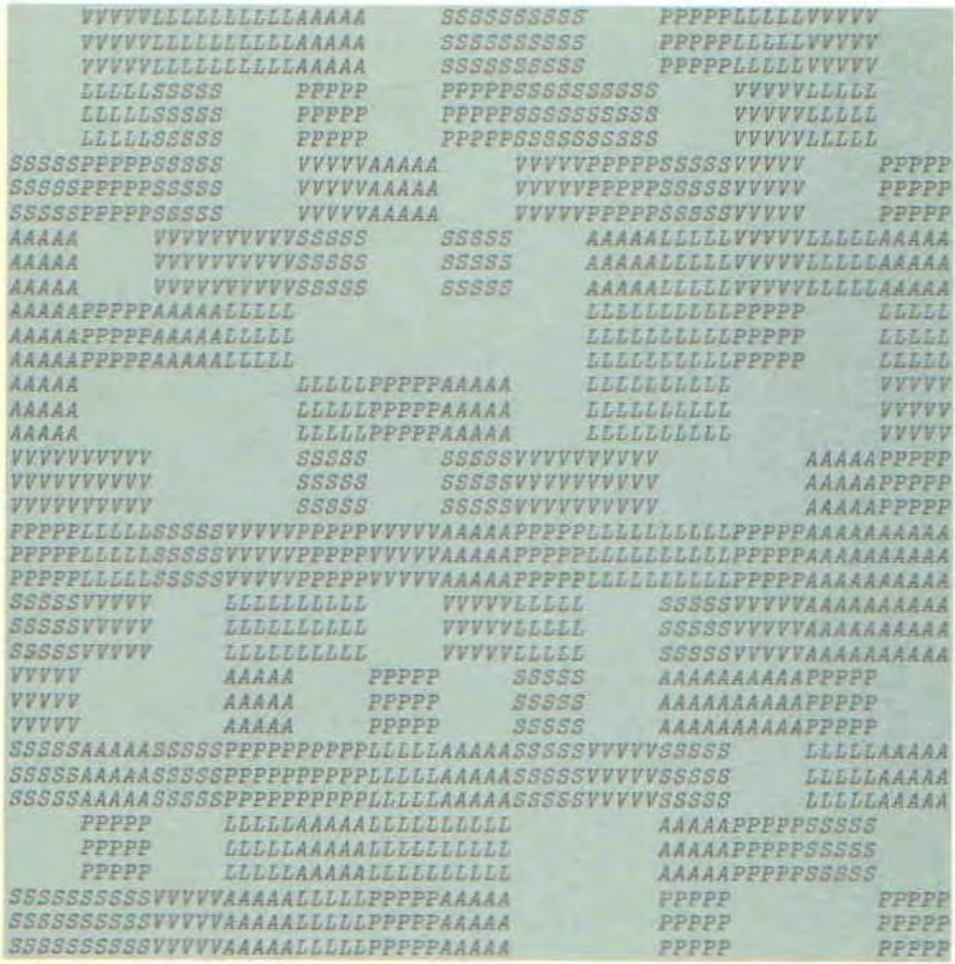
APLSV User's Manual

APL SHARED VARIABLE SYSTEM



IBM Philadelphia Scientific Center

APLSV User's Manual

A. D. Falkoff

K. E. Iverson

ACKNOWLEDGEMENT

The enhancements to APL described herein are the result of more than six years of personal experience with APL\360, and countless comments and suggestions from many other users. The shared variable concept goes back in principle to early publications by Falkoff and Iverson, but it was R. H. Lathwell who recognized the practical potential of the idea and, with L. A. Morrow, designed an effective data processing facility based upon it. The system was implemented by Lathwell, Morrow, J. A. Brown, and C. F. Shen, all of whom contributed to the design of the other new features as well. R. J. Creasy first proposed the use of surrogate names.

This manual is also available as
IBM Publication SH20-1460.

TABLE OF CONTENTS

APL SHARED VARIABLE SYSTEM

PART 1
INTRODUCTION

This APL system is fundamentally the same as that described in the APL\360 User's Manual (IBM Publication GH20-0906-1). Except as noted herein, this document also describes the present system, and it will be assumed that the reader is familiar with it or has it available for reference.

The major difference is the addition of a shared variable facility which provides simple and effective channels of communication between programs running at different terminals, and also forms a basis for managing files and high speed input and output from an APL terminal. The facility itself is managed by a group of dynamically executable system functions provided for this purpose. This system also differs in the following respects:

- There are several minor changes and additions to the system commands and primitive functions, changes in certain aspects of system behavior, and improvements in the efficiency of execution.

- The scan operator provides efficient representation and execution of algorithms which otherwise require iteration or the generation of relatively large arrays.

- The execute and format functions provide efficient conversion between character arrays and numerical arrays, as well as a number of other desirable actions.

- The canonical representation of a function definition is established (as a character matrix), and system functions are provided for conversion between such a representation and a defined function, making possible the storage of functions as data and the generation and application of defined functions under program control.

- System variables, a special instance of shared variables, are introduced to control parameters such as index origin and printing width, and to provide information such as time of day and computer time used. The ad hoc facilities represented by I19 through I28 and the workspace functions are now redundant but remain in the system to permit a smooth transition. The system variables also provide one new facility, a latent expression which is automatically executed when a workspace is activated.

- An auxiliary processor called TSIO is provided to give the APL user convenient control of high-speed printers, files, and other system facilities through the medium of shared variables. TSIO is described in IBM Publication SH20-1463.

PART 2
GENERAL SYSTEM CHANGES

CHANGES IN KEYBOARD ENTRY AND IN OUTPUT

Automatic closing of open quote. If a carriage return is entered in an open quote (i.e., before an opening quote is paired by a closing quote) the computer automatically types the closing quote on the next line. The situation then is exactly as if the user had typed the closing quote, i.e., the carriage return is the last character of the string entered, and the user can backspace and erase the quote by an attention signal if desired.

Character errors. If character errors occur in an input line, the line is printed out up to the first such error, at which point the keyboard unlocks to allow further entry as if the printed line had been entered from the keyboard.

No commands executed in function definition. A system command entered in function definition mode is no longer executed directly but is entered as a line of the function definition. In execution, such a line will be treated as an APL statement and will invoke an appropriate error message.

Escape from literal input. Overstruck *O U T* interrupts execution but no longer causes an exit from the function.

Extended printing width. The printing width (as set by the *WIDTH* command and other facilities) can now be set to 390.

Bare output. Normal output includes a concluding carriage return so that the succeeding entry (either input or output) will begin at a standard position on the following line. Bare output, denoted by expressions of the form ⍞←*X*, does not include this gratuitous carriage return if it is followed either by another bare output or by character input (of the form *X*←⍞). Character input following a bare output is treated exactly as if the user had spaced over to the position occupied at the conclusion of the bare output, i.e., the characters entered in response will normally be prefixed by a number of space characters. For example:

```
      ∇ F
[1]   ⍞←'TRUE OR FALSE: THE SQUARE OF '
[2]   ⍞←?4
[3]   ⍞←' IS '
[4]   ⍞←(?4)*2
[5]   X←⍞ ∇

      F
TRUE OR FALSE: THE SQUARE OF 2 IS 9FALSE
      X
                                FALSE
```


The carriage returns normally occasioned by the page width limitation setting are also absent from bare output.

Because any expression of the form ⍞←*X* entered at the keyboard (rather than being executed within a defined function) is necessarily followed by another keyboard entry, it is concluded by a carriage return and its effect is therefore indistinguishable (except for possible page width limitations) from the effect of the corresponding normal output.

Heterogeneous output. Parentheses surrounding a heterogeneous output statement are no longer permitted. They can be systematically removed from any unlocked function by user-defined editing functions, employing the dynamic function definition capability provided by the functions ⎕*CR* and ⎕*FX* described in Part 4.

The facility for heterogeneous output does not represent a proper APL function; in particular, its result cannot be assigned to a name. It was introduced as an early convenience to obviate awkward conversions of numbers to character representations. The format function described in Part 3 now provides such conversions conveniently, and output combinations are easily formulated as proper APL objects. Therefore, the user is advised to avoid the use of the heterogeneous output facility.

CHANGES IN ERROR HANDLING

Depth error. Depth errors no longer occur.

Errors in locked functions. A locked function is treated essentially as primitive and its execution can invoke only a *DOMAIN* error, although conditions (such as *WS FULL*) arising from system limitations will also be reported. Moreover, execution of a locked function is terminated by any error occurring within it, or by a double attention.

CHANGES IN FUNCTION DEFINITION

Line editing. An entry of the form '*N*⎕*M*' while in function definition mode now invokes the following special action for the case when *M* is zero: line *N* is displayed with the carrier resting at the end of the line, as if the line had just been entered from the keyboard. At this point the line can be extended, or modified by backspace and attention, in the usual manner.

Because of the change in handling of character errors (noted under Changes in Keyboard Entry), a deliberate character error can no longer be used to abort the revision after the second display of a line in the editing process.

Stop and trace in locked functions. Settings of stop and trace are automatically nullified when a function definition is locked.

Display of comments. Comment lines are, like lines with labels, exdented one space to the left in the display of function definitions.

CHANGES IN PRIMITIVE FUNCTIONS

Monadic transpose. The monadic transpose now reverses the order of all coordinates rather than interchanging only the last two. Formally, it is defined in terms of the dyadic transpose as follows: ⍉*A* is equivalent to (⍳⍴⍴*A*)⍉*A*.

With this change the identity *M*+.×*N*←→⍉(⍉*N*)+.×⍉*M* which held for matrices *M* and *N* now holds for higher-dimensional arrays. Indeed, the corresponding identity holds for any inner product f.g if g is commutative.

The residue function. The residue function was previously defined to depend only on the absolute value of its left argument. It is now defined as follows:

1. If *A*=0 then *A*|*B* is equal to *B*.
2. If *A*≠0 then *A*|*B* lies between *A* and zero (being permitted to equal zero but not *A*) and is equal to *B*-*N*×*A* for some integer *N*.

For example:

```
      A←3 0 ¯3
      B←¯6 ¯5 ¯4 ¯3 ¯2 ¯1 0 1 2 3 4 5 6
      A∘.|B
 0  1  2  0  1  2  0  1  2  0  1  2  0
¯6 ¯5 ¯4 ¯3 ¯2 ¯1  0  1 ¯2 ¯1  0 ¯2 ¯1
 0 ¯2 ¯1  0 ¯2 ¯1  0 ¯2 ¯1  0 ¯2 ¯1  0

      X←21.824
      .01|X
0.004
```

The new definition of residue can be stated formally as follows: *A*|*B*←→*B*-*A*×⌊*B*÷*A*+*A*=0

Encode. The definition of the encode function ⊤ is based on the residue function in the manner specified by the following function for vector *A* and scalar *B*:

```
    ∇ Z←A E B
[1]   Z←0×A
[2]   I←⍴A
[3]  L:→(I=0)/0
[4]   Z[I]←A[I]|B
[5]   →(A[I]=0)/0
[6]   B←(B-Z[I])÷A[I]
[7]   I←I-1
[8]   →L
    ∇
```

The definition of the encode function for a left argument having one or more negative elements is therefore affected by the change in the definition of residue. For example:

```
      2 2 2⊤13
1 0 1
      ¯2 ¯2 ¯2⊤13
¯1 ¯1 ¯1
      2 0 2⊤13
0 6 1
```

```
      2 2 2⊤¯13
0 1 1
      ¯2 ¯2 ¯2⊤5
¯1 ¯1 ¯1
      ¯2 2 ¯2⊤5
0 1 ¯1
```

Generalized matrix product and matrix divide. The domain of the ⌹ function described in the APL\360 User's Manual (IBM Publication GH20-0906-1) has been extended slightly to include vector and scalar arguments. This section defines the extensions, and also provides a more comprehensive discussion of the function and its potential applications.

The domino (⌹) represents two functions which are useful in a variety of problems including the solution of systems of linear equations, determining the projection of a vector on the subspace spanned by the columns of a matrix, and determining the coefficients of a polynomial which best fits a set of points in the least-squares sense.

When applied to a non-singular matrix *A* the expression ⌹*A* (monadic) yields the inverse of *A*, and the expression *X*←*B*⌹*A* (dyadic) yields a value of *X* which satisfies the relation ∧/,*B*=*A*+.×*X* and is therefore the solution of the system of linear equations conventionally represented as Ax=b. In the following examples the floor function is used only to obtain a compact display:

```
      A←(⍳4)∘.≥⍳4
      A                  ⌊⌹A                ⌊A+.×⌹A
1 0 0 0            1  0  0  0        1 0 0 0
1 1 0 0           ¯1  1  0  0        0 1 0 0
1 1 1 0            0 ¯1  1  0        0 0 1 0
1 1 1 1            0  0 ¯1  1        0 0 0 1
```

```
      B←1 3 6 10
      X←B⌹A
      B                X                A+.×X            (⌹A)+.×B
1 3 6 10       1 2 3 4          1 3 6 10       1 2 3 4

      C←4 2⍴1 2 3 5 6 9 10 14
      Y←C⌹A
      C                ⌊Y               ⌊A+.×Y           ⌊(⌹A)+.×C
 1  2          1 2              1  2          1 2
 3  5          2 3              3  5          2 3
 6  9          3 4              6  9          3 4
10 14          4 5             10 14          4 5
```

The final example above shows that if the left argument is a matrix C, then $C⌹A$ yields a solution of the system of equations for each column of C.

If A is non-singular and if I is an identity matrix of the same dimension, then the matrix inverse $⌹A$ is equivalent to the matrix divide $I⌹A$. More generally, for any matrix P the expression $⌹P$ is equivalent to the expression $((\iota R)\circ.=\iota R)⌹P$, where R is the number of rows in P.

The domino functions apply more generally to singular and non-square matrices, and to vectors and scalars; any argument of rank greater than 2 is rejected (*RANK ERROR*). For matrix arguments A and B the expression $X←B⌹A$ is executed only if

1. A and B have the same number of rows, and

2. the columns of A are linearly independent.

If the expression $X←B⌹A$ is executable, then ρX is equal to $(1↓\rho A),1↓\rho B$ and X is determined so as to minimize the value of the expression $+/,(B-A+.\times X)\star 2$.

The domino functions apply to vector and scalar arguments as follows: except that the shape of the result is determined as specified above, a vector is treated as a one-column matrix (since a one-rowed matrix of more than one column would be rejected by condition 2 above) and a scalar is treated as a one-by-one matrix. In the case of scalar arguments X and Y, the expression $⌹Y$ is equivalent to $\div Y$ and, except that it yields a domain error for the case $0⌹0$, the expression $X⌹Y$ is equivalent to $X\div Y$.

Although the following examples of the use of ⌹ add nothing to its definition, they may be of interest to readers familiar with problems of polynomial fitting and of geometry.

The use of ⌹ for a singular argument can be illustrated as follows: if X is a vector and $Y←F\ X$, then the expression $Y⌹X\circ.\star 0,\iota D$ yields the coefficients of the polynomial of degree D which best fits (in the least squares sense) the function F at the points X.

The general definition of $B⌹A$ has certain useful geometric interpretations. If B is a vector and A is a matrix, then saying that $+/(B-A+.\times B⌹A)\star 2$ is a minimum is equivalent to saying that the length of the vector $B-A+.\times B⌹A$ is a minimum. But $A+.\times B⌹A$ is a point in the space spanned by the column vectors of A and is therefore the point in this space which is closest to B. In other words, $P←A+.\times B⌹A$ is the projection of B on the space spanned by the columns of A. Moreover, the vector $B-P$ must be normal to every vector in the space; in particular, $(B-P)+.\times A$ is a zero vector.

If A and B are single-column matrices, then $B⌹A$ is a 1 by 1 matrix and $A+.\times B⌹A$ is equivalent to $A\times S$, where S is the scalar $''\rho B⌹A$. If A and B are vectors, then $B⌹A$ is a scalar and the projection of B on A is therefore given by the simpler expression $A\times B⌹A$. For example:

```
      A←4.5 1.7
      B←2 5
      P←A×B⌹A
      P
3.403197926 1.28565255
      N←B-P
      N
¯1.403197926 3.71434745
      N+.×A
2.44249065435¯15
```

Similar analysis shows that if A is a vector then $⌹A$ is a vector in the direction of A; that is, $⌹A$ is equal to $S\times A$ for some scalar S. Moreover, $A+.\times⌹A$ is equal to 1. In other words, $⌹A$ is the "image" of the vector A obtained by inversion in the unit circle (or sphere).

CHANGES IN SYSTEM COMMANDS

Communication commands. There are many occasions when a user may wish to be undisturbed by messages arriving at the terminal; for example, when producing finished output, or while concentrating on a difficult problem. Even under such circumstances, however, it may be necessary to communicate briefly with another port.

The command *)MSG OFF* blocks any message except one coming in response to a transmitted message in which a reply was requested (that is, a message of the form *)OPR* or *)MSG*) and arriving before the keyboard is again unlocked. Any blocked message is treated as if the keyboard were continuously unlocked, and it will so appear to the sender.

The command *)MSG ON* restores the acceptance of messages and causes the last public address message, if any, to be printed.

Copy commands. The commands *)COPY* and *)PCOPY* now accept a multiplicity of object names. The response lists separately the objects not found (headed by *NOT FOUND*) and those found but not copied (headed by *NOT COPIED*).

Erase command. The erase command now acts on any global object, and no longer distinguishes between pendent functions and others. Problems that may possibly arise from erasing a pendent function are forestalled by the response *SI DAMAGE*, which warns the user to take appropriate action before resuming function execution.

Save command. An attempt to save a workspace during execution of a function, as had been possible through an explicit command during a request for ⍞ input or a forced disconnect in ⎕ input, will result in the function's being interrupted prior to the execution of the command.

When the workspace is next activated, therefore, function execution will not automatically resume. Such automatic resumption can be invoked by the system variable *⎕LX* defined in Part 4.

Symbol table size. The command *)SYMBOLS* without a number prints the current number of names accommodated.

Workspace identification. The command *)WSID* can be used to set a lock as well as the workspace name, using the same form as the *)SAVE* command. Used as an inquiry, *)WSID* will continue to return only the workspace identification.

Privacy. In the interest of privacy, a workspace saved with the name *CONTINUE* cannot be activated or copied by a user other than the one in whose library it has been saved.

Public libraries. Workspaces named *CONTINUE* cannot be stored in public libraries.

Local function names. As a result of the introduction of the system function *⎕FX* (defined in Part 4), local names may now refer to functions as well as variables. Consequently, the phrase "functions and global variables" occurring in the APL\360 User's Manual should now be read as "global functions and variables".

PART 3
NEW PRIMITIVE FUNCTIONS

SCAN

For any dyadic function α and any vector *X*, the α-scan of *X* (denoted by α*X*) yields a result *R* of the same shape as *X* such that *R*[*I*] is equal to α/*I*↑*X*. For example:

```
      +\1 2 3 4 5 6
1 3 6 10 15 21
      ρ+\⍳0
0
```

The scan is extended to any array as follows: if *R*←α\[*I*]*A*, then ρ*R* equals ρ*A* and the vectors along the *I*th coordinate of *R* are the α-scans over the vectors along the *I*th coordinate of *A*; scan applied to a scalar yields the scalar unchanged.

The following examples show some interesting uses of the scan:

```
      L←0 0 1 0 1 0 1
      L
0 0 1 0 1 0 1
      ~L
1 1 0 1 0 1 0
      ∨\L                    All 1's following first 1.
0 0 1 1 1 1 1
      ∧\~L                   All 0's following first 0.
1 1 0 0 0 0 0
      <\L                    Removes all 1's following
0 0 1 0 0 0 0                the first.
      X
 2 3 5 7
 3 1 7 8
 4 7 9 2
      ∧/X=⌈\X                1's indicate rows of X which
1 0 0                          are in ascending order.
      +\⍳5                   Triangular numbers.
1 3 6 10 15
      ×\⍳5                   Factorials.
1 2 6 24 120
```

For any associative function α the following definition of *R*←α*X* is formally equivalent to the definition *R*[*I*]=α/*I*↑*X*:

```
      R[1]=X[1]
      R[I]=R[I-1]αX[I] for I∈1↓⍳ρX
```

This definition requires only ¯1+ρ*X* applications of α (as compared to .5×(ρ*X*)×¯1+ρ*X*), and is the one actually used for associative functions. Because of the finite precision used

in machine arithmetic the results of the two definitions may differ, and differ significantly, if the elements of *X* differ by many orders of magnitude. For example, compare the last element of the scan with the corresponding reductions in the following case:

```
      X←1E6 ¯1E6 1E¯16
      +\X
1E6 0 1E¯16
      +/X
0
      +/⌽X
1E¯16
```

Therefore the scan (as well as reduction) should be used with care in work requiring high precision.

EXECUTE

Any character vector can be regarded as a representation of an APL statement (which may or may not be well-formed). The monadic function denoted by ⍎ (⊥ and ∘ overstruck) takes as its argument a character vector or scalar and evaluates or executes the APL statement it represents. When applied to a character array that might be construed as a system command or the opening of function definition, an error will necessarily result when evaluation is attempted, because neither of these is a well-formed APL statement.

There are several major uses of the execute function:

1. In those instances where it is desired to use the name of an APL object as an argument of a defined function, rather than its value, the name can be enclosed in quotes, and the argument later evaluated within the function by means of the execute function. A common example of this is in the use of a general integration function whose arguments might be the vector of grid points and the name of the function to be integrated. For example:

```
      ∇ Z←L INT X;Y
[1]   Z←(1↓X-¯1⌽X)+.×0.5×1↓Y+¯1⌽Y←⍎L,' X'
      ∇
      'Q' INT .1×⍳5
0.0162
```

```
      ∇ Z←Q X
[1]   Z←X*3
      ∇
```

2. When applied to a vector of characters representing numerical constants, the execute function will convert them to numerical values. This is particularly useful in this system, in which access to data generated by alien systems is provided through the shared-variable facility (see Part 5), and large quantities of such data may need to be converted to numerical APL arrays.

3. Where it is necessary to treat collections of data that are related but cannot be combined into a single array, the execute function allows families of names to be used for related variables. The proper variable for each case can be generated and used under program control, either by selecting one of a set of names from a character matrix, by computing a numerical suffix to a generic name, or by other means.

4. The construction ⍎⍞ is nearly equivalent to the use of ⍞ for requesting input from the keyboard during function execution, and has certain advantages: it allows complete control over the output prior to the requested input, and permits the input to be examined by the function prior to attempted execution.

5. Conditional expressions can be constructed in which execution is applied only to the expression selected by the condition, avoiding possible error generation or unnecessary computation. For example, a recursive definition of the factorial function can be written as a single conditional statement:

```
      ∇Z←FACT N
[1]   ⍎3 ¯12[1+N≠0]↑'Z←1  Z←N×FACT N-1' ∇
```

The execute function may appear anywhere in a statement, but it will successfully evaluate only valid (complete) expressions, and its result must be at least syntactically acceptable to its context. Thus, execute applied to a vector that is empty, contains only spaces, or starts with → (branch symbol) or ⍝ (comment symbol) produces no explicit result and therefore can be used only on the extreme left. For example:

```
      ⍎''
      Z←⍎''
VALUE ERROR
      Z←⍎''
      ^
```

The domain of ⍎ is any character array of rank less than two, and *RANK* and *DOMAIN* errors are reported in the usual way:

```
      C←'3 4'
      +/⍎C
7
      ⍎1 3⍴C
RANK ERROR
      ⍎ 1 3 ⍴C
      ^
```

```
      ⍎3 4
DOMAIN ERROR
      ⍎ 3 4
      ^
```


An error can also occur in the attempted execution of the APL expression represented by the argument of ⍎; such an indirect error is reported by the error type prefaced by the symbol ⍎ and followed by the character string and the caret marking the point of difficulty. For example:

```
      ⍎'4÷0'
⍎ DOMAIN ERROR
      4÷0
       ^
      ⍎')WSID'
⍎ VALUE ERROR
      )WSID
       ^
```

FORMAT

The symbol ⍕ (⊤ and ∘ overstruck) denotes two format functions which convert numerical arrays to character arrays. There are several significant uses of these functions in addition to the obvious one for composing tabular output. For example, the use of format is complementary to the use of execute in treating bulk input and output (via the shared variable facility), and in the management of combined alphabetic and numeric data.

The monadic format function produces a character array identical to the printing normally produced by its argument, but makes this result explicitly available. For example:

```
      M←2=?4 4⍴2
      R←⍕M
      M                    R                    R[;2×⍳4]
 0 1 0 1              0 1 0 1              0101
 0 0 1 1              0 0 1 1              0011
 1 0 1 1              1 0 1 1              1011
 0 0 1 1              0 0 1 1              0011
      ⍴M                   ⍴R
4 4                  4 8

      ⍴⍕2 5
3
      ∧/,R=⍕R
1
      ⍕'ABCD'
ABCD
```

The format function applied to a character array yields the array unchanged, as illustrated by the last two examples above. For a numerical array, the shape of the result is the same as the shape of the argument except for the required expansion along the last coordinate, each number going, in general, to several characters. The format of a scalar number is always a vector.

The dyadic format function accepts only numerical arrays as its right argument, and uses variations in the left argument to provide progressively more detailed control over the result. Thus, for $F\,\overline{\triangledown}\,A$, the argument F may be a single number, a pair of numbers, or a vector of length $2\times\,\overline{}\,1\uparrow 1,\rho A$.

In general, a pair of numbers is used to control the result: the first determines the total width of a number field, and the second sets the precision. For decimal form the precision is specified as the number of digits to the right of the decimal point, and for scaled form it is specified as the number of digits in the multiplier. The form to be used is determined by the sign of the precision indicator, negative numbers indicating scaled form. Thus:

```
      ρ□←A
    12.34    ¯34.567
     0        12
    ¯0.26    ¯123.45
3 2
      R←9 2▼A
      S←9 ¯2▼A
      ρ□←R
    12.34    ¯34.57
     0.00     12.00
    ¯0.26    ¯123.45
3 18
      ρ□←S
  1.2E01   ¯3.5E01
  0.0E00    1.2E01
 ¯2.6E¯01  ¯1.2E02
3 18
```

```
      ρ□←12 3▼A
    12.340     ¯34.567
     0.000      12.000
    ¯0.260     ¯123.450
3 24

      ρ□←6 0▼A
    12   ¯35
     0    12
     0   ¯123
3 12

      ρ□←7 ¯1▼A
   1E01   ¯3E01
   0E00    1E01
 ¯3E¯01  ¯1E02
3 14
```


If the width indicator of the control pair is zero, a field width is chosen such that at least one space will be left between adjacent numbers. If only a single control number is used, it is treated like a number pair with a width indicator of zero:

```
      ρ□←2▼A
    12.34    ¯34.57
    ¯0.00     12.00
    ¯0.26    ¯123.45
3 16

      ρ□←0 2▼A
    12.34    ¯34.57
    ¯0.00     12.00
    ¯0.26    ¯123.45
3 16
```

```
      ρ□←¯2▼A
   1.2E01   ¯3.5E01
   0.0E00   ¯1.2E01
  ¯2.6E¯01  ¯1.2E02
3 18

      ρ□←0 ¯2▼A
   1.2E01   ¯3.5E01
   0.0E00   ¯1.2E01
  ¯2.6E¯01  ¯1.2E02
3 18
```

Each column of an array can be individually composed by a left argument that has a control pair for each:

```
      ρ□←0 2 0 2▼A
    12.34    ¯34.57
    ¯0.00     12.00
    ¯0.26    ¯123.45
3 15

      ρ□←6 2 12 ¯3▼A
  12.34    ¯3.46E01
   0.00     1.20E01
  ¯0.26    ¯1.23E02
3 18
```

```
      ρ□←8 3 0 2▼A
   12.340    ¯34.57
    0.000     12.00
   ¯0.260    ¯123.45
3 16

      ρ□←8 0 0 ¯2▼A
    12  ¯3.5E01
     0   1.2E01
     0  ¯1.2E02
3 17
```

```
      6 2 8 3 3 0 4 0 5 0 12 4▼,A
 12.34  ¯34.567  0  12  0  ¯123.4500
```

The format function applied to an array of rank greater than two applies to each of the planes defined by the last two coordinates. For example:

```
      L←2=?2 2 2 5ρ2
      L
1 1 0 0 1
1 1 1 0 1

1 0 0 1 0
0 0 0 0 0
```

```
      4 1▼L
1.0 1.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0

1.0 0.0 0.0 1.0 0.0
0.0 0.0 0.0 0.0 0.0
```

Tabular displays incorporating row and column headings, or other information between columns or rows, are easily

configured using the format function together with extended catenation. For example:

```
      ROWHEADS←4 3⍴'JANAPRJULOCT'
      YEARS←71+⍳5
      TABLE←.001×¯4E5+?4 5⍴8E5
      (' ',[1]ROWHEADS),(2⍕9 0⍕YEARS),[1]9 2⍕TABLE
        72      73      74      75      76
JAN  318.13 ¯351.55   3.62 ¯144.77  ¯4.82
APR ¯327.41 ¯341.00 ¯92.69  331.05 ¯28.44
JUL ¯359.93  216.16 ¯299.71 150.77  103.64
OCT  180.33  310.86 ¯154.94  10.62  276.79
```

It is no longer necessary to use "heterogeneous output" in order to conveniently combine literal statements with numerical results. For example:

```
      X←17.34
      'THE VALUE OF X IS ';X
THE VALUE OF X IS 17.34
      'THE VALUE OF X IS ',⍕X
THE VALUE OF X IS 17.34
```

```
      M←'THE VALUE OF X IS '
      X←25.4
      M;X
THE VALUE OF X IS 25.4
      (⍕M),(⍕X)
THE VALUE OF X IS 25.4
```

There are obvious restrictions on the left argument of format, since the width of a field must be large enough to hold the requested form, and if the specified width is inadequate the result will be a *DOMAIN* error. However, the width need not provide open spaces between adjacent numbers. For example, boolean arrays can be tightly packed:

```
      1 0⍕2=?4 4⍴2
1001
0000
1101
0111
```

The following formal characteristics of the format function need not concern the general user, but may be of interest in certain applications:

The least width required to represent a column of numbers C for an indicated precision P is determined as $W←(\vee/C<0)+(\sim P\in 0\ ¯1)+(|P)+(4,\lceil/0,1+\lfloor 10\circledast|C+C=0)[1+P\geq 0]$. If the width indicator is zero, the width used is $1+W$.

The expressions $(M⍕A),N⍕B$ and $(M,N)⍕A,B$ are equivalent if M and N are full control vectors, that is, if $((\rho M)=2\times ¯1\uparrow\rho A)\wedge(\rho N)=2\times ¯1\uparrow\rho B$. If $2=\rho M$, then $(M⍕A),M⍕B$ and $M⍕A,B$ are equivalent.

PART 4
SYSTEM FUNCTIONS AND SYSTEM VARIABLES

INTRODUCTION

Although the primitive functions of APL deal only with abstract objects (arrays of numbers and characters), it is often desirable to bring the power of the language to bear on the management of the concrete resources or the environment of the system in which APL operates. This can be done within the language by identifying certain variables as elements of the interface between APL and its host system, and using these variables for communication between them. While still abstract objects to APL, the values of such system variables may have any required concrete significance to the host sytem.

In principle all necessary interaction between APL and its environment could be managed by use of a complete set of system variables, but there are situations where it is more convenient, or otherwise more desirable, to use functions based on the use of system variables which may not themselves be made explicitly available. Such functions are called, by analogy, system functions.

System variables and system functions are denoted by distinguished names that begin with a quad. The use of such names is reserved for the system and cannot be applied to user-defined objects. They cannot be copied, grouped, or erased; those that denote system variables can appear in function headers, but only to be localized. Within APL statements, distinguished names are subject to all the normal rules of syntax.

SYSTEM FUNCTIONS

Like the primitive abstract functions of APL, the system functions are available throughout the system, and can be used in defined functions. 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.

Altogether, 13 system functions are provided. Six of these are concerned with the management of the shared-variable facility and are described in Part 5. The other seven are given in Table 1, and are described here.

FUNCTION	REQUIREMENTS RANK	REQUIREMENTS LENGTH	REQUIREMENTS DOMAIN	EFFECT ON ENVIRONMENT	EXPLICIT RESULT
$\square CR\ A$	$1 \geq \rho\rho A$		Array of characters.	None.	Canonical representation of object named by A. The result for anything other than an unlocked defined function is of dimension 0 0.
$\square FX\ M$	$2 = \rho\rho M$		Matrix of characters.	Fix (establish) definition of the function represented by M, 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.
$\square EX\ A$	$2 \geq \rho\rho A$		Array of characters.	Expunge (erase) objects named by rows of A, except groups, labels, or halted functions.	A boolean vector whose Ith element is 1 if the Ith name is now free.
$\square NL\ N$	$1 \geq \rho\rho N$	$1 \geq \rho, S$	$\wedge/N \in 1\ 2\ 3$	None.	A matrix of rows (in accidental order) representing names of designated kinds in the dynamic environment: 1,2,3 for labels, variables, functions.
$A\ \square NL\ N$	$1 \geq \rho\rho N$ $1 \geq \rho\rho A$		$\wedge/N \in 1\ 2\ 3$ Elements of A must be alphabetic.	None.	As for the monadic form, except that only names beginning with letters in A will be included.
$\square NC\ A$	$2 \geq \rho\rho M$		Array of characters.	None.	A vector giving the usage of the name in each row of A: 0,1,2,3,4 for name is available, a label, a variable, a function, other.
$\square DL\ S$	$1 \geq \rho\rho S$		Numeric value.	None, but requires S seconds to complete.	Scalar value of actual delay.

Table 1: System Functions

Canonical representation. The character array printed in displaying the definition of a function F is clearly an unambiguous representation of the function F. The representation remains unambiguous if the ∇ symbols and the line numbers and their brackets are removed, and the rows representing lines containing labels are shifted to the right to remove the exdenting. If the rows are then padded with spaces on the right, where necessary to make them all of equal length, the resulting matrix is called the canonical representation of F. The canonical representation of a defined function is obtained as a result of applying the system function $\square CR$ to the character array representing the name of the function. For example:

```
      ∇BIN[□]∇
    ∇ Z←BIN X
[1]   Z←1
[2]   L1:Z←(0,Z)+Z,0
[3]      →(X≥ρZ)/L1
      ∇
      M←□CR 'BIN'
      M
Z←BIN X
Z←1
L1:Z←(0,Z)+Z,0
→(X≥ρZ)/L1
      ρM
4 14
      BIN 4
1 4 6 4 1
```

The function $\square CR$ applied to any argument which does not represent the name of an unlocked defined function yields a matrix of dimension 0 by 0. Possible error reports for $\square CR$ are *RANK* error if the argument is not a vector or a scalar, or *DOMAIN* error if the argument is not a character array.

Function establishment. The definition of a function can be established or fixed by applying the system function $\square FX$ to its canonical representation. To continue the preceding example:

```
      M[3;11]←'-'
      □FX M
BIN
    ∇BIN[□]∇
  ∇ Z←BIN X
[1]   Z←1
[2]   L1:Z←(0,Z)-Z,0
[3]      →(X≥ρZ)/L1
  ∇
      BIN 4
1 -4 6 -4 1
```

As shown by the foregoing example, the function ⎕*FX* produces as an explicit result the array of characters which represents the name of the function being fixed, while replacing any existing definition of a function with the same name. The argument to ⎕*FX* is, of course, unaffected. The name provided by the explicit result can be conveniently used in a variety of ways. For example:

```
      ⍷(⎕FX M),' 4'
1 ¯4 6 ¯4 1
```

The name of any function established by the function ⎕*FX* obeys the normal rules of localization. Thus if a function *ABC* is established within a function *G* in which the name *ABC* is local, the definition of *ABC* disappears upon termination of execution of the function *G*. Function definition mode continues to apply to global names only.

An expression of the form ⎕*FX M* will establish a function if both the following conditions are met:

1. *M* is a valid representation of a function. Any matrix which differs from a canonical matrix only in the addition of non-significant spaces (other than rows consisting of spaces only) is a valid representation.

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

If the expression fails to establish a function then no change occurs in the workspace and the expression returns a scalar index of the row in the matrix argument where the fault was found. If the argument of ⎕*FX* is not a matrix a *RANK* error will be reported, and if it is not a character array a *DOMAIN* error will result.

Dynamic erasure. Certain name conflicts can be avoided by using the expunge function ⎕*EX* to dynamically eliminate an existing use of a name. Thus ⎕*EX* '*PQR*' will erase the object *PQR* unless it is a label, a group, or a pendent or suspended function. The function returns an explicit result of 1 if the name is now unencumbered, and a result of 0 if it is not, or if the argument does not represent a well-formed name. The expunge function applies to a matrix of names and then produces a logical vector result. ⎕*EX* will report a *RANK* error if its argument is of higher rank than a matrix, or a *DOMAIN* error if the argument is not a character array.

The expunge function is like the erase command except that it applies to the active referent of a name (which may be local), and cannot expunge certain names.

Name list. The dyadic function ⎕*NL* yields a character matrix, each row of which represents the name of an object in the dynamic environment. The right argument is an integer scalar or vector which determines the class of names produced as follows: 1, 2, and 3 respectively invoke the names of labels, variables, and functions. The left argument is a scalar or vector of alphabetic characters which restricts the names produced to those with an initial letter occurring in the argument. The ordering of the rows is accidental.

The monadic function ⎕*NL* behaves analogously with no restriction on initial letters. For example, ⎕*NL* 2 produces a matrix of all variable names, and either of ⎕*NL* 2 3 or ⎕*NL* 3 2 produces a matrix of all variable and function names.

The uses of ⎕*NL* include the following:

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

In conjunction with ⎕*CR*, functions can be written to automatically display 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. The monadic function ⎕*NC* accepts a matrix of characters and returns a numerical indication of the class of the name represented by each row of the argument. A single name may also be presented as a vector or scalar.

The result of ⎕*NL* is a suitable argument for ⎕*NC*, but other character arrays may also be used, in which case the possible results are integers ranging from 0 to 4. The significance of 1, 2, and 3 are as for ⎕*NL*; a result of 0 signifies that the corresponding name is available for any use; a result of 4 signifies that the argument is not available for use as a name. The latter case may arise because the name is in use for denoting a group, or because the argument is a distinguished name or not a valid name at all.

Delay. The delay function, denoted by ⎕*DL*, evokes a pause in the execution of the statement in which it appears. The argument of the function determines the duration of the pause, in seconds, but the accuracy is limited by possible contending demands on the system at the moment of release. Moreover, the delay can be aborted by a single attention signal, which also causes an exit from the program using ⎕*DL*. The explicit result of the delay function is a scalar value equal to the actual delay. If the argument of ⎕*DL* is not a scalar or vector with a single numerical value, a *RANK* or *DOMAIN* error will be reported.

Generally speaking, the delay function uses only a negligible amount of computer time (as opposed to connect time). It can therefore be used freely in situations where repeated tests may be required at intervals to determine whether an expected event has taken place. This is useful in work with shared variables (as in the example on page 28), as well as in certain kinds of interactions between users and programs.

SYSTEM VARIABLES

System variables are instances of shared variables, which are treated in Part 5. The characteristics of shared variables that are most significant here are these:

1. If a variable is shared between two processors, the value of the variable when used by one of them may well be different from what that processor last specified, and

2. each processor is free to use or not use a value specified by the other, according to its own internal workings.

System variables are shared between a workspace and the APL processor. Sharing takes place automatically each time a workspace is activated and, when a system variable is localized in a function, each time the function is used.

The system variables are listed in Table 2, which gives their significance and use. Also listed are the workspace functions and I-beam functions they are intended to replace. These earlier, ad hoc, facilities are still available, but are expected to be supplanted by the use of system variables. The old definitions of the workspace functions will no longer work. New definitions may be copied from 1 *WSFNS*, or defined for each according to the following example:

```
      ∇ Z←ORIGIN N
[1]   Z←⎕IO
[2]   ⎕IO←N ∇
```


NAME	PURPOSE	VALUE IN CLEAR WS	MEANINGFUL RANGE	ALTERNATE FACILITY [1]
⎕*CT*	Comparison tolerance (relative): used in ⌈ ⌊ < ≤ = ≥ > ≠.	$1E^{-}13$	0-1	*SETFUZZ*
⎕*IO*	Index origin: used in indexing and in *?* and ⍳.	1	0 1	*ORIGIN*
⎕*LX*	Latent expression executed on activation of workspace.	''	Characters	None
⎕*PP*	Printing precision: affects numeric output and monadic ⍕.	10	⍳16	*DIGITS*
⎕*PW*	Printing width: affects all but bare output.	120	29+⍳361	*WIDTH*
⎕*RL*	Random link: used in *?*.	7*5	$⍳^{-}1+2*31$	*SETLINK*
⎕*AI*	Account information: identification, computer time, connect time, keying time (all times in milliseconds and cumulative during session).			I29 21 24 19
⎕*AV*	Atomic vector			None
⎕*LC*	Line counter: line numbers of functions in execution, innermost first.	⍳0		I27 26
⎕*TS*	Time stamp: year, month, day, hour, minute, second, millisecond.			I25 20
⎕*TT*	Terminal type: 0 for 1050; 1 for Selectric; 2 for PTTC/BCD.			I28
⎕*UL*	User load.			I23
⎕*WA*	Working area available.			I22

NOTE 1: The old definitions of the workspace functions will no longer work. See text for corrective action.

Table 2: System Variables

Two classes of system variables can be discerned:

Comparison tolerance, index origin, latent expression, printing precision, printing width, and random link: In these cases the value specified by the user (or available in a clear workspace) is used by the APL processor during the execution of operations to which they relate. If this value is inappropriate, or if no value has been specified after localization, an *IMPLICIT* error will be evoked at the time of execution. A non-scalar value is treated as inappropriate.

Account information, atomic vector, line counter, time stamp, terminal type, user load, and work area: In these cases localization or setting by the user are immaterial. The APL processor will always reset the variable before it can be used again.

The APL statement represented by the latent expression is automatically executed whenever the workspace is activated. Formally, ⎕*LX* is used as an argument to the execute function (⍎⎕*LX*), and any error message will be appropriate to the use of that function. Common uses of the latent expression include the form ⎕*LX*←'*G*' used to invoke an arbitrary function *G*, the form ⎕*LX*←'''*FOR NEW FEATURES IN THIS WS ENTER: NEW*''' used to print a message upon activation of the workspace, and the form ⎕*LX*←'→⎕*LC*' used to automatically restart a suspended function. The variable ⎕*LX* may also be localized within a function and respecified therein to furnish a different latent expression when the function is suspended:

```
      ⎕LX←'F'
      ∇ F;⎕LX
[1]   ⎕LX←'→⎕LC,⍴⎕←''WE CONTINUE FROM WHERE WE LEFT OFF'''
[2]   'WE NOW BEGIN LESSON 2'
[3]   DRILLFUNCTION ∇
      )SAVE ABC
```

On the first activation of workspace *ABC*, the function *F* would be automatically invoked; if it were later saved with *F* halted, subsequent activation of the workspace would automatically continue execution from the point of interruption.

The atomic vector ⎕*AV* is a 256-element vector of all possible characters. If *V* is any 8-element logical vector, then (in 0-origin) ⎕*AV*[2⊥*V*] is the character whose internal representation is *V*. Certain elements of ⎕*AV* are terminal control characters, e.g., in 0-origin ⎕*AV*[156 158 159] is the carriage return, backspace, and linefeed. Many elements of ⎕*AV* neither print nor exercise control. The indices of any known characters can be determined as in this example: ⎕*AV*⍳'*ABCabc*' yields 86 87 88 113 114 115.

PART 5
SHARED VARIABLES

INTRODUCTION

Two otherwise independent concurrently operating processors can communicate, and thereby be enabled to cooperate, if they share one or more variables. Such shared variables constitute an interface between the processors, through which information can be passed to be used by each for its own purposes. In particular, variables can be shared between two active APL workspaces, or between an APL workspace and some other processor that is part of the overall APL system, to achieve a variety of effects including the control and utilization of devices such as printers, card readers, magnetic tape units, and magnetic disk storage units.

In use in an APL workspace, a shared variable may be either global or local, and is syntactically indistinguishable from ordinary variables. It may appear to the left of an assignment, in which case its value is said to be set, or elsewhere in a statement, where its value is said to be used. Either form of reference is an access.

At any instant a shared variable has only one value, that last assigned to it by one of its owners. Characteristically, however, a processor using a shared variable will find its value different from what it might have set earlier. A familiar example of this in APL is the quote quad when it is used successively for output from a function and input to it from the keyboard; ⍞ is, in fact, a variable shared between the function and the user at the terminal.

A given processor can simultaneously share variables with any number of other processors. However, each sharing is bilateral; that is, each shared variable has only two owners. This restriction does not represent a loss of generality in the systems that can be constructed, and commonly useful arrangements are easily designed. For example, a shared file can be made directly accessible to a single control processor which communicates bilaterally with (or is integral with) the file processor itself. In turn, the central processor shares variables bilaterally with each of the using processors, controlling their individual access to the data, as required.

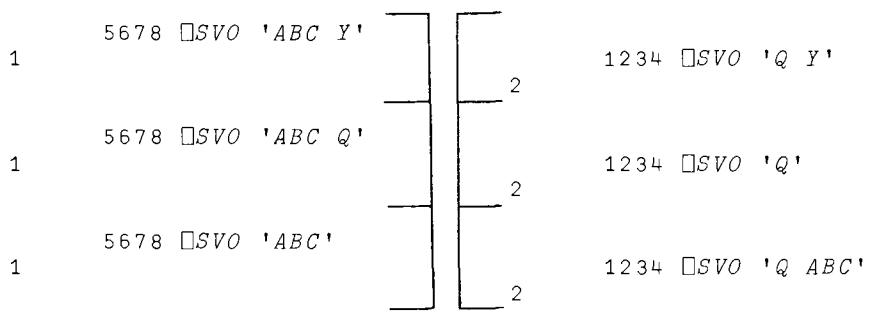
It was noted in Part 4 that system variables are instances of shared variables in which the sharing is automatic. It was not pointed out, however, that access sequence disciplines are also imposed on certain of these variables, although one effect of this was noted; namely, variables like the time stamp accept any value specified,

but continue to provide the proper information when used. The discipline that accomplishes this effect is an inhibition against two successive accesses to the variable unless the sharing processor (the system) has set it in the interim.

When ordinary, "undistinguished", variables are to be shared, explicit actions are necessary to effect the sharing and establish a desired access discipline. Six system functions are provided for these purposes; three for the actual management and three to provide related information. The functions are summarized in Table 3.

OFFERS

A single offer to share is of the form *P* □*SVO N*, where *P* is the identification of another processor and *N* is a character vector representing a pair of names. The first of this pair is the name of the variable to be shared, and the second is a surrogate name which is offered to match a name offered by the other processor. The name of the variable may be its own surrogate, in which case only the one name need be used, rather than two. For example, the three sets of actions shown below all have the same effect, which is to share one variable between users 1234 and 5678, the variable being known to the former as *ABC*, and to the latter as *Q*. Note that the processor identification of a user is his account (sign-on) number on the APL system.



The surrogate names have no effect other than to control the matching, making it possible for one processor to operate with no direct knowledge of, or concern with, the variable name used by the other. The same surrogate can be used in a succession of offers to the same processor, in which case they are matched in sequence by appropriate counter-offers. The same surrogate may also be used for offers to any number of other processors at the same time. However, since a variable may be offered to (or shared with) only one other processor at a time, each coincident use of a particular surrogate name must be associated with a different variable name.

FUNCTION	REQUIREMENTS [1] RANK	REQUIREMENTS [1] LENGTH	REQUIREMENTS [1] DOMAIN	EFFECT ON ENVIRONMENT	EXPLICIT RESULT
P □*SVO N*	2≥ρρ*N* 1≥ρρ*P*	(×/ρ*P*)∈1,¯1↓ρ*N*	*P*∈¯1+⍳2*31 [2]	Tenders offer to processor *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. The offer is general (to anyone) if 0=*P*.	Degree of coupling now in effect for the name pair. Dimension: ×/¯1↓ρ*N*.
□*SVO N*	2≥ρρ*N*	None.	[2]	None.	Degree of coupling now in effect for the name pair. Dimension: ×/¯1↓ρ*N*.
C □*SVC N*	2≥ρρ*N* 2≥ρρ*C*	(1≥ρρ*C*)∧1=×/ρ*C* or (ρ*C*)=(¯1↓ρ*N*),4	∧/*C*∈0 1 [2]	Sets access control.	New setting of access control. Dimension: (¯1↓ρ*N*),4
□*SVC N*	2≥ρρ*N*	None.	[2]	None.	Existing access control.
□*SVR N*	2≥ρρ*N*	None.	[2]	Retracts offers (ends sharing).	Degree of coupling before this retraction. Dimension: ×/¯1↓ρ*N*.
□*SVQ P*	1≥ρρ*P*	1≥ρ,*P*	*P*∈¯1+⍳2*31	None.	If 0=ρ*P*: Vector of identifications of processors making offers to this user. If 1=×/ρ*P*: Matrix of names offered by processor *P* but not yet shared.

NOTES:
1. If a requirement is not met the function is not executed and a corresponding error report is printed.
2. 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.

TABLE 3: Functions for the Management of Sharing

The explicit result of the expression *P* ⎕*SVO N* is the degree of coupling of the name or name pair in *N*: zero if no offer has been made, one if an offer has been made but not matched, two if sharing is completed. An offer to any processor (other than the offering processor itself) increases the coupling of the name offered if the name has zero coupling and is not the name of a label, function, or group. An offer never decreases the coupling.

The monadic function ⎕*SVO* does not affect the coupling of the name represented by its argument, but does report the degree of coupling as its explicit result. If the degree of coupling is one or two, a repeated offer has no further implicit result and either monadic or dyadic ⎕*SVO* may be used for inquiry. Advantage is taken of this in the following example of a defined function for offering a name (to be entered on request) to a processor *P* for a period of *T* seconds:

```
     ∇ Z←P OFFER T;I;Q
[1]   ⍞←'NAME: '
[2]   →(' '∧.=Q←⍞)/Z←I←0
[3]  L1:→(2=Z←P ⎕SVO Q)/L2
[4]   →(T≥I←I+1+0×⎕DL 1)/L1
[5]   'NO DEAL'
[6]   →0
[7]  L2:'ACCEPTED' ∇
```

If the arguments of ⎕*SVO* fail to meet any of the basic requirements listed in Table 3, the appropriate error report is evoked and the function is not executed. If a user attempts to share more variables than the quota allotted to him by those responsible for the general management of the system the error report will be *INTERFACE QUOTA EXHAUSTED*, and if, for any reason, the shared variable facility itself is not available the report will be *NO SHARES*. An offer to a processor will be tendered, whether or not the processor happens to be available.

The value of a shared variable when sharing is first completed is determined thus: if both owners had assigned values beforehand, the value is that assigned by the first to have offered; if only one owner had, that value obtains; if neither had, the variable has no value. Names used in sharing are subject to the usual rules of localization.

A set of offers can be made by using a vector left argument (or a scalar or one-element vector which is automatically extended) and a matrix right argument, each of whose rows represents a name or name pair. The offers are then treated in sequence and the explicit result is the vector of the resulting degrees of coupling. If the quota of shared variables is exhausted in the course of such a multiple offer, none of the offers will be tendered.

An offer made with zero as left argument is a general offer, that is, an offer to any processor. A general offer will be matched only with a counter-offer which is not general, that is, one that explicitly identifies the processor making the general offer. The processor identification associated with a user is his account (sign-on) number. Auxiliary processors such as TSIO are usually identified by numbers between 1 and 1000.

ACCESS CONTROL

Consider the following simple example of sharing the variable *V* between two users 1234 and 5678:

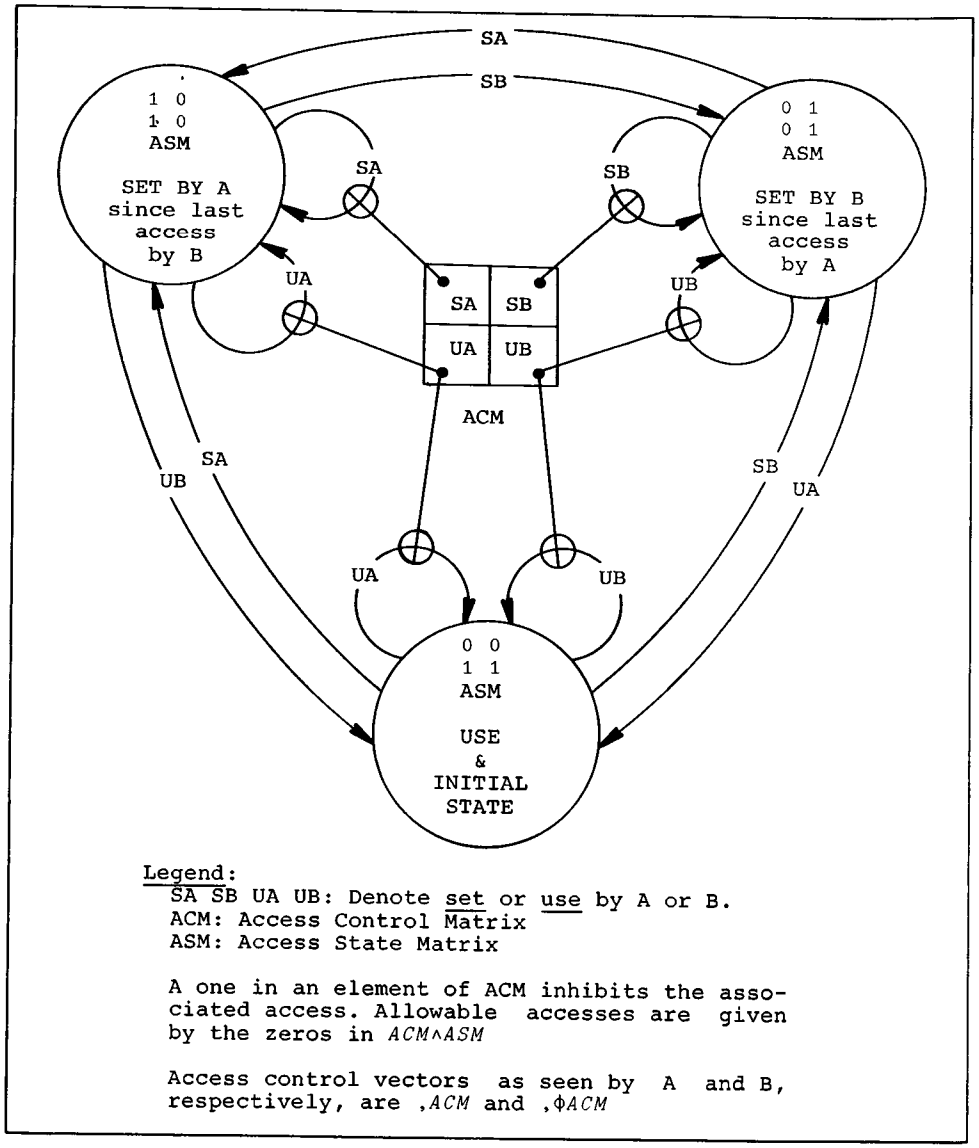
```
      User 1234                          User 5678

      5678 ⎕SVO 'V'
1                                                    1234 ⎕SVO 'V'
                                      2
      V←5                                            V←3×V*2

      V
75
```

The relative sequence of events in the two workspaces, after sharing, is significant; for example, had the use of *V* by 1234 in the foregoing example preceded the setting by 5678, the resulting value would have been 5 rather than 75.

In most practical applications it is important to know that a new value has been assigned between successive uses of a shared variable, or that use has been made of an assigned value before a new one is set. Since, as a practical matter, this cannot be left to chance, an access control mechanism is embodied in the shared variable facility.

The access control operates by inhibiting the setting or use of a shared variable by one owner or the other, depending upon the access state of the variable and the value of an access control matrix which is set jointly by the two owners, using the dyadic form of the system function ⎕*SVC*. If, in the example above, one user (say 5678) had followed his offer to share *V* by the expression `1 1 1 1 ⎕SVC 'V'`, then the desired sequence would have been enforced. That is, the use of *V* by 5678 would be automatically delayed until *V* is set by 1234, and the use by 1234 would be delayed until *V* is set by 5678.

The delay occasioned by the inhibition of any access uses only a negligible amount of computer time. Interruption by a double attention signal during the period of delay aborts the access and unlocks the keyboard.



Legend:
SA SB UA UB: Denote set or use by A or B.
ACM: Access Control Matrix
ASM: Access State Matrix

A one in an element of ACM inhibits the associated access. Allowable accesses are given by the zeros in $ACM \wedge ASM$

Access control vectors as seen by A and B, respectively, are $,ACM$ and $,\phi ACM$

Figure 4: Access Control of a Shared Variable

Figure 4 shows the three access states possible for a shared variable, the possible transitions between states, and the potential inhibitions imposed by the access control matrix, ACM. The first row of ACM is associated with setting of the variable by each owner, and the second with its use. The permissible operations for any state are indicated by the zeros in $ACM \wedge ASM$, where ASM is the representation of the access state shown in the figure. This can be confirmed by using Figure 4 to validate each of the following statements:

If $ACM[1;1]=1$, then two successive sets by A require an intervening access (set or use) by B.

If $ACM[1;2]=1$, then two successive sets by B require an intervening access by A.

If $ACM[2;1]=1$, then two successive uses by A require an intervening set by B.

If $ACM[2;2]=1$, then two successive uses by B require an intervening set by A.

The value of the access state representation is not directly available to a user, but the value of the access control matrix is given by the monadic function $\square SVC$. For a shared variable V the result of the expression $\square SVC$ 'V' executed by user A is the access control vector $,ACM$ (the four-element ravel of ACM). However, if user B executed the same expression he would obtain the result $,\phi ACM$. The reason for the reversal is that sharing is symmetric: neither owner has precedence over the other, and each sees a control vector in which the first one of each pair of control settings applies to his own accesses. This symmetry is evident in Figure 4; if it were redrawn to interchange the roles of A and B the control matrix would be the row-reversal of the matrix shown.

The setting of the access control matrix for a shared variable is determined in a manner which maintains the functional symmetry. An expression of the form $L \square SVC$ 'V' executed by user A assigns the value of the logical left argument L to a four-element vector which, for the purposes of the present discussion, will be called QA. Similar action by user B sets QB. The value of the access control matrix is determined as follows:

$$ACM \leftarrow (2\ 2 \rho QA) \vee \phi 2\ 2 \rho QB$$

Since ones in ACM inhibit the corresponding actions, it is clear from this expression that one user can only increase the degree of control imposed by the other (although he can, by using $\square SVC$ with a left argument of zeros, restore the control to that minimum level at any time).

Access control can be imposed only after a variable is offered, either before or after the degree of coupling reaches two. The initial values of *QA* and *QB* when sharing is first offered are zero.

The access state when a variable is first offered (degree of coupling is one) is always the initial state shown in Figure 4. If the variable is set or used before the offer is accepted, the state changes accordingly. Completion of sharing does not change the access state.

Table 5 lists a number of settings of the access control vector which are of common practical interest. Any one of them can be represented by a simplification of Figure 4 obtained by omitting the control matrix and deleting the lines representing those accesses which are inhibited in the particular case. For example, with maximum constraints all the inner paths would be removed from the figure.

A group of *N* access control matrices can be set at once by applying the function ⎕*SVC* to an *N* by 4 matrix left argument and an *N*-rowed matrix right argument of names. The explicit result is an *N* by 4 matrix giving the current values of the (ravels of) control matrices. When control is being set for a single variable the left argument may be a single 1 or 0 if all inhibits or none are intended.

RETRACTION

Sharing offers can be retracted by the monadic function ⎕*SVR* applied to a name or a matrix of names. The explicit result is the degree (or degrees) of coupling prior to the retraction. The implicit result is to reduce the degree of coupling to zero.

Retraction of sharing is automatic if the connection to the computer is interrupted or if the user signs off or loads a new workspace. Sharing of a variable is also retracted by its erasure or, if it is a local variable, upon completion of the function in which it appeared.

Once a variable has non-zero coupling its access state depends only upon the sequence of accesses that follows, and its access control matrix depends only upon explicit use of dyadic ⎕*SVC*. This means that a variable may be repeatedly retracted and reshared by either owner with no change in these attributes, as long as no overt action is taken to change them and the degree of coupling never becomes less than one. This makes it possible, under suitable settings of the access control vector, to recover gracefully from inadvertent retractions due to communication failures or other mishaps.

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.)

Table 5: Some Useful Settings for the Access Control Vector

The nature of the shared-variable implementation is such that the current value of a variable set by a partner will not be represented within a user's workspace until actually required to be there. This requirement obtains when the variable is to be used, when sharing is terminated, or when a *SAVE* command is issued (since the current value of the variable must be stored). Under any of these conditions it is possible for a *WS FULL* error to be reported. In all cases the prior access state remains in effect and the operation can be retried after corrective action.

INQUIRIES

There are three monadic inquiry functions which produce information concerning the shared variable environment but do not alter it; the functions ⎕*SVO* and ⎕*SVC* already discussed, and the function ⎕*SVQ*. A user who applies the latter function to an empty vector obtains a vector result containing the identification of each user making any sharing offer to him. A user who applies the function ⎕*SVQ* to a non-empty argument obtains a matrix of the names offered to him by the processor identified in the argument. This matrix includes only those names which have not been accepted by counter-offers.

The expression (0≠⎕*SVO* ⎕*NL* 2)/[⎕*IO*] ⎕*NL* 2) can be used to produce a character matrix whose rows represent the names of all shared variables in the environment.

BIBLIOGRAPHY

This bibliography includes only recent items of tutorial interest. For an extensive guide to literature on APL see the bibliography provided by J. C. Rault and G. Demars in the Proceedings of the Fourth International APL User's Conference, July 1972, published by the Board of Education of the City of Atlanta, Georgia.

Iverson, K. E.

Introducing APL to Teachers
IBM Tech. Report No. 320-3014

An Introduction to APL for Scientists and Engineers
IBM Tech. Report No. 320-3019

APL in Exposition
IBM Tech. Report No. 320-3010

Berry, P. C., G. Bartoli, C. Dell'Aquila, and V. Spadavecchia

APL and Insight: The Use of Programs to Represent Concepts in Teaching
IBM Tech. Report No. 320-3020

Falkoff, A. D. and K. E. Iverson

"The Design of APL"
IBM Journal of Research and Development, pp 324-334, Vol. 17, No. 4, July, 1973

Lathwell, R. H.

"System Formulation and APL Shared Variables"
IBM Journal of Research and Development, pp 353-359, Vol. 17, No. 4, July, 1973