)
Printing precision, ⎕PP	10
Printing width, ⎕PW	80
Comparison tolerance, ⎕CT	1E 13
Random link, ⎕RL	0
Language, ⎕LA	'APL'
Assert level, ⎕AL	0
Horizontal tabs, ⎕HT	0
Virtual memory, ⎕VM	256 -24
Index origin, ⎕IO	1

)ERASE COMMAND

The form of the)ERASE command is

)ERASE [namelist]

The)ERASE command deletes objects (functions and variables) identified by the namelist parameter from the workspace. Shared variable offers pertaining to any of these objects are retracted.

If a halted function is erased, the report SI DAMAGE is displayed. It is not possible to resume the execution of an erased function, and the the state indicator should be cleared of indications of damage (see Section X).

If an object specified in the namelist parameter cannot be erased, the message NOT ERASED: is reported, followed by the name of the object not erased.

An example of the)ERASE command is shown below:

```
      )VARS
A       ALTER   APL101  APL102  APL103  APL104  APL11   APL31   APL32   APL33
APL34   APL35   APL51   APL52   APL61   APL62   APLGOL1 APLGOL2 APLGOL3 APLGOL4
APLGOL5 APLGOL6 APLGOL7 APLGOL8 APLGOL9 APLSET  ARRAY   B       C       CHAR
D       E       EDIT1   INCOME  N       RESHAPE1        RESHAPE2        SHAPE
TIME    VEC     VECTOR  X       XQR     Y       YIELD   Z
      )ERASE ALTER VEC XXQR
      )VARS
A       APL101  APL102  APL103  APL104  APL11   APL31   APL32   APL33   APL34
APL35   APL51   APL52   APL61   APL62   APLGOL1 APLGOL2 APLGOL3 APLGOL4 APLGOL5
APLGOL6 APLGOL7 APLGOL8 APLGOL9 APLSET  ARRAY   B       C       CHAR    D
E       EDIT1   INCOME  N       RESHAPE1        RESHAPE2        SHAPE   TIME
VECTOR  X       XQR     Y       YIELD   Z
```

)COPY COMMAND

The form of the)COPY command is

)COPY wsname [namelist]

The)COPY command copies the objects specified in the namelist parameter from the workspace indicated by wsname (the source workspace) into the active workspace. If namelist is omitted, all objects (except system variables) in the source workspace are copied.

When an object to be copied has the same name as an object in the active workspace, the copied object replaces the object in the active workspace. If there is a shared variable offer pending with respect to the object thus replaced, the offer is retracted.

If names explicitly included in the)COPY command are not the names of objects in the source workspace, APL reports NOT COPIED:, followed by a list of the objects not found.

An example of the)COPY command is shown below:

```
      )COPY WS2
SAVED 12:44 10/14/76
```

)PCOPY COMMAND

The form of the)PCOPY command is

)PCOPY wsname [namelist]

The)PCOPY (protected copy) command works like the)COPY command, except that if the namelist parameter specifies objects having the same name of objects in the active workspace, the objects in the source workspace are not copied. APL reports objects not copied for this reason by displaying

NOT COPIED: list of objects

An example of the)PCOPY command is shown below:

```
          )PCOPY WS2 ROOTS
      NOT COPIED:     ROOTS
      SAVED 12:44 10/14/76
```

)FNS COMMAND

The form of the)FNS command is

)FNS [letter]

The)FNS command lists functions in the active workspace in alphabetic order, starting with the letter specified. If letter is omitted, all functions are listed.

An example of the)FNS command is shown below:

```
          )FNS
      BOOTS   CIRCLEAREA        GOLFSCORE         ROOTS
```

)VARS COMMAND

The form of the)VARS command is

)VARS [letter]

The)VARS command lists variables in the active workspace in alphabetic order, starting with the letter specified. If letter is omitted, all variables are listed.

An example of the)VARS command is shown below:

```
          )VARS
      A         ALTER     APL101    APL102    APL103    APL104    APL11     APL31     APL32     APL33
      APL34     APL35     APL51     APL52     APL61     APL62     APLGOL1   APLGOL2   APLGOL3   APLGOL4
      APLGOL5   APLGOL6   APLGOL7   APLGOL8   APLGOL9   APLSET    ARRAY     B         C         CHAR
      D         E         EDIT1     INCOME    N         RESHAPE1            RESHAPE2            SHAPE
      TIME      VEC       VECTOR    X         XQR       Y         YIELD     Z
          )VARS G
      INCOME    N         RESHAPE1            RESHAPE2            SHAPE     TIME      VEC       VECTOR
      X         XQR       Y         YIELD     Z
```

)SI COMMAND

The form of the)SI command is

)SI N

The)SI command displays the state indicator, which shows the status of halted functions. The most recently halted function is listed first. If N is specified, it must be an integer between 0 and 15, and it causes the environment specified by N to be displayed. See Section X for a discussion of the use of environment numbers.

The list shows the name of the function and the number of the line at which the function halted. Actions which can be taken with respect to a halted function are discussed in Section X.

Suspended functions are denoted in the state indicator list by an asterisk, while pendent functions appear without an asterisk.

An example of the)SI command is shown below:

```
              )SI
        ROOTS[3]*
```

)SIV COMMAND

The form of the)SIV command is

```
    )SIV N
```

The)SIV command displays the state indicator in the same way as the)SI command, but in addition, lists names local to each function.

If N is specified, it must be an integer between 0 and 15, and it causes the environment specified by N to be displayed. See Section X for a discussion of the use of environment numbers.

An example of the)SIV command is shown below:

```
              )SIV
        ROOTS[3]*       LABEL1  LABEL2  LABEL3  LABEL4
```

WORKSPACE STORAGE AND RETRIEVAL

A duplicate of the active workspace for may be saved later use. When this duplicate is subsequently reactivated, the entire workspace is restored as it was, except that variables which were shared in the active workspace when saved are not shared automatically again when the workspace is reactivated.

LIBRARIES OF SAVED WORKSPACES

The set of saved workspaces is called a library. Each workspace is identified by group and account names as well as the actual name assigned to it. In referring to workspaces in the user's own library, however, the group and account names may be omitted, because they are supplied automatically.

In systems with multiple APL users, it often is convenient to use functions or variables contributed by others. A user may activate an entire workspace saved by another user, or he may copy selected items from another user's workspace. In order to copy another user's workspace, the group and account names, if different, must be supplied together with the workspace name.

Some libraries (usually identified by a special group and account name, for example, PUB.SYS) are not assigned to individual users, but are designated as <u>public</u> libraries. There may be restrictions, however, on who can save, delete, or modify a workspace in a public library. In general, a public library workspace can be re-saved or deleted only by the user who first saved it.

NAMES AND PASSWORDS OF WORKSPACES

A saved workspace must be named. The name of a workspace may duplicate a name used for an APL object within the workspace. A password may be used with the name of a workspace. If a password is used, any reference to the saved workspace must specify this password.

)WSID COMMAND

The form of the)WSID command is

```
    )WSID wsname
```

The)WSID command renames an active workspace with the name specified by <u>wsname</u>.

APL displays WAS. . ., followed by the former name.

Another form of the)WSID command with no parameters is

```
    )WSID
```

This form reports the identification of the active workspace, listing the group and account names (if other than the user's) and the password.

Examples of the)WSID command are shown below:

```
          )WSID
IS NOT NAMED
          )WSID WS4
WAS NOT NAMED
          )WSID
IS WS4
```

)SAVE COMMAND

The form of the)SAVE command is

```
    )SAVE wsname
```

The)SAVE command saves a duplicate of the active workspace with the name specified by wsname. The workspace is saved in the group library associated with the user unless otherwise specified. A password is included in the name if the password portion of wsname is specified.

APL acknowledges saving by a report listing the date and time at which the workspace was saved, and the wsname.

An example of the)SAVE command is shown below:

```
          )SAVE WS2
    SAVED 14:05 10/14/76  WS2
```

)CONTINUE COMMAND

The form of the)CONTINUE command is

```
    )CONTINUE
```

The)CONTINUE command saves the active workspace under the name CONTINUE and terminates the session.

Additionally, when a session is aborted for any reason except a normal log-off (such as the connection to the computer being broken), the workspace is saved with a name such as A2661516, where the first three digits specify the day of the year (the 266th day in this case), and the last four digits specify the time of day (3:16 PM in this case).

An example of the)CONTINUE command is shown below:

```
    )CONTINUE
```

)LOAD COMMAND

The form of the)LOAD command is

```
    )LOAD wsname
```

The)LOAD command discards the active workspace and then transfers a duplicate of the saved workspace specified by wsname into the active workspace. Shared variable offers in the former active workspace are retracted.

APL displays the date and time at which the loaded workspace was last saved. The latent expression (⎕LX) in the loaded workspace is executed automatically.

An example of the)LOAD command is shown below:

```
          )LOAD WS2
    SAVED 14:05 10/14/76
```

)DROP COMMAND

The form of the)DROP command is

)DROP wsname

The)DROP command removes the workspace specified by wsname from the library in which it is contained. The password is required in the wsname parameter to drop a workspace.

The)DROP command has no effect on the active workspace.

An attempt to drop a workspace by someone other than the user who saved it is rejected with the error report IMPROPER LIBRARY REFERENCE.

An example of the)DROP command is shown below:

```
          )DROP WS1
      DROPPED
          )DROP WS3
      WS NOT FOUND
```

)LIB COMMAND

The form of the)LIB command is

)LIB [groupname[.accountname]]

The)LIB command displays the names of the workspaces, in alphabetic order, in the specified library.

An example of the)LIB command is shown below:

```
          )LIB
      A2881407  CONTINUE  JWSAVE    WS2       WS4
```

)HELP COMMAND

The form of the)HELP command is

)HELP [cmdname]

The)HELP command returns a listing of the system commands. If the optional cmdname parameter is specified, the)HELP command returns a brief description of the specified command.

Examples:

```
          )HELP
      COMMANDS LEGAL FROM CALCULATOR MODE:
      CLEAR     CONTINUE  COPY      DROP      EDIT      ERASE     FILES     FNS
      MPE       HELP      LANGUAGE  LIB       LOAD      OFF       PCOPY     BIND
      RESET     RESUME    SAVE      SI        SIV       VARS      WSID      TIME
      DEPTH     TERM      TERSE     VERBOSE
      ENTER )HELP <COMMAND> FOR A BRIEF DESCRIPTION OF THE COMMAND
          )HELP MPE
        THE )MPE COMMAND IS USED TO LEAVE APL AND ENTER MPE.
```

)TERM COMMAND

The form of the)TERM command is

)TERM [termtype]

where termtype signifies the type of terminal being used. Possible values for termtype are:

ASCII - ASCII terminal

BP - Bit-pairing

CDI - Computer Devices, Inc.

CP - Character-pairing

DM - DataMedia

GSI - GenCom Systems, Inc.

HP - Hewlett-Packard

An example of the)TERM command:

```
      )TERM
IS ASCII
      )TERM HP
WAS ASCII
```

)TERSE COMMAND

The)TERSE command sets error messages to "terse." For example,

```
      6÷0
REAL DIVIDE BY 0
       6÷0
       ↑
      )TERSE
      6÷0
DOMAIN ERROR
       6÷0
       ↑
```

)VERBOSE COMMAND

The)VERBOSE command sets error messages to "verbose." For example,

```
          6÷0
    DOMAIN ERROR
          6÷0
           ↑
          )VERBOSE
          6÷0
    REAL DIVIDE BY 0
          6÷0
           ↑
```

Verbose is the default mode.

)BIND COMMAND

The)BIND command sets a BIND flag on or off. If off when the)BIND command is entered, the flag is turned on; if on, the flag is turned off.

If a binding error occurs during program execution and the BIND flag is on, the statement in which the binding error occurred is listed along with an indication of the position of the binding error.

An example of the)BIND command is shown below:

```
          )BIND
    NOW ON
          )BIND
    NOW OFF
```

)FILES COMMAND

The form of the)FILES command is

)FILES [groupname.acctname]

The)FILES command is used to list all files in the user's account. If followed by the optional groupname.acctname parameter, all files in the account specified are listed.

An example of the files command is shown below:

```
          )FILES
    A2881407  JWSAVE     WS2        WS4
```

)MPE COMMAND

The)MPE command is used to exit APL and enter the MPE operating system. For example,

```
      )MPE
:LISTF

FILENAME

A2881412      JWSAVE      WS2      WS4

:RESUME
```

Note that when the MPE :RESUME command is entered, the READ PENDING message is not displayed (as it is when BREAK is used).

)TIME COMMAND

The)TIME command turns on or off the reporting of wall/CPU elapsed times for an APL function to execute. If off,)TIME turns the reporting on; if on, the reporting is turned off. The first value returned is the elapsed wall time, the second value is the CPU time.

An example of the)TIME command is shown below:

```
      )TIME
NOW ON
TIMES:      .0,      .009
      A←⍳1000
TIMES:      .5,      .218
      B←A*÷12
TIMES:     4.9,     3.534
      C←B*4
TIMES:     7.7,     4.640
      )TIME
NOW OFF
```

TERMINATING AN APL SESSION

An APL session may be terminated with either the)OFF or)CONTINUE commands.

If the)OFF command is used, the active workspace is discarded and, if it has not been saved with the)SAVE command, is not retrievable.

The)CONTINUE command terminates the session and saves the active workspace under the name CONTINUE.

Examples of the)OFF and)CONTINUE commands are shown below:

```
          )OFF
:LISTF

FILENAME

JWSAVE     WS2     WS4


          )CONTINUE
:LISTF

FILENAME

CONTINUE     JWSAVE     WS2     WS4
  ↑
  └──────────────────────── CONTINUE file saved
```

ERROR MESSAGES

SECTION XI

Table 12-1 contains error messages produced by APL\3000. Table 12-2 contains file system (FCHECK) error messages and the corresponding APL\3000 error numbers.

Table 12-1. APL\3000 Error Messages

TERSE	VERBOSE
TRANSLATION ERRORS	
SYNTAX ERROR	CONSTANT ERROR
SYNTAX ERROR	COMMENT ERROR
DOMAIN ERROR	EXPONENT OVERFLOW
LABEL ERROR	DUPLICATE LABEL
DEFN ERROR	DUPLICATE NAME IN HEADER
SYNTAX ERROR	SYNTAX ERROR
SYNTAX ERROR	NON-EXISTENT CONTROL STRUCTURE
LABEL ERROR	CASE LABEL TOO BIG
LABEL ERROR	REAL CASE LABEL
DOMAIN ERROR	CASE RANGE TOO BIG
DOMAIN ERROR	CASE RANGE MUST BE INTEGER
SYNTAX ERROR	DUPLICATE DEFAULT CASE
LABEL ERROR	DUPLICATE CASE LABEL
DEFN ERROR	MISSING NAME
DEFN ERROR	TOO MANY NAMES
DEFN ERROR	ILLEGAL IN HEADER
DEFN ERROR	LOCAL LIST ERROR
SYNTAX ERROR	ERROR IN EMPTY STATEMENT
DEFN ERROR	KEYWORD ' PROCEDURE ' MISSING

Table 12-1. APL\3000 Error Messages (continued)

DEFN ERROR	FUNCTION ALREADY EXISTS
CONST BLK OVFLW	CONSTANT BLOCK OVERFLOW
SCODE BLK OVFLW	SECCODE BLOCK OVERFLOW
CMNT BLK OVFLW	COMMENT BLOCK OVERFLOW
EXECUTION ERRORS	
CHARACTER ERROR	ILLEGAL CHARACTER
SYNTAX ERROR	SYNTAX ERROR
DEPTH ERROR	FUNCTION CALLS TOO DEEP
DOMAIN ERROR	DOMAIN ERROR
DEFN ERROR	DEFN ERROR
INDEX ERROR	INDEX ERROR
LABEL ERROR	LABEL ERROR
LENGTH ERROR	LENGTH ERROR
RANK ERROR	RANK ERROR
SYMBOL TABLE FULL	TOO MANY SYMBOLS IN WS
SYSTEM ERROR	SYSTEM ERROR
VALUE ERROR	VALUE ERROR
WS FULL	WORKSPACE FULL
DOMAIN ERROR	INTEGER DIVIDE BY 0
DOMAIN ERROR	REAL DIVIDE BY ZERO
DOMAIN ERROR	INTEGER OVERFLOW
DOMAIN ERROR	REAL OVERFLOW
DOMAIN ERROR	INTEGER UNDERFLOW
DOMAIN ERROR	REAL UNDERFLOW
NONCE ERROR	NOT YET IMPLEMENTED
SYNTAX ERROR	FUNCTION VALENCE CHANGED

Table 12-1. APL\3000 Error Messages (continued)

INCORRECT COMMAND	INCORRECT COMMAND
INTERRUPT	INTERRUPT
BINDING ERROR	BINDING ERROR
DOMAIN ERROR	NON-EXISTENT ENVIRONMENT
DOMAIN ERROR	ENVIRONMENT NOT ON STACK
NO ENVIRONMENTS	ENVIRONMENT LIMIT EXHAUSTED
ASSERTION FAILED	ASSERTION FAILED
EDITOR ERRORS	
INTERNAL OVERFLOW	INTERNAL OVERFLOW
SYNTAX ERROR	SYNTAX ERROR
MUST BE APL OR APLGOL	MUST BE APL OR APLGOL
ILLEGAL LINE RANGE	ILLEGAL LINE RANGE
NUMBER TOO LARGE	NUMBER TOO LARGE
TOO MANY DECIMAL POINTS	TOO MANY DECIMAL POINTS
ILLEGAL NAME	ILLEGAL NAME
NUMBER TOO LARGE	NUMBER TOO LARGE
MISSING COLON	MISSING COLON
MISSING START LINE	MISSING START LINE
MISSING LINE COUNT	MISSING LINE COUNT
MISSING DELTA	MISSING DELTA
MISSING ASSIGNMENT	MISSING ASSIGNMENT
MISSING DELTA VALUE	MISSING DELTA VALUE
ILLEGAL DELTA VALUE	ILLEGAL DELTA VALUE
NO SUCH COMMAND	NO SUCH COMMAND
CHANGE STRING NOT DEFINED	CHANGE STRING NOT DEFINED
FIND STRING NOT DEFINED	FIND STRING NOT DEFINED

Table 12-1. APL\3000 Error Messages (continued)

PATTERN STRING NOT DEFINED	PATTERN STRING NOT DEFINED
NO LINE NUMBER ROOM	NO LINE NUMBER ROOM
NONCE ERROR	NOT YET IMPLEMENTED
LINE NOT FOUND	LINE NOT FOUND
STRING NOT FOUND	STRING NOT FOUND
WS FULL	WORKSPACE FULL
LIBRARY COMMAND ERRORS	
SYSTEM ERROR	UNEXPECTED FILE ERROR
WS LOCKED	INCORRECT PASSWORD SUPPLIED
WS NOT FOUND	WORKSPACE DOES NOT EXIST
FILE NOT WS	FILE IS NOT AN APL WORKSPACE
NO SPACE	NO DISC SPACE AVAILABLE
NO SUCH LIB	ACCOUNT OR GROUP NON-EXISTENT
BAD WSID	INCORRECT WORKSPACE NAME
ACCESS ERROR	CANNOT OBTAIN EXCLUSIVE ACCESS
ACCESS ERROR	SECURITY DISALLOWS ACCESS
ACCESS ERROR	FILE CREATOR CONFLICT
NO SPACE	DIRECTORY OVERFLOW
FILE EXISTS	NOT SAVED - FILE ALREADY EXISTS
UNNAMED WS	NOT SAVED - WORKSPACE HAS NO NAME
INTERRUPT	INTERRUPT - WS NOT LOADED
INTERRUPT	INTERRUPT - WS NOT SAVED
EDITOR ERRORS	
WILL NOT OVERLAY LINE	WILL NOT OVERLAY LINE
INTERRUPT	

Table 12-1. APL\3000 Error Messages (continued)

FILE SYSTEM ERRORS		
	FILE SYSTEM ERROR	FILE SYSTEM ERROR
	SYSTEM ERROR	UNEXPECTED FILE ERROR
	WS LOCKED	INCORRECT PASSWORD SUPPLIED
	WS NOT FOUND	WORKSPACE DOES NOT EXIST
	FILE NOT WS	FILE IS NOT AN APL WORKSPACE
	NO SPACE	NO DISC SPACE AVAILABLE
	NO SUCH LIB	GROUP OR ACCOUNT NUMBER
	BAD WSID	INVALID WORKSPACE NAME
	ACCESS ERROR	CANNOT OBTAIN EXCLUSIVE ACCESS
	ACCESS ERROR	SECURITY DISALLOWS ACCESS
	ACCESS ERROR	FILE CREATOR CONFLICT
	NO SPACE	DIRECTORY OVERFLOW
	FILE EXISTS	NON-WS FILE BY THAT NAME
	UNNAMED WS	NOT SAVED - WORKSPACE HAS NO NAME
¯1000	FILE ALREADY OPENED	FILE ALREADY OPENED
¯1001	FILE NOT OPEN	FILE NOT YET OPENED
¯1002	STACK OVFLW	RECORD SIZE TOO LARGE

Table 12-2. File System (FCHECK) Error Codes

ERROR NUMBER	MEANING	APL ERROR NUMBER
20	Invalid operation	100
21	Data parity error.	100
22	Software time-out.	100
23	End of tape.	100
24	Unit not ready.	100
25	No write ring on tape.	100
26	Transmission error.	100
27	Input/output time-out.	100
28	Timing error or data overrun.	100
29	Start input/output (SIO) failure.	100
30	Unit failure.	100
31	End of line indicated by special character terminator.	100
32	Software abort of input/output operation.	100
33	Data lost.	100
34	Unit not on-line.	100
35	Data set not ready.	100
36	Invalid disc address.	100
37	Invalid memory address.	100
38	Tape parity error.	100
39	Recovered tape error.	100
40	Operation inconsistent with access type.	100
41	Operation inconsistent with record type.	100
42	Operation inconsistent with device type.	100

Table 12-2. File System (FCHECK) Error Codes (continued)

43	Transfer count (tcount) exceeds record size parameter (recsize) when multi-record write (aoption) not specified when file opened.	100
44	FUPDATE intrinsic requested but file is positioned at record zero; FUPDATE must reference last record read but no previous record was read.	100
45	Privileged file violation.	100
46	Insufficient disc file space on all discs in specified device class.	104
47	Input/output error while accessing file label.	100
48	Invalid operation due to multiple file access.	100
49	Unimplemented function.	100
50	Referenced account does not exist.	105
51	Referenced group does not exist.	105
52	Referenced permanent file not found in system directory.	102
53	Referenced temporary file not found in job directory.	102
54	Invalid file reference.	106
55	Referenced device is not available.	100
56	Device specification is invalid or undefined.	100
57	Virtual memory insufficient for specified file.	100
58	File not passed; typically caused by request for $OLDPASS when there is no $OLDPASS.	100
59	Standard label violation.	100
60	Global RIN not available.	100
61	Group disc file space exceeded.	104
62	Account disc file space exceeded.	104
63	User does not have non-sharable device (ND) capability required by this operation.	100

Table 12-2. File System (FCHECK) Error Codes (continued)

64	User does not have multiple RIN (MR) capability required by this operation.	100
71	Too many files opened for process.	100
72	Invalid file number.	100
73	Bounds check violation.	100
80	Configured maximum number of spoolfile sectors exceeded by this output request.	100
81	SPOOL class not defined in system.	100
82	Insufficient space in SPOOL class for this input/output request.	100
83	Extent size greater than 65K (maximum allowed).	100
84	Device in SPOOL class is down; that is, next extent in this spoolfile is on device that is not available to system.	100
85	Requested operation inconsistent with spooling; for example, an attempt to read hardware status.	100
86	Spool process internal error.	100
87	Offset to data is greater than 255 sectors.	100
89	Power failure.	100
90	Calling process requested exclusive access to file being accessed by another process.	107
91	Calling process requested access to file to to which another process has exclusive access.	107
92	Lockword violation.	101
93	Security violation.	108
94	Conflict in use of FRENAME intrinsic because user is not the creator.	109
100	Duplicate permanent file name in system file directory.	102
101	Duplicate temporary file name in job file directory.	102

Table 12-2. File System (FCHECK) Error Codes (continued)

102	Directory input/output error.	100
103	System directory overflow.	110
104	Job directory overflow.	110
105	Illegal variable block structure.	
106	Extent size exceeds 65K (maximum allowed).	100
107	Offset to data exceeds 255 sectors.	100
110	Attempt to save permanent system file in job (temporary) directory.	100

APL\3000 CHARACTER SET

CHARACTER NAME	FUNCTION OR USE MONADIC	DYADIC	APL SYMBOL	ASCII SYMBOL	3-CHAR "ASCII"	0-ORIGIN ⎕AV INDEX	ASCII DECIMAL	ASCII OCTAL	OVERSTRUCK CHARACTER
Zero	NUMBERS	NAMES	0	0		0	48	60	
One			1	1		1	49	61	
Two			2	2		2	50	62	
Three			3	3		3	51	63	
Four			4	4		4	52	64	
Five			5	5		5	53	65	
Six			6	6		6	54	66	
Seven			7	7		7	55	67	
Eight			8	8		8	56	70	
Nine			9	9		9	57	71	
Space	Separator	↓				10	32	40	
A		NAMES	A	A		11	65	101	
A-underscore			A		"UA	12			A —
B			B	B		13	66	102	
B-underscore			B		"UB	14			B —
C			C	C		15	67	103	
C-underscore			C		"UC	16			C —
D			D	D		17	68	104	
D-underscore			D		"UD	18			D —
E			E	E		19	69	105	
E-underscore			E		"UE	20			E —
F			F	F		21	70	106	
F-underscore			F		"UF	22			F —
G			G	G		23	71	107	
G-underscore			G		"UG	24			G —
H			H	H		25	72	110	
H-underrscore			H		"UH	26			H —
I			I	I		27	73	111	
I-underscore			I		"UI	28			I —
J			J	J		29	74	112	
J-underscore			J		"UJ	30			J —
K			K	K		31	75	113	
K-underscore			K		"UK	32			K —
L			L	L		33	76	114	
L-underscore			L		"UL	34			L —
M		↓	M	M		35	77	115	
M-underscore		NAMES	M		"UM	36			M —

CHARACTER NAME	FUNCTION OR USE MONADIC	DYADIC	APL SYMBOL	ASCII SYMBOL	3-CHAR “ASCII”	0-ORIGIN ⎕AV INDEX	ASCII DECIMAL	ASCII OCTAL	OVERSTRUCK CHARACTER	
N			N	N		37	78	116		
N-underscore			N		“UN	38			N	—
O			O	O		39	79	117		
O-underscore			O		“UN	40			0	—
P			P	P		41	80	120		
P-underscore			P		“UP	42			P	—
Q			Q	Q		43	81	121		
Q-underscore			Q		“UQ	44			Q	—
R			R	R		45	82	122		
R-underscore			R		“UR	46			R	—
S			S	S		47	83	123		
S-underscore			S		“US	48			S	—
T			T	T		49	84	124		
T-underscore			T		“UT	50			T	—
U			U	U		51	85	125		
U-underscore			U		“UU	52			U	—
V			V	V		53	86	126		
V-underscore			V		“UV	54			V	—
W			W	W		55	87	127		
W-underscore			W		“UW	56			W	—
X			X	X		57	88	130		
X-underscore			X		“UX	58			X	—
Y			Y	Y		59	89	131		
Y-underscore			Y		“UY	60			Y	—
Z			Z	Z		61	90	132		
Z-underscore		↓	Z		“UZ	62			Z	—
DELTA		NAMES	∆		“LD	63				
DELTA-Under		NAMES	⍙		“DU	64			∆	—
less		less	<	<		65	60	74		
not greater		not greater	≤		“LE	66				
greater		greater	>	>		67	62	76		
not less		not less	≥		“GE	68				
equal		equal	=	=		69	61	75		
not equal		not equal	≠		“NE	70				

CHARACTER NAME	FUNCTION OR USE MONADIC	DYADIC	APL SYMBOL	ASCII SYMBOL	3-CHAR “ASCII”	0-ORIGIN ⎕AV INDEX	ASCII DECIMAL	ASCII OCTAL	OVERSTRUCK CHARACTER	
or		or	V		“OR	71				
and		and	∧		“ND	72				
tilde	not		~		“NT	73				
epsilon		member	ϵ		“EP	74				
up (arrow)		take	↑	↑or		75	94	136		
down (arrow)		drop	↓		“DP	76				
base		decode	⊥		“BV	77				
top		encode	⊤		“RP	78				
slash		compress	/	/		79	47	57		
slope		expand	\	\		80	92	134		
open paren	Grouping	Grouping	(	(		81	40	50		
close paren	Grouping	Grouping	)	)		82	41	51		
open bracket	Indexing	Indexing	[	[		83	91	133		
close bracket	Indexing	Indexing	]	]		84	93	135		
overbar	neg.	constant	—	#		85	35	43		
right (arrow)	branch		→		“RA	86				
left (arrow)		assign	←	←or		87	95			
del	None	None	∇		“DL	88				
quad	Input, Output, Distinguished Names	Input, Output, Distinguished Names	⎕		“QD	89				
quote	Char. Constants	Char. Constants	'	'		90	39	47		
null		Outer Product	∘		“UT	91				
dot		Operator Number Consts.	•	•		92	46	56		
semicolon	List Separator	List Separator	;	;		93	59	73		
colon	Labels	Labels	:	:		94	58	72		
diamond	Statement	Separator	◊		“DI	95				
bar	neg.	minus	–	–		96	45	55		
plus	conjugate	plus	+	+		97	43	53		
divide	reciprocal	divide	÷		“DV	98				
times	signum	times	X		“TM	99				
query	roll	deal	?	?		100	63	77		
rho	shape	reshape	⍴		“RO	101				
iota	index generator	index of circular	⍳		“IO	102				

CHARACTER NAME	FUNCTION OR USE		APL SYMBOL	ASCII SYMBOL	3-CHAR "ASCII"	0-ORIGIN ⎕AV INDEX	ASCII DECIMAL	ASCII OCTAL	OVERSTRUCK CHARACTER	
	MONADIC	DYADIC								
circle	πtimes	Hyperbolic, etc.	○		"CR	103				
star	exponential	power	*	*		104	52	42		
upstile	ceiling	maximum	⌈		"MX	105				
downstile	floor	minimum	⌊		"MN	106				
stile	magnitude	residue	|		"RD	107				
comma	ravel	catenate	,	,		108	44	54		
log	Natural Logarithm	General Logarithm	⍟		"LG	109			0	X
circle bar	1st coordinate reverse	1st coordinate rotate	⊖		"CD	110			0	—
circle slope	transpose	transpose	⍉		"TP		111		0	\
quote dot	Factorial	Binomial	!	!		112	33	41	'	∘
domino	Matrix Inverse	Matrix Division	⌹		"DM	113			□	%
nor		Nor	⍱		"NR	114			v	~
nand		Nand	⍲		"NA	115			^	~
circle stile	Reverse	Rotate	⌽		"RV	116			0	|
I-beam	None	None	⌶		"IB	117				
del stile		Grade Down	⍒		"GD	118			∇	|
delta stile		Grade up	⍋		"GU	119			Δ	|
quote quad	INPUT	OUTPUT	⍞		"QQ	120			'	□
cap hull	Comment	Comment	⍝		"CM	121			∩	∘
slope bar		1st coordinate Expand	⍀		"BD	122			\	-
slash bar		1st coordinate Compress	⌿		"SD	123			/	-
del tilde	None	None	⍫		"DT	124			∇	~
base null	Extended Execute	Execute	⍎		"CX	125			⊥	∘
top null	Format	Format	⍕		"FT	126			⊤	∘

CHARACTER NAME	FUNCTION OR USE MONADIC DYADIC	APL SYMBOL	ASCII SYMBOL	3-CHAR "ASCII"	0-ORIGIN ⎕AV INDEX	ASCII DECIMAL	ASCII OCTAL	OVERSTRUCK CHARACTER	
out	INTERRUPT	⏀		"OU	127			OUT	
dieresis	NONE	¨		"DR	128				
left tack	NONE	⊢		"LK	129				
right tack	NONE	⊣		"RK	130				
doliar	NONE	$	$		131	36	44		
omega	NONE	ω		"OM	132				
alpha	NONE	α		"AL	133				
open shoe	NONE	⊂		"PS	134				
close shoe	NONE	⊃		"BS	135				
cup	NONE	∪		"SU	136				
cap	NONE	∩		"SI	137				
cnul	null	$\leftarrow^c$	$@^c$		138	0	0		
attn	attention	Y^c	Y^c		139				
linefeed	linefeed	line-feed	J^c		140	10	12		
tab cbel	tab bell	tab G^c	I^c G^c		141 142	9_7	11_7		
csoh	start of heading	A^c	A^c		143	1	1		
cstx	start of text	B^c	B^c		144	2	2		
cetx	end of text	C^c	C^c		145	3	3		
ceot	end of transmission	D^c	D^c		146	4	4		
ceng	enquiry	E^c	E^c		147	5	5		
backspace	backspace	back-	H^c		148	8	10		
cack	acknowledge	F^c	F^c		149	6	6		
cvt	vertical tab	K^c	Kc2c		150	11	13		
cff	form feed	L^c	L^c		151	12	14		
return	return	return	M^c		152	13	15		
cso	shift out	N^c	N^c		153	14	16		
csi	shift in	O^c	O^c		154	15	17		
cdle	data link escape	p^c	p^c		155	16	20		
cdc1	device control l	Q^c	Q^c		156	17	21		
cdc2	device control 2	R^c	R^c		157	18	22		
cdc3	device control 3	S^c	S^c		158	19	23		
cdc4	device control 4	T^c	T^c		159	20	24		
cnak	negative acknowledge	U^c	U^c		160	21	25		
csyn	synchronous idle	V^c	V^c		161	22	26		

CHARACTER NAME	FUNCTION OR USE MONADIC	DYADIC	APL SYMBOL	ASCII SYMBOL	3-CHAR "ASCII"	0-ORIGIN ⎕AV INDEX	ASCII DECIMAL	ASCII OCTAL	OVERSTRUCK CHARACTER	
cetb	end of transmission block		Wc	W^c		162	23	27		
ccan	cancel line		X^c	X^c		163	24	30		
cem	end of medium		Y^c	Y^c		164	25	31		
csub	substitute		Z^c	Z^c		165	26	32		
escape	escape		escape	escape or $[^c$		166	27	33		
cfs	file separator		$\diamond^c$	^{c}		167	28	34		
cgs	group separator		$\{^c$	$]^c$		168	29	35		
crs	record separator		X^c	$\uparrow^c$ or ^{c}		169	30	36		
cus	unit separator		$\wedge^c$	$\_^c$		170	31	37		
delete	delete			delete		171	127	177		
doublequote	NONE			delete		172	34	42		
underbar	NAMES		—		"NL	173				
eol	APL Terminal Control					174				
eof	APL Terminal Control					175				
charerr	unprintable character					176				
pound	NONE			#		177	35	43		
percent				%		178	37	45		
ampersand				&		179	38	46		
atsign				@		180	64	100		
open brace			{	{		181	123	173		
close brace			}	}		182	125	175		
a				a		183	97	141		
b				b		184	98	142		
c				c		185	99	143		
d				d		186	100	144		
e				e		187	101	145		
f				f		188	102	146		
g				g		189	103	147		
h				h		190	104	150		
i				i		191	105	151		
j				j		192	106	152		
k				k		193	107	153		
l				l		194	108	154		

CHARACTER NAME	FUNCTION OR USE MONADIC	DYADIC	APL SYMBOL	ASCII SYMBOL	3-CHAR "ASCII"	0-ORIGIN ⎕AV INDEX	ASCII DECIMAL	ASCII OCTAL	OVERSTRUCK CHARACTER
m	NONE			m		195	109	155	
n	↓			n		196	110	156	
o				o		197	111	157	
p				p		198	112	160	
q				q		199	113	161	
r				r		200	114	162	
s				s		201	115	163	
t				t		202	116	164	
u				u		203	117	165	
v				v		204	118	166	
w				w		205	119	167	
x				x		206	120	170	
y				y		207	121	171	
ASCII not						209	126	176	
ASCII vdash						210	124	174	
grave accent	↓					211	96	140	

APL\3000 PRIMITIVE FUNCTIONS AND OPERATORS

NAME	SYMBOL	SYNTAX
And	∧	X∧Y
Arccosine	○	¯2○X
Arcsine	○	¯1○X
Arctangent	○	¯3○X
Axis operator	[ ]	[expression]
Binomial	!	A!B
Catenate	,	A,B
Ceiling	⌈	A
Compress	/ or ⌿	boolean argument/A
Conjugate	+	+A
Cosine	○	2○X
Deal	?	A?B
Decode	⊥	A⊥B
Divide	÷	A÷B
Drop	↓	A↓B
Encode	⊤	A⊤B
Equal	=	A=B
Execute	⍎	⍎A or A⍎B
Expand	\ or ⍀	boolean argument\A
Exponential	*	*A
Factorial	!	!A
Floor	⌊	⌊A

NAME	SYMBOL	SYNTAX
Format	⍕	⍕A or A⍕B
General logarithm	⍟	A⍟B
Grade down	⍒	⍒A
Grade up	⍋	⍋A
Greater	>	A>B
Hyperbolic arccosine	○	¯6○X
Hyperbolic arcsine	○	¯5○X
Hyperbolic arctangent	○	¯7○X
Hyperbolic cosine	○	6○X
Hyperbolic sine	○	5○X
Hyperbolic tangent	○	7○X
Index generator	⍳	⍳A
Index of	⍳	A⍳B
Indexing	[ ]	A[expression]
Inner product operator	.	Afn1.fn2B
Laminate	,[ ]	A,[fraction]B
Less	<	A<B
Magnitude	|	|A
Matrix divide	⌹	A⌹B
Matrix inverse	⌹	⌹A
Maximum	⌈	A⌈B
Membership	∊	A∊B
Minimum	⌊	A⌊B
Minus	-	A-B
Nand	⍲	X⍲Y

NAME	SYMBOL	SYNTAX
Natural logarithm	⍟	⍟A
Nor	⍱	X⍱Y
Not	~	~A
Not equal	≠	A≠B
Not greater	≤	A≤B
Not less	≥	A≥B
Or	∨	X∨Y
Outer product operator	∘.	A∘.fnB
Pi times	○	○A
Plus	+	A+B
Power	*	A*B
Pythagorean (¯1+X*2)*.5	○	¯4○X
Pythagorean (1+X*2)*.5	○	4○X
Pythagorean (1-X*2)*.5	○	0○X
Quad input	⎕	A←⎕
Quad output	⎕	⎕←A
Quote quad input	⍞	A←⍞
Quote quad output	⍞	⍞←A
Ravel	,	,A
Reciprocal	÷	÷A
Reduction operator	/	primitive function/A
Reshape	⍴	A⍴B
Residue	∣	A∣B
Reversal	⌽ or ⊖	⌽A or ⊖A
Roll	?	?A

NAME	SYMBOL	SYNTAX
Rotate	⌽ or ⊖	A⌽B or A⊖B
Scan operator	\	primitive function\A
Shape	⍴	⍴A
Signum	×	×A
Sine	○	1○X
Take	↑	A↑B
Tangent	○	3○X
Times	×	A×B
Transpose	⍉	A⍉B

APL\3000 SYSTEM COMMANDS

APPENDIX C

NAME	SYNTAX
Bind	)BIND
Clear	)CLEAR
Continue	)CONTINUE
Copy	)COPY [namelist]
Depth	)DEPTH num
Drop	)DROP wsname
Edit	)EDIT [name]
Erase	)ERASE [namelist]
Files	)FILES [groupname.acctname]
Functions	)FNS [letter]
Help	)HELP [cmdname]
Library	)LIB [groupname[.accountname]]
Load	)LOAD wsname
MPE	)MPE
Namelist - name [name] [name]. . .[name]	
Off	)OFF
Protected copy	)PCOPY wsname [namelist]
Reset	)RESET [n]
Resume	)RESUME
Save	)SAVE wsname
State indicator	)SI [n]
State indicator with variables	)SIV [n]

NAME	SYNTAX
Time	)TIME
Terminal type	)TERM [termtype]
Terse	)TERSE
Variables	)VARS [letter]
Verbose	)VERBOSE
Workspace identification	)WSID [wsname]

Wsname = workspace identification [/lockword] [groupname[.accountname]

APL\3000 SYSTEM VARIABLES

APPENDIX D

NAME	FORM	SYNTAX
Account information	⎕AI	⎕AI
Alphabet	⎕A	⎕A
Assertion level	⎕AL	⎕AL[←value]
Atomic vector	⎕AV	⎕AV
Backspace	⎕B	⎕B
Branch trace	⎕BT	⎕BT
Comparison tolerance	⎕CT	⎕CT[←value]
Digits	⎕D	⎕D
Escape	⎕E	⎕E
Execution trace	⎕XT	⎕XT[←value]
Horizontal tab setting	⎕HT	⎕HT[←integer vector]
Index origin	⎕IO	⎕IO[←value]
Language	⎕LA	⎕LA←['APL' 'APLGOL']
Latent expression	⎕LX	⎕LX[←'expression']
Line counter	⎕LC	⎕LC
Linefeed	⎕L	⎕L
Null	⎕N	⎕N
Printing precision	⎕PP	⎕PP[←value]
Printing width	⎕PW	⎕PW[←value]
Random link	⎕RL	⎕RL[←value]
Return	⎕R	⎕R

NAME	FORM	SYNTAX
Stack names	⎕SN	⎕SN
Tab	⎕T	⎕T
Terminal type	⎕TT	⎕TT[←<u>termtype</u>]
Time Stamp	⎕TS	⎕TS
Virtual memory	⎕VM	⎕VM[←<u>integer vector</u>]
Work area available	⎕WA	⎕WA
Workspace identification	⎕WI	⎕WI

APPENDIX

E

APL\3000 SYSTEM FUNCTIONS

NAME	FORM	SYNTAX
Canonical representation	⎕CR	⎕CR 'name'

Capture stack environment	⎕CSE	A←F ⎕CSE C S D
		A = assigned environment number
		F = function name
		C = count
		S = starting environment
		D = desired environment number

Convert	⎕CV	control ⎕CV data
Delay	⎕DL	⎕DL seconds
Expunge	⎕EX	⎕EX 'name'
Function establishment	⎕FX	⎕FX name
Monitor values	⎕MV	⎕MV 'name'
Name classification	⎕NC	⎕NC 'name'
Name list	⎕NL	['letters'] ⎕NL integers
Query monitor	⎕QM	⎕QM 'name'
Query stop	⎕QS	⎕QS 'name'
Query trace	⎕QT	⎕QT 'name'

Release stack environment	⎕RSE	RL ⎕RSE EL
		RL = released stack environment
		EL = environment list

Reset monitor	⎕RM	⎕RM 'name'
Reset stop	⎕RS	⎕RS 'name'
Reset trace	⎕RT	⎕RT 'name'
Set monitor	⎕SM	⎕SM 'name'

NAME	FORM	SYNTAX
Set stop	⎕SS	⎕SS 'name'
Set trace	⎕ST	⎕ST 'name'
Shared variable control	⎕SVC	⎕SVC ['processid']
Shared variable offer	⎕SVO	['processid'] ⎕SVO 'shared variable id'
Shared variable retract	⎕SVR	⎕SVR 'shared variable id'
Shared variable query	⎕SVQ	⎕SVQ ['processid']
Vector representation	⎕VR	⎕VR 'name'

APL\3000 EDIT INSTRUCTION SYNTAX

APPENDIX F

```
A[DD] ┌linespec┐   [delta]
      │string  │
      └        ┘

B[RIEF]

C[HANGE]  [character [patternstring] character [changestring]
           character [rangelist]]

CO[PY]  lineblock

lineblock = linerange  ⎧  :  ⎫linespec   [delta]
                       ⎨  ,  ⎬
                       ⎩blank⎭

⎧CU[RSOR]⎫ ┌linespec ┐
⎩*       ⎭ │+ integer│
           │- integer│
           │string   │
           └         ┘
D[ELETE]   ┌string   ┐
           │rangelist│
           └         ┘

delta = [,] linenumber

⎧DELT[A]⎫  ⎧=⎫   [decimalnumber]
⎩   Δ   ⎭  ⎩←⎭

END  ┌APL   ┐
     │APLGOL│
     └      ┘

FIND [string] [rangelist]


⎧H[ELP] ⎫ [instruction]
⎨EXPLAIN⎬
⎩?      ⎭

linerange = ┌linespec                                    ┐
            │<linespec> <separator> <linespec>           │
            │<linespec> <separator>                      │
            │<separator> <linespec>                      │
            │separator                                   │
            └ALL                                         ┘
```

```
linespec =  ┌line number┐
            │FIRST      │
            │LAST       │
            │CURSOR     │
            └*          ┘

L[IST]  ┌rangelist┐
        │string   │
        │ALL      │
        │FIRST    │
        └LAST     ┘

LOCK  ┌APL   ┐
      └APLGOL┘

MAT[RIX]  [variablename]

M[ODIFY]  ┌string   ┐
          └rangelist┘

QUIT

rangelist = ┌linerange  [,linerange]. . . [,linerange]┐
            └rnge [,rlist]                             ┘

R[EPLACE]  ┌string   ┐   [delta]
           └rangelist┘

RES[EQUENCE]  lineblock

separator = ┌/┐
            └|┘

string = <character> <text not containing character> <character>

UNDO  [integer] [grainspec]

grainspec = ⎧  ,   ⎫ ⎧L[INES]   ⎫
            ⎨  |   ⎬ ⎩C[OMMANDS]⎭
            ⎩blank ⎭

VEC[TOR]  [variablename]

VER[BOSE]
```

APPENDIX G

APLGOL STATEMENT SYNTAX

```
ASSERT expression : expression

BEGIN  statement list  END

CASE expression OF integer constant
BEGIN subcase list + END CASE

EXIT  [expression]

FOREVER DO statement

HALT  [expression]

IF expression DO statement

IF expression THEN statement ELSE
statement

NULL

REPEAT statement list UNTIL expression

WHILE expression DO statement
```

SYSTEM SUPPLIED UTILITY SHARED VARIABLES

APPENDIX H

PROCESSOR:	UTIL
VARIABLES:	VERBOSE FLAG
	INPUTCONTROL

VERBOSE FLAG = Boolean. 1 if error messages is in VERBOSE mode; 0 otherwise. Can be set dynamically.

INPUTCONTROL = Takes as input a 1 or 2 element vector of integers from ¯32768 to 32767 (unit or scalar extends to 1-element vector). If second value is omitted, the system sets it to 0.

The two values are used as the two parameters for the FCONTROL intrinsic on the standard APL input file 'APLIN'.

After FCONTROL executes, the value of the second parameter (which may be changed by MPE) is saved.

For a READ, the value saved is returned (saved from the last WRITE call) and initialized to 0 so that a READ before any WRITE will return an answer.

INDEX

READER COMMENT SHEET

HP 3000 Series Computer System
APL\3000 Reference Manual

32105-90002 November 1976

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

Is this manual technically accurate?

Did you have any difficulty in understanding concepts or wording? Where?

Is the format of this manual convenient in size, arrangement, and readability? What improvements would you suggest?

Other comments?

FROM:

Name ______________________

Company ______________________

Address ______________________

____

____

FOLD FOLD

FIRST CLASS
PERMIT NO. 1020
SANTA CLARA
CALIFORNIA

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States. Postage will be paid by

Publications Manager, Product Support Group
Hewlett-Packard Company
General Systems Division
5303 Stevens Creek Boulevard
Santa Clara, California 95050

FOLD FOLD

HEWLETT PACKARI

Sales and service from 172 offices in 65 countrie
5303 Stevens Creek Blvd., Santa Clara, California 95050

Part No. 32105-90002
Printed in U.S.A. 11/76