APL★PLUS System

FOR THE VAX VMS ENVIRONMENT

Reference Manual

Release 1
August 1987

A PLUS★WARE™ PRODUCT

STSC

APL★PLUS System

FOR THE VAX VMS ENVIRONMENT

Reference Manual

Release 1
August 1987

A PLUS★WARE™ PRODUCT

STSC

10 9 8 7 6 5 4 3 2 1

ISBN 0-926683-32-2

Printed in the United States of America.

Contents

LANGUAGE SUMMARY

Chapter 1
APL Language Summary

This summary provides a general overview of the APL language, data structures, primitive functions and operators, and user-defined functions. If you are not already familiar with the APL language you should first review the book *APL Is Easy!*, which is included with your APL⋆PLUS System. If you are familiar with APL, however, this chapter will give you a good overview of the many features of the APL language.

System commands, distinguished by the leading right parenthesis ( )), are described in Chapter 2 of this manual. System functions and variables, distinguished by the leading quad (⎕) character, are described in Chapter 3.

1-1 APL Data and Arrays

One of the greatest strengths of the APL language is its handling of entire arrays of data as single objects. Here is what you need to know about these arrays and the data in them.

Datatypes

The APL language recognizes two fundamentally different datatypes:

- character data, which can include any of the 256 different symbols in the character set

- numeric data, which is restricted to numbers.

Numbers can be subclassified by the ways they are internally represented. See *Internal Representation and Storage*, later in this section, for details.

Data Constants and Variables

You can use either type of data directly in an APL statement or you can name and store it for later use. Data used without named storage is called a **constant**. Stored data is called a **variable** since you can re-use the name to store different values or even different types of data. You can distinguish character constants from other objects by enclosing them in single quotes ('); for example `'CHARACTER'`. To include a single quote in a character constant, type it twice in a row; for example, `'JOE''S'`. This technique enters one single quote (used here as an apostrophe) so that the stored data contains only the five characters `JOE'S`.

The rules for variable names (also called **identifiers**) follow.

- A variable name can contain any combination of the letters *A* through Z, (either lowercase or uppercase), the digits 0 through 9, ∆ and ⍙. (On some terminals the underscored letters are substituted for the lowercase letters. For example, the lowercase letter "a" is displayed as "A̲". Note that on systems where lowercase letters are substituted for underscored in identifiers, lowercase letters can appear only as data elements in character variables.)

- A digit cannot be used as the first character in a variable name.

- The maximum length of a variable name is usually 77 characters although it may be longer on some systems.

Variables are formed by assigning values with the assignment arrow (←).

```
A←23 15 18 7.3
LAST⍙NAME←'MCMANN'
```

Data Elements and Arrays

An element of character data is a single character (letter, digit, or other symbol); for example, a, *A*, 8, +, ←, ., or ⎕.

An element of numeric data is a single number, regardless of how many characters are needed to represent it; for example, 9, 19, ¯19, ¯19.04, or 2.3E¯11.

Collections of data elements are called **arrays.** In conventional APL, each position or element of an array must contain a single character or number all of one datatype; these are called **simple arrays**. In this

APL*PLUS System implementation, each position of an array (called an item) can contain an array of any rank and datatype. These are called **nested arrays.**

Nested arrays are a powerful extension to APL data storage since they allow mixing data of different types in the same array, as well as non-rectangular data structures.

A calendar is a good example of a nested table. The variable *JULY*87 contains a mixture of data all organized neatly into one format:

```
      JULY 87
 SUN MON TUE     WED THU FRI SAT
                   1   2   3   4
   5   6   7       8   9  10  11
  12  13  14      15  16  17  18
  19  20  21 B-DAY  23  24  25
  26  27  28      29  30  31   *
```

The shape function (ρ) indicates that the variable has 42 items organized into a 6 by 7 table.

```
      ρJULY87
6 7
```

The utility function, *DISPLAY* (available as ⎕*SHOW* on some systems), graphically illustrates what information is stored in each of the items.

```
.→------------------------------------------------.
↓.→--.  .→--.  .→--.  .→--.  .→--.  .→--.  .→--.|
||SUN|  |MON|  |TUE|  |WED|  |THU|  |FRI|  |SAT||
|'---'  '---'  '---'  '---'  '---'  '---'  '---'|
|.θ.    .θ.    .θ.                               
|| |    | |    | |    1      2      3      4     |
|'-'    '-'    '-'                               |
|                                                |
|5      6      7      8      9      10      11   |
|                                                |
|12     13     14     15     16     17      18   |
|                    .→---.                      |
|19     20     21     |BDAY|  23     24      25  |
|                     '----'                     |
|26     27     28     29     30     31      *    |
|                                           -    |
'∊------------------------------------------------'
```

Arrays can be of various shapes and ranks. The shape of an array tells the dimensions of that array (the length of the array along each coordinate). For example, `6 10` is the shape of a 6- by 10-item table; the shape of a 10-item list is `10`; and the shape of a 2-unit 3-dimensional cube is `2 2 2`.

The rank of an array is the number of coordinates it has (how many numbers are needed to specify its dimensions). Arrays can be classified as follows:

Name	Rank	Description
Scalar	0	An array with a single item is called a scalar or element and has no coordinates.
Vector	1	A linear (or one-dimensional) array of elements is called a vector or list and has a single coordinate.

Matrix	2	A two-dimensional array, such as a table of numbers, is called a matrix or table and has two coordinates.
n-dimensional array	*n*	A three-dimensional array, such as a set of matching tables (for example, sales tax tables for each state) has rank 3 and so forth, up through the maximum allowed rank of 63.

A rank 3 array displays as a series of matrices (rank 2 arrays) with one line skipped between them. Similarly, a rank 4 array displays as a series of rank 3 arrays with two lines skipped between them.

Sub-arrays can be extracted by using functions such as compress `(/)`, drop `(↓)`, index `[;]`, take `(↑)`, and pick `(⊃)`.

Empty Arrays

Arrays or items of an array are empty if they have no elements. The shape of an empty array contains one or more zeros (indicating no length along the corresponding coordinate). For example, finding the shape of matrix *M* shows that it is empty because it has no rows:

```
      ⍴M
0 12
```

The shape of a scalar is an empty vector; the rank is 0.

```
      ⍴JULY87[4;4]

      ⍴⍴JULY87[4;4]
0
```

Empty numeric or character arrays can result from executing various functions. Empty vector constants can be included in APL expressions; for example:

```
      A←''⍴A
```

or stored in a variable name just like any other data array; for example:

```
ECV←''
```

Empty character vectors are different from empty numeric or Boolean vectors. Empty vectors can be created using the following expressions:

Character	''
Numeric	⍳0

Empty scalar arrays do not exist because scalars are rank 0 and have no coordinates (and therefore cannot have a coordinate of 0). Scalars always have one data element.

Empty arrays are useful in APL. For example, they can be the starting value of a variable that grows in successive executions of a program or in successive iterations of a loop within a program. In many other programming languages, you must use special tests to detect empty arrays and avoid potential errors. Typical APL statements will work regardless of whether an array is empty.

Strand Notation

Strand notation is a means of entering vectors, either simple or nested. Three kinds of constructs appear in strand notation: constant numeric values such as `12` or `1 2 3`, constant character values such as `'A'` or `'HIERONYMUS BOSCH'`, and expressions such as (`PICKLE×JUICE`). When two or more of these are adjacent, each is interpreted to be an item. Constructs that evaluate to simple scalars remain simple.

Strand notation is an extension of the familiar notation used to enter a constant numeric vector. A position can consist of a number or character, an array of any valid rank or shape, or an expression. An expression may need to be enclosed in parentheses to limit the scope of the functions within it.

Note that stranding occurs only when two or more values are adjacent.

All of the following statements (excluding the initial assignment) return three-item vectors. To better illustrate the structure, the display form (using ⎕*SHOW* or a comparable utility function) is also provided after some of the examples.

```
      A←1 ◊ B←2 ◊ C←3 ◊ D←1 2 3
      A B C
1 2 3
      DISPLAY A B C
.→----.
|1 2 3|
'~----'

      A B D
1 2  1 2 3
      ⍴ A B D
3
      DISPLAY A B D
.→----------.
|     .→----.|
|1 2 |1 2 3||
|     '~----'|
'∊----------'

      A B C×2
2 4 6
      DISPLAY A B C × 2
.→----.
|2 4 6|
'~----'

      A B D + 10
11 12  11 12 13
      DISPLAY A B D + 10
.→---------------.
|       .→-------.|
|11 12 |11 12 13||
|       '~-------'|
'∊---------------'

      A B (D+10)
1 2  11 12 13

      (1 9 4 1)  4 'YOU'
1 9 4 1  4 YOU
```

```
      ρ(1 9 4 1)  4 'YOU'
3
      DISPLAY (1 9 4 1)  4 'YOU'
.→----------------.
|.→------.   .→--.|
||1 9 4 1| 4 |YOU||
|'~------'   '---'|
'∊----------------'

      A 'SNARK' 3.14
 1 SNARK 3.14
      DISPLAY A 'SNARK' 3.14
.→-------------.
|   :→----.    |
|1  |SNARK| 3.14|
|   '-----'    |
'∊-------------'

      (2 3) 4 5
 2 3  4 5
      DISPLAY (2 3) 4 5
.→--------.
|.→--.    |
||2 3| 4 5|
|'~--'    |
'∊--------'

      5 '=' 'V'
 5 =V
      DISPLAY 5 '=' 'V'
.→---.
|5 =V|            (Simple heterogeneous array)
'+---'

      5 '=V'
5    =V
      DISPLAY 5 '=V'
.→-----.
|   .→-.|
|5  |=V||         (Heterogeneous nested array)
|   '--'|
'∊-----'
```

The expression *A B D* [2] is ambiguous. Some APL systems interpret this as

```
      A B (D[2])
```

giving the result

```
1 2 2
```

Others might interpret it as

```
      (A B D) [2]
```

giving

```
2
```

Use parentheses to clear up the ambiguity and ensure that such expressions produce the desired result.

Strand Notation Assignment

Strand notation assignment allows more than one variable to be assigned in one operation. For example:

```
      C D E←R
```

Each variable to the left of the assignment arrow receives the corresponding item of the vector to the right. The right argument is a vector with as many items as there are names to the left of the assignment arrow. A scalar or one-item vector right argument is extended into a vector with one item for each variable name on the left.

Caution: The syntax of strand assignment in current APL⋆PLUS Systems differs from APL2 which requires parenthesis around the list of names to the left of the assignment arrow. For example, `(A B C)←1 2 3`. Future versions of the APL⋆PLUS System may be changed to use this syntax.

Some examples follow.

```
      A B C← 1 2 3
      A ◊ B ◊ C
1
2
3
```

```
      A B C ←4
      A ⋄ B ⋄ C
4
4
4

      A B C←⊂1 2 3
      A ⋄ B ⋄ C
1 2 3
1 2 3
1 2 3

      1⍴A
1

      A B C←'YOU' 'ARE' 'OUR BUSINESS'
      A B C
 YOU ARE OUR BUSINESS
```

Now, let's exchange the values of *A* and *C*:

```
      A C←C A
      A B C
 OUR BUSINESS ARE YOU
```

Internal Representation and Storage

Data occupies memory space in the computer. Even constants are internally represented in memory. Each simple element of an array requires the following storage.

Boolean	1	bit
Character	8	bits
Integer	32	bits
Floating Point	64	bits

In additon, some overhead is associated with each variable. The system function ⎕*SIZE* will report how much memory space a particular variable consumes.

Note that storage of data can vary from one system to another.

The primitive functions and those system functions and variables that require integer data as arguments will ignore tiny differences from true integral values.

`2.9999↑1` would produce the same result as `3↑1` if the system fuzz is .0001, but a *DOMAIN ERROR* if the system fuzz is .000001. (Note: This is not the same as ⎕*CT*, which is used in computing scalar primitive results.)

1-2 Syntax

The word **syntax** means "the correct order or arrangement of the parts to form a valid whole." In English, the whole is a sentence or a phrase. In APL, the whole is a statement or an expression.

APL syntax is the description of how data can be used with functions and operators to produce valid APL statements or expressions. The system reports syntax problems with the message:

SYNTAX ERROR

The system then prints the faulty APL statement and positions a caret (^) beneath the part of the statement that is in error.

There is a good analogy between English grammar and APL syntax.

English	APL
Noun	Data
Verb	Function
Adverb	Operator
Phrase	Expression
Sentence	Statement

Types of Functions

Functions tell the system what to do with data objects. These functions can be

- primitive APL functions (an intrinsic part of the language)

- system functions (particular to each implementation of the language)
- user-defined functions (programs you write).

Each of these function types uses the same set of APL syntactic structures.

The objects of any given function can be:

- to the left of the function name
- to the right of the function name.

These objects are the formal **arguments** of the function. An APL function can have at most two formal arguments.

APL has four kinds of functions:

Function Type	Number of Arguments	Example
niladic	0	⎕*FNAMES* *FOO*
monadic	1	+1 *REPEAT* 10
dyadic	2	2×3 '*LAST*' *OVER* '*FIRST*'
ambivalent	1 or 2	ρ*A* 2ρ*A* *PRINT REPORT* 1260 *PRINT REPORT*

When a function is called with an incorrect number of arguments, the result is an error or possibly incorrect results.

Because APL has many more primitive functions than the keyboard has keys, two techniques are used to represent them:

- The same symbol can represent one monadic function and one dyadic function. The system can always determine which function to perform by the number of arguments. You must be sure which function you want, since using the wrong number of arguments may perform a different function instead of producing an error message.

- Operators can take one or two functions and apply them differently to the data arguments (See Section 1-5 for more information).

Explicit Results

The explicit result of an APL function is the value produced by executing the function. The value is available for further use by another function or for storage. In the example 5 + 4 + 3, the result of the first addition (4+3) is available for immediate re-use in the second addition (5+*result*). This re-usability distinguishes an explicit result from implicit output (see Section 1-6).

While most system functions have an explicit result, some do not. For example, ⎕*FUNTIE* closes a component file and removes its name from the list of those currently in active use but returns no value. Many user-defined functions also have no explicit result.

1-3 Primitive Functions

A function produces a result according to specific rules that act on argument data. A **primitive** function is a function that is built into the APL⋆PLUS system.

Scalar Functions

A scalar function is a function whose data manipulation rule works with a single element at a time. When array arguments are used, the result is the repetition of the scalar operation for corresponding elements in the arrays. For example:

```
      -12 5 20
¯12 ¯5 ¯20
```

because 0-12=¯12, 0-5=¯5, and 0-20=¯20

The primitive scalar functions include all of the simple arithmetic functions and several less familiar function

Scalar dyadic functions take both a left and a right argument. They accept only data arrays of identical shape, with one important exception: either of the argument arrays can have only one element (the other argument can be of any rank). In this case, the single element (or **singleton**) is "extended" and used with each element of the other argument. This extension is illustrated in the following examples for the addition function, but applies to all the functions.

```
      1 2 3 + 10 20 30
11 22 33

      1 2 3 + 10
11 12 13

      1 + 2 3⍴10 20 30 40 50 60
11 21 31
41 51 61

      1 2 3 + 10 20     (3 on left, 2 on right)
LENGTH ERROR
      1 2 3 + 10 20
            ^
```

Non-Scalar Functions

Non-scalar functions, sometimes called mixed functions, do not follow the matching argument rules for scalar functions. Non-scalar functions have various rules for the shape and values of their arguments and results. Many of these functions select or restructure the data without changing the data values by computation, as shown in the following examples.

The reshape function (⍴) creates a new array with the dimensions specified in the left argument using the data in the right argument.

```
      MAT ← 2 3 ⍴1 2 3 4 5 6
      MAT
1 2 3
4 5 6
```

The catenate function (,) joins two arrays specified by the arguments. You can specify the coordinate along which to join multi-dimensional arrays.

```
      1 2 3,9 8 7
1 2 3 9 8 7

      1 2 3,[1]MAT
1 2 3
1 2 3
4 5 6

      1 2,MAT
1 1 2 3
2 4 5 6

      1 2 3,MAT
LENGTH ERROR
      1 2 3,MAT
         ^
```

In the last example, the *LENGTH ERROR* occurred because the last coordinate is the default for catenation. In this case, the function wants to add a new column to the matrix. The vector has three elements, but the matrix has two rows, so the new column cannot be constructed.

The replicate function (/) copies the elements in the right argument the number of times specified in the left agrument.

```
      1 2 3 / 4 5 6
4 5 5 6 6 6

      1  0  1  2  1  2  2 / 'CHOMITE'
COMMITTEE
```

1-4 Operators

Operators produce a new function by modifying the actions of a dyadic function. An operator is essentially a function that takes another function or functions as its argument(s). Following are descriptions and examples of four operators: reduction, inner product, outer product, and each.

Reduction

The reduction operator (/) allows you to perform a function along a dimension of an entire array. The process "reduces" the rank of the data by 1. In reduction, APL conceptually inserts the function to the left of the operator between elements along a dimension of the array.

```
      +/10 20 30
60

      10+20+30
60

      ×/10 20 30
6000

      +/2 3ρι6
6 15

      ,/'MARES' 'EAT' 'OATS'
MARESEATOATS
```

Inner Product

The inner product operator (.) operates on two functions to produce a derived dyadic function that requires the last dimension of the left argument to be equal to the first dimension of the right argument. The right function is applied first and the result is reduced using the left function. For vectors, *A*+.×*B* is equivalent to +/*A*×*B*. For matrices, +.× is used to do matrix multiplication.

```
      MAT1
1 2 3
4 5 6

      MAT2
 7  8
 9 10
11 12
```

```
      MAT1 +.× MAT2
 58  64
139 154
```

(that is, `64=+/1 2 3 × 8 10 12`)

Outer Product

The outer product operator (`∘.`) allows you to generate all possible combinations of the left and right arguments, using the function to the right of the operator. In the following examples, outer product is used to generate a multiplication table.

```
      VEC1 ← ⍳5
      VEC1
1 2 3 4 5

      VEC2 ← 5+VEC1
      VEC2
6 7 8 9 10

      VEC1 ∘.× VEC2
 6  7  8  9 10
12 14 16 18 20
18 21 24 27 30
24 28 32 36 40
30 35 40 45 50
```

Each

The each operator (`¨`) applies a function to the items of its argument or between the items of its arguments to produce the items of its result. The display form of the object is provided for illustration.

```
      1 2 3 ⍴¨ 4 5 6
 4  5 5  6 6 6

      DISPLAY 1 2 3 ⍴¨ 4 5 6
.→----------------.
|.→. .→--. .→----.|
| 4  5 5   6 6 6 |
|'~' '~--' '~----'|
'∊----------------'
```

```
      1 2 3 ,¨4 5 6
 1 4  2 5  3 6

      DISPLAY 1 2 3 ,¨ 4 5 6
.→----------------.
|.→--. .→--. .→--.|
||1 4| |2 5| |3 6||
|'~--' '~--' '~--'|
'∊----------------'

      R←(⊂2 3 5),⊂7 11 13
      R
2 3 5  7 11 13

      DISPLAY R
.→-----------------.
| .→----. .→------.|
| |2 3 5| |7 11 13||
|'~-----' '~------'|
'∊-----------------'

      ⌽¨R
 5 3 2  13 11 7

      ⌽⌽¨R
 13 11 7  5 3 2
```

User-Defined Functions Used with Operators

Powerful array-oriented control structures are provided for user-defined functions called by operators. This new feature can also be used to explore the behavior of an operator, as in the following example.

```
    ∇ Z←L MINUS R
[1]   Z←L-R
[2]   ,'I2,< ->,I2,< =>,I2' ⎕FMT 1 3 ⍴L R Z
    ∇

      5 MINUS 3
 5 - 3 = 2
2

      -/⍳4
¯2
```

```
      MINUS/⍳4
3 - 4 =   ¯1
2 -¯1 =    3
1 - 3 =   ¯2
¯2
```

The next example builds a five-item vector, where each item is a two-item vector. Each two-item vector is used as an argument to the ⎕FREAD function. The result is a five-item vector (FILE), where each item is a component read from the file.

```
      FILE←⎕FREAD¨ ⎕←2 ,¨ ⍳5
2 1  2 2  2 3  2 4  2 5
```

Operator Sequences

Operators have a long left scope and a short right scope. An operator takes as its left argument the function or derived function to the left. Parentheses can be used to limit the scope in the usual way. An operator takes as its right argument only the first function to its right. Parentheses may be necessary to lengthen an operator's right argument. For example,

```
      (1 2)∘.(,¨) (10 20) 30
1 10   1 20    1 30
2 10   2 20    2 30

      DISPLAY (1 2)∘.(,¨) (10 20) 30
.→------------------------.
↓.→------------. .------. |
||.→---. .→---.| |.→---.| |
|||1 10| |1 20|| ||1 30|| |
||'~---' '~---'| |'~---'| |
|'∊------------' '∊------' |
|.→------------. .------. |
||.→---. .→---.| |.→---.| |
|||2 10| |2 20|| ||2 30|| |
||'~---' '~---'| |'~---'| |
|'∊------------' '∊------' |
'∊------------------------'
```

Here the operator is ∘.*f* ,where *f* is the derived function built with the each operator (,¨).

In the following example, the each operator takes as its left argument the derived function plus-reduction (+/).

```
      +/¨ (1 2) (3 4) (5 6)
3 7 11
```

1-5 Data Input and Output

You can move data into and out of the active workspace in several ways:

- You can use the APL input and output functions described in this section in an APL function or in immediate execution mode.
- You can enter constant data from the keyboard in either immediate execution mode or function definition mode.
- You can move data in and out of APL★PLUS component files.
- You can use auxiliary processors to pass data between the active workspace and operating system files.

Evaluated Input

You can use the explicit result of evaluated input immediately within a statement or you can assign the result to a variable. When ⎕ is executed, the prompt ⎕: appears on the screen in columns 1 and 2, with the cursor waiting in column 7 of the next line for input. You can enter any valid APL statement; it will be evaluated and its result will be returned as the result of the input request. The following examples show useful and correct responses for evaluated input.

```
⎕:
      75.3                    Enter a scalar.
⎕:
      2 ¯5 7.56               Enter a vector.
⎕:
      10×⍳20                  Enter a calculation.
```

```
⎕:
      DATAVARIABLE           Enter a variable containing data.
⎕:
      ⎕FREAD 5 7             Enter data stored in a file
⎕:
      'CHARACTER DATA'       Enter a character constant.
⎕:
      →                      End this program execution.
```

If the expression does not return a value or an error occurs, the prompt will reappear:

```
⎕:
      NOT∆PRESENT
VALUE ERROR
      NOT∆PRESENT
                 ^
⎕:
```

If you enter a sequence of statements separated with diamonds (◊) in response to the ⎕: prompt, all statements are executed and the value of the last statement (the rightmost statement) is the explicit result of the ⎕. (See *Compound APL Statements* in Section 1-6).

```
⎕:
      'DFILE' ⎕FTIE 10 ◊ ⎕FREAD 10 2
```

Character Input

APL requests character input with a quote-quad (⍞) and returns it as the explicit result. This type of input is also called quote-quad input. You can assign the result to a variable, or you can use it immediately without assignment (as in →('Y'=1↑⍞)ρ*YES*). The input resulting from ⍞ is always a vector. If you do not enter any characters before pressing ENTER, the vector will be empty.

The ⍞ accepts, but does not execute, any character sequence, even if it looks like an APL statement or a system command. The result vector contains exactly what was typed as input and displayed on the screen, up to but not including the newline character.

When the ⍞ is executed, the only prompt it displays is a cursor. User entry begins wherever the cursor is located. The cursor is located at the left edge of the display unless the request for character input was preceded by a character prompt issued by the same program. When a character prompt appears on the same line, it is included in the explicit result (on some systems, the prompt is replaced by spaces or the contents of ⎕*PR*).

You can interrupt the executing program requesting character input by typing *O* - backspace - *U* - backspace - *T*, and then pressing Enter; or by pressing the key that is defined to have this behavior.

Implicit Output

The calculated explicit result of an APL statement is automatically printed unless it is assigned to a variable.

More precisely, implicit (or default) output occurs from executing every APL statement when:

- the last executed function produced an explicit result
- the last executed function is not assignment (←) or indexed assignment (`[]←`).

All the primitive functions and operators used with them except branch (→) produce explicit results. Many system functions also produce explicit results (see Chapter 3 of this manual).

An APL statement consisting of a single variable name causes implicit output of the data associated with the variable.

Most output from APL programs uses the implicit output syntax, shown in the following examples.

`      I ← ⍳4`	Result is assigned; no output.
`      I×2` `2 4 6 8`	Result is not assigned; output shown.

<code> I</code> <code>1 2 3 4</code>	Result is not assigned; output shown.
<code> B[3]←10×+/I</code>	Result is index assigned; no output.
<code> D ← 4 1 ⍕ I</code>	Result is assigned; no output.
<code> 4 1 ⍕ I</code> <code>1.0 2.0 3.0 4.0</code>	Result is not assigned; output shown.
<code> D←'F4.1'⎕FMT I</code>	Result is assigned; no output.
<code> 'F4.1' ⎕FMT I</code> <code>1.0</code> <code>2.0</code> <code>3.0</code> <code>4.0</code>	Result is not assigned; output shown.

The output is displayed according to the following conventions:

- Character data is not changed–its arrangement is the same, character by character, column by column, as it is in the APL scalar or array. If the data contains characters such as newline or linefeed characters (⎕*TCNL* or ⎕*TCLF*), these will cause their usual effect on the display.

- Each element of numeric data is formatted according to the print precision (⎕*PP*) in effect, with the rows and columns of matrices preserved.

- The rows of data resulting from the preceding step are displayed within the print width (⎕*PW*) in effect. If more than one line is needed to display a row of data, all lines after the first line will be blocked to fit within ⎕*PW* columns.

- For arrays of rank greater than two, the default output inserts blank lines between submatrices (formatted as described above) to indicate the higher coordinates.

Since matrices always have one line of output for each row, a matrix with no rows prints no lines. You can use this behavior to suppress incidental implicit output that a function might otherwise produce as it executes some part of its task; for example:

```
        0 0 ρ ⎕DL 5
```

yields no output.

Requested Output with Trailing Newline

To display data produced by evaluating an expression, using the same display rules as for implicit output, use the following function.

```
        ⎕ ← expression
```

You can use this output syntax to display an intermediate value in an expression or statement. This technique can be useful in debugging; for example:

```
        ⎕FREAD ⎕←TN,CN                       Show file selection.
10 43
        APPLES
        ORANGES
        BANANAS
        PEACHES
```

Requested Output without Trailing Newline

To display the result of an expression without an automatic newline after the data, use the following function.

```
        ⍞ ← expression
```

This technique allows the results of more than one expression to appear on the same line; for example:

```
    DATE ← 1982 ◇ X ← 56.1
    ⍞←DATE ◇ ⍞←' RECORD IS ' ◇ ⍞←X×2 ◇ ' MILES.'
1982 RECORD IS 112.2 MILES.
```

Input on Same Line as Character Prompt

You may want to accept input on the same line as a prompt supplied by your program. Quote-quad (⍞) input does not supply a prompt of its own. Implicit output and quad (⎕) output are both followed by a newline character (⎕*TCNL*), causing the input to be accepted at the left margin on a new line.

To display output and input on the same line, use the following pair of statements.

```
        ⍞ ← output ◇ input ← ⍞
```

Note that *output* or an equal number of blanks is included as part of the result of the character input (*input*). To avoid this side effect, use the statement ⎕*ARBOUT* ⍳0 to clear the output buffer as in the following example.

```
        ⍞←'COMPANY NAME IS '◇⎕ARBOUT⍳0◇CN←⍞
COMPANY NAME IS _          The _ represents the cursor.
```

You then complete the sentence.

```
COMPANY NAME IS STSC, INC.

        CN
STSC, INC.

        ⍴CN
10
```

In the preceding syntax, *output* can be the result of any expression. The righthand statement can be any statement containing a ⍞; for example:

```
.
.
.
[15]    Q←'IS THIS A NEW CUSTOMER?'
[16]    ⍞←Q,' [Y N]   ' ◇ ⎕ARBOUT ⍳0
[17]    →('Y'=1↑⍞)⍴Y3
.
.
.
```

When lines [15] through [17] are executed, the prompt and reply look like:

```
IS THIS A NEW CUSTOMER? [Y N]  Y
```

1-6 Types of APL Statements

APL has only five types of simple statements – far fewer than most programming languages. Three of them (assignment, branch, and implicit output) are executable; two (function header and comment) are non-executable.

The principal part of all APL statements is an **expression**. An expression is a sequence of data constants, data variables, primitive APL functions and operators, system functions, and system variables. The order of this sequence must conform to the syntax rules of each function and operator used, as explained in this chapter and in Chapter 3. The simplest expression is a single data object. An expression can be a part of a larger expression; if it is not, it is called a **statement**.

Executable APL Statements

The three types of executable statements are

- the **assignment** statement, whose leftmost function is assignment; for example, `Y←X⋆2`
- the **branch** statement, that begins with → for example, `→LABEL1`
- the **implicit output** statement, including all executable APL statements that are neither assignment statements nor branch statements; for example, `2+3` .

Non-Executable APL Statements

The two types of non-executable APL statements are

- the **function header** (see Section 1-8)

- the **comment** statement.

The comment statement begins with the lamp symbol (⍝) and continues to the end of the line on which the lamp symbol appears. Use the comment statement in your programs to explain or document them. The ⍝ ensures that the remainder of the line is not executed. Consequently, unmatched quotes, parentheses, and square brackets after a ⍝ cause no problems. Additional ⍝ symbols, ∇, ⍫, or ◊ are also viewed as part of the text of the comment.

In immediate execution mode, comments can be used to annotate your terminal session.

A ⍝ that is enclosed in quotes as part of a character constant does not begin a comment statement.

Compound APL Statements

More than one APL statement can occupy a line. The diamond character (◊) separates two statements on the same line. On some terminals, the diamond is represented by the "hash" symbol (#). A compound APL statement is a line containing two or more simple APL statements. (A function header cannot occur in a compound statement.) A comment statement, if used, must be the last statement on the line. For example:

```
X←⍳10 ◊ X←X×2      This is a compound statement.
```

When multiple statements occur on the same line, they are executed in the order of appearance from left to right. Do not confuse this order with the order of evaluation within each statement, which is from right to left. For more details, see the following subsection and Section 1-8.

A compound statement can be used as a single line in a function and can then be preceded by a label set off by a colon (:), but the label is not considered to be a part of the statement. You cannot use colons within a statement, except as characters within quotes or in comments. For more details, see Section 1-9.

Order of Execution

Often an APL expression contains more than one function. APL expressions always execute the rightmost function first, unless the order is overridden by parentheses. The following example illustrates this order of execution.

```
      7-5-3
5
```

First, 5-3 is performed. Its explicit result (2) is used as the right argument for the remaining subtraction. The entire expression is read as "seven minus the difference between five and three." The left argument, therefore, is simply the nearest single data object named immediately to the left of the function. In our example, the 3 was subtracted from the 5, not from the difference of 7 and 5.

In larger or more complex left arguments, you can use parentheses to enclose an expression to be evaluated before it is used. The parentheses, in effect, make the result of the enclosed expression a single data object that must be evaluated before use; for example:

```
      (7-5)-3
¯1
```

Similarly, an indexed variable (or expression) is evaluated before being used as an argument, thus forcing evaluation of any expression in the indexing brackets ([ ]).

This "right-to-left" order of execution rule applies to all functions: scalar and mixed, primitive, system, or user-defined. The following examples illustrate the order of execution.

```
      2,3⍴10,20-1
2 10 19 10

      (2,3)⍴(10,20)-1
 9 19  9
```

```
19  9 19

      (2,3ρ10,20)-1
1 9 19 9

      2,(3ρ10),20-1
2 10 10 10 19
```

1-7 *Structure of User-Defined Functions*

The APL language supports the creation of user-defined functions, also called programs, routines, or subroutines. A user-defined function consists of a series of one or more APL statements that have been recorded under one name and that can be used by simply typing the name along with any needed input arguments. The series need not be executed in its entirety, but can be selectively executed by testing and branching. This technique also allows sections of a program to repeat or loop.

The elements of a function definition are

- a header, which defines the syntax of the function, identifies the local names of the left and right arguments and explicit result, and defines other local identifiers protected from possible conflict with more global names

- line numbers and labels to represent them, either of which can be used with branching to control the flow of execution (see Section 1-9)
- the body of the function, made up of numbered function lines, consisting either of executable APL statements or of comments for clarity and documentation (see Section 1-7)

- local identifiers, meaningful only within the function or functions called by the function

- a ∇, which signifies the closing or end of the function, or a ⍫, which locks the function definition from further view or changes, even by its owner.

System commands cannot be executed as part of a function definition. Function definition mode prompts cannot be incorporated in a function.

The Function Header

The header line of a function is the first line of the function definition that is entered or displayed. It determines the syntax for calling the function, but is not itself executed. The header always includes the function's name; anything else is optional. The syntax is specified in the header by what surrounds the function's name; for example:

∇*BEGIN*	Niladic function, no explicit result.
∇*RES* ← *SQUARE NUM*	Monadic function, explicit result.
∇*NUM RAISEDTO EXPR*	Dyadic function, no explicit result.

In general, user-defined function header syntax is

result ← *l functionname r;lv1;lv2;lv3*. . .

result	explicit result
l	left argument
functionname	name of the function
r	right argument
lv1, *lv2*, and *lv3*	local variables

The result, function name, argument names, and local variable names must be different.

User-defined functions need not have two arguments; they can be monadic or niladic. They also need not return an explicit result, in which case you would omit "*result* ←" from the function header.

Dyadic (two-argument) user-defined functions are also ambivalent. This means that the left argument is optional. If the function is used without a left argument, the variable *l* is undefined. The following function

MINUS emulates the ambivalent primitive function - .

```
      ∇ R←A MINUS B
[1]     →(0≠⎕NC 'A')⍴DYADIC
[2]     A←0
[3]     DYADIC: R←A-B
      ∇

        1 MINUS 2
-1

        MINUS 3
-3
```

When an incorrect number of arguments is supplied to a user-defined function, the result is often a *SYNTAX ERROR*.

The Explicit Result

If the header begins with an assignment, the function returns an explicit result. This result will be whatever value is stored in the variable to the left of the ← in the header at the time that function execution terminates.

The name used for the explicit result within the body of the function has no initial value when execution begins, even if a variable by the same name exists outside the function in the global environment.

If the function exits before the result variable is assigned, a *VALUE ERROR* will occur if the function result is required in the calling environment.

Arguments of a Defined Function

A name occuring before the function name but after the assignment (if any) is the left argument. A name occuring after the function name is the right argument. They represent the values that will be used in those positions when the function is called. The values used beside the function name when it is executed will be the initial values assigned to these arguments when they are used in the body of the function. The arguments are also considered local variables, and are distinct from objects in the global environment that may have the same names. The local variables cease to exist upon termination of the function execution.

Local Identifiers

You can create other local identifiers by placing those names in the function header. They can appear anywhere after the definition of the function's syntax, and must be separated by semicolons.

All identifiers in the header (except the function name itself) are local, and do not have the same meaning in the global environment that they do within the function. The global objects that are unavailable from within the function are said to be **shadowed**. All identifiers referred to in the body of the function that do not occur in the header (except labels) are global. Assignments made to them survive function execution.

Local identifiers can be used for:

- user-defined local variables (including the arguments and explicit result)
- labels
- user-defined local functions created using ⎕*DEF* or ⎕*FX* within the function
- localized system variables (changes to their values do not survive termination of function execution)
- variables global to sub-functions.

Lines of a Defined Function

Each line of a defined function consists of an APL statement or comment. The lines are numbered automatically by the function editor, and may have labels between the line number and the statement. A label remains with the APL statement or comment it begins, even if the lines are renumbered. Labels are therefore a good way to refer to a particular line of a function when branching (see next section). Labels are variables local to the function in which they are defined and have a value equal to the line number of the line on which they are found.

Comments can start anywhere on the line, but once the ⍝ symbol has appeared, the rest of that line becomes part of the comment. Thus, comments beginning ⍝∇ are possible, and are called public comments (see ⎕*CRLPC* in Chapter 3).

1-8 Control of Execution

The lines in a user-defined function are numbered in ascending order from top to bottom and, in the absence of a branch, will be executed in numeric order. The system variable ⎕*LC* contains the line number of the currently executing line.

The function and line being executed are tracked in the state indicator, and can be examined with)*SI*, or)*SINL*. The state indicator shows the name of the user-defined function and, in square brackets, the number of the line that is being executed or that is suspended. It does not show which statement on the line is executing if the line has multiple statements.

Suspended functions are those that have stopped because of an error or an interrupt. They are marked in the state indicator by a star. **Pendent functions** are those that have called a subfunction that has stopped. They appear in the state indicator without a star. The execute or evaluated input primitives will appear in the state indicator as ⍎ and ⎕ if a function they call suspends. (See Section 1-10.)

A call to a user-defined function interrupts the calling function statement and control goes to the called function until its execution is complete. The state indicator adds a new top line to the previous display. This new line shows the name of the called function and identifies the line that is executing or suspended. Thus, there is more than one line in the state indicator if it is displayed or examined under program control while the second function is executing. The top line disappears when a function named in that line finishes its execution, and control passes back to the line of the function that called it.

A function that calls itself directly or indirectly is **recursive**. A recursive function should be coded with a branch test so that it does not call itself again every time it is called. If too many recursive calls are made, the state indicator fills as it tracks them, finally producing an error message.

The execute function (⍎) and evaluated input (⎕) can conditionally execute simple or compund statements. While they are executing, the state indicator shows a line containing ⍎ or ⎕ (see Section 1-10).

A stop can be set on any line of an unlocked function using a stop vector

result ← *linenumbers* ⎕*STOP* *functionname*

or on some systems,

S∆*function name* ← *linenumbers*

This technique is useful primarily in debugging functions. Function execution can be monitored with

result ← *linenumbers* ⎕*TRACE* *functioname*

or on some systems

T∆*functioname* ← *linenumbers*

Statement Separator (⋄)

The diamond (⋄) separates multiple statements on a function line, in immediate execution mode, or in the character argument to the execute (⍎) function.

The leftmost statement of such a sequence is executed first, followed by the succeeding statements in left-to-right order.

When control branches to a function line, execution begins with the leftmost statement. Thus, statements separated by diamonds on a line of a function are a structural block of code. You can escape the block by branching out, but you can only re-enter at the leftmost statement.

Labels

Labels are most useful in user-defined functions. They are variables local to the function in which they are defined and contain the number of the function line that they begin. Like any other local variables, labels are known to lower-level functions unless they are shadowed.

A given label is defined only once in a given function by appearing to the left of a colon (:). The colon separates the label from the statement in the function line and establishes the label for possible use elsewhere. Labels are used mainly in branch statement expressions, but they can be used in any computation.

Branching

The branch arrow (→) is used with APL expressions that calculate the next function line to be executed. These calculations are usually based on labels or the constant `0`. The branch is a monadic or niladic function that can take a line number as its argument. Following are the results of branching with various values of *v* (which must be an integer vector or scalar).

- If *v* is empty, do not branch, but execute the next statement in sequence.

- If *v* is not empty, transfer immediately to the beginning of the function line whose number is the first element of *v*. If *v* has more than one element, all elements after the first are ignored. Execution always begins with the leftmost statement in the target line, even if the line has a sequence of statements separated by diamonds (◊).

- If the first element of *v* is not a line number in the body of the function, exit from the function, returning control to the point of call. The function header line (line `[0]`) does not count as an executable line of the function, so →`0` can be used to exit a function.

Branching only redirects the flow of execution within the most recently called function. The number branched to is always a line number in that function, even if a ⍎ or ⎕ appears in the state indicator above it.

A branch statement can appear anywhere in a sequence of statements separated by diamonds. If the branch action is other than branch to an empty array, none of the remaining statements in the sequence will be executed. A variety of techniques can be used to create the vector of values provided to →; for example:

- Unconditional branch
```
→LABEL
 .
 .
 LABEL: . . .
```
- Exit from function
```
→0
```
- Conditional branch
```
→(X≥0)⍴NONEG

→(∧100≥,MAT)⍴THEN
'*DATA IS TOO LARGE '⋄→0
THEN:
```
- Loop *n* times
```
I←0
LOOPTOP:→(N<I←I+1)⍴ENDLOOP
(...iterative calculation...)
→LOOPTOP
ENDLOOP: . .
```
- Indexed Branch
```
→(C1,C2,C3,C4)[CASENUM]
```

Note: Do not use the same name to label more than one line in a function, since only one line can be reached by branching to that label.

A loop is a sequence of statements repeated by branching back to the beginning. It is typically controlled by branching back only if some condition is met or by branching back unconditionally but branching out of the loop if some condition is met.

Loops are useful for repetitive tasks like reading and processing successive components of APL⋆PLUS SHAREFILE files. In APL, however, they are generally not needed to handle the elements of arrays as they are in many other programming languages. Using the array-handling capabilities of APL to reduce the programming task and execution time needed for such cases is generally faster and easier than using loops. For example, *+/MATRIX1-MATRIX2* will give the row sums of the table of differences between the corresponding positions in the two matrices. This technique saves a number of explicitly programmed loops with user-defined and user-controlled temporary storage.

The each (¨) operator also eliminates loops (see Section 1-11). APL code written without loops is sometimes more readable and often more efficient.

Ending Execution

The niladic branch (→) ends the current execution. The niladic branch can appear as a statement in a function or it can be entered from the keyboard. If executed from the keyboard, the niladic branch removes the most recent sequence of pendent executions, if any, from the state indicator (see *)RESET* and *)SI* in Chapter 2).

Restartable Statements and Functions

Since branching can only direct execution to the beginning of a numbered function line, a function is only **restartable** if each line can safely be executed starting at the beginning. Restartability is good practice, but not imperative to good APL code. If a statement following a diamond halts because of an error, you cannot return to the halted statement after fixing the problem without repeating the preceding statement(s). Do not, therefore, use a statement followed by a diamond and another statement unless repetition of the earlier statement will yield the same results the second time as the first time. For example, a calculation based on variables that have not yet changed is acceptable, and using *⎕FREPLACE* to replace the value into the position in which it was already placed is also acceptable. However, a second use of *⎕FAPPEND* would put an additional component on file, increasing the file length.

Similarly, a calculation that is stored in one of the variables referenced earlier on the line prevents a second execution from yielding the same result as the first; for example:

```
X←+/Y ◇ Y←0 ◇ Z←Xρ' '
```

If you do not plan each function line to be restartable, you may have to use *)RESET* and repeat the entire application if it halts. Branching back into the function at the point where it stopped is faster and more convenient (use →⎕*LC*). To ensure restartability, use multiple function lines, breaking long statements where they would become non-restartable.

1-9 Execute, Scan, Domino, and Grade

This section describes some advanced APL functions in detail: the execute function (⍎), the "domino" functions matrix divide and matrix inverse (⌹), the grade functions (⍒ and ⍋), and the scan operator (\). Throughout this section, the term "represented statement" refers to the APL statement that the argument represents.

Execute ⍎

Syntax: ⍎ *data*
result ← ⍎ *data*

The execute primitive function accepts a character image of a well-formed APL statement and evaluates that statement as if it were entered from the keyboard. Some of its uses are conditional execution, conversion of

numeric constants, and a limited form of passing unevaluated arguments to functions.

A simple example of execute is

```
      ⍎'2+2'
4
```

The argument to execute is a character singleton or vector. It can represent a simple or compound statement.
Since the argument can be constructed from several different parts, the execute function can be used to perform conditional execution. For example, `M←⍎'M',⍕N` would execute `M←M0` if `N` was `0`; `M←M1` if `N` was `1`, and so on.

You can also use execute to convert character vectors representing numeric constants to their numeric values.

```
      A←⍎'1 2 3'
      A+1
2 3 4
```

(See also `⎕FI` and `⎕VI` in Chapter 3.)

Since system commands are not APL statements, they cannot be "executed" by this function.

Execute can call itself recursively.

Presence of Explicit Results

Whether the execute function returns an explicit result depends upon whether the represented statement, when evaluated, returns an explicit result. If it does, the result of the represented statement is the result of execute. If it does not, execute has no result.

`⍎'1+2×⍳⌊0.5×⍴V'`	Returns an explicit result.
`⍎'⎕FUNTIE 1'`	Does not return an explicit result.

Consequently, the first statement in the preceding example can be embedded in a larger statement:

```
      V[⍎'1+2×⍳⌊0.5×⍴V']
```

but the second statement cannot.

```
      A←⍎'⎕FUNTIE 1'
```

```
VALUE ERROR
      A←⍎'⎕FUNTIE 1'
      ∧
```

If the represented statement does not develop a value, the calling environment should not require that a value be returned in order to avoid a *VALUE ERROR*. Statements that result in no value are

- a user-defined, primitive, or system function that terminates without returning a result
- a branch
- an empty or all-blank statement
- a comment.

Display of Explicit Results

If execute returns an explicit result, the result is displayed only if the result would normally be displayed.

`⍎'⍳5'`	Displays a value.
`⍎'A←⍳5'`	Does not display a value
`T←⍎'⍳5'`	Does not display a value.

Evaluation of Compound Statements

Several statements can be evaluated in one call to execute if they are separated by diamonds in the represented statement.

```
⍎'A←B/⍳⍴B ⋄ RA←⍴A'
```

In this case, the value (if any) returned by execute is determined by the last statement evaluated. Results from other statements are displayed if appropriate.

Occurrence in State Indicator

If execute has been invoked but has not completed execution, it appears in the state indicator as a separate line. For example, if *FN* is a function invoked by ⍎'FN' or a latent expression (⎕LX), and its execution is suspended on line [3], then the state indicator appears as:

```
      )SI
FN[3] *
⍎
```

A pendent call to execute is not represented in the vector of line numbers (⎕LC) in the state indicator.

Relationship between Execute and Its Calling Environment

Upon successful completion of any statement, the system examines three **potentials** that were set during evaluation of the argument:

- Branch potential indicates whether the last statement evaluated is a successful branch.

- Value potential indicates whether the last statement evaluated returns a value.

- Display potential indicates whether the value of the last statement evaluated is to be displayed. If the last statement evaluated returns no value, display potential is undefined.

When the execute primitive completes, the setting of these potentials is determined by the last statement evaluated. These potentials are normally considered and acted upon at the completion of evaluation of each simple statement. However, for the last simple statement evaluated in a statement created by use of execute, consideration of the potentials is deferred to the calling environment.

If any statement evaluated by execute results in a successful branch:

- No more statements of a compound statement are evaluated.
- The branch potential is set to on.
- Execute returns to the calling environment.

Otherwise, the branch potential is off.

Value and display potentials are related in that display potential implies value potential, but value potential does not imply display potential.

Only four combinations of potentials can occur, shown in the following table (0=Off, 1=On, U=Undefined).

	Potential		
Branch	**Value**	**Display**	**Example**
0	0	0	`⍎'⎕FUNTIE 1'`
0	1	0	`⍎'A←⍳5'`
0	1	1	`⍎'⍳5'`
1	0	U	`⍎'→0'`

The calling environment of execute may or may not require that a value be returned.

`⍎'⎕FUNTIE 1'`	Does not require a value.
`A←⍎'⎕FUNTIE 1'`	The assignment requires a value.

If the calling environment does not require a value and the branch potential is on, then the branch is taken. However, an escape (`⍎'→'`) is acted upon immediately without consideration of the calling environment.

If the calling environment requires a value and the value potential is off, then a *VALUE ERROR* is reported with the caret (`^`) pointing to the execute (`⍎`) symbol. In this case, the represented statement is evaluated and any side effects that might be caused by that evaluation occur.

If the calling environment does not require a value and the value potential is on, then the value is displayed according to the setting of the display potential.

Error Reports During Execution of the Represented Statement

Error conditions occurring during execution of the represented statement immediately display an error message, the statement in error, and the caret.

The statement containing the error is displayed, rather than the one at the level of the calling environment of execute.

```
        ⍎'A←⎕FUNTIE 1'
VALUE ERROR
⍎       A←⎕FUNTIE 1
        ^
```

The execute symbol is displayed in the left margin to indicate that the statement originated from a call to execute.

Scan \

Syntax: *result* ← *f**a*
result ← *f*⍀*a*
result ← *f*\\[*k*]*a*

f	any scalar dyadic function
a	any APL array
k	specified scan coordinate

The scan operator complements and extends other APL functions by producing the results of successive reductions. (See the reduction example in Section 1-5.) The scan operator combines with any primitive scalar dyadic function to form a new monadic function. The new function forms successive elements in the result by applying the scalar dyadic function to successive take (↑) operations of the right argument using reduction. The shape of the result is identical to that of the right argument.

Scan has many uses, including the calculation of cumulative sums and products and the manipulation of Boolean data.

The definition of scan for a vector *V* is as follows:

Let *result* ← *f**V*.
Then, *result*[*I*] ← is defined as *f*/*I*↑*V* for all *I* ∊ ⍳⍴*V* in origin 1.

For arrays of rank 2 or greater, the function is applied along the implicit or explicit coordinate, similar to reduction. For example, you can specify the scan coordinate by writing:

*f**a*
f⍀*a*
f\\[*k*]*a*

as it is applied along the last, first, or *k*th coordinate, respectively.

Examples

```
      TRANSACTIONS ← 100 5 ¯20 3 ¯50
      +\TRANSACTIONS     Calculates running account
100 105 85 88 38         balances.
```

Scans of Boolean vectors by relational and logical functions are particularly useful. For a Boolean vector *BV*, the following are true:

If *R*←∧*BV* then *R*↔*BV* with all 0s after the first 0 in *BV*.
If *R*←<*BV* then *R*↔*BV* with all 0s after the first 1 in *BV*.
If *R*←≤*BV* then *R*↔*BV* with all 1s after the first 0 in *BV*.
If *R*←∨*BV* then *R*↔*BV* with all 1s after the first 1 in *BV*.

≠*BV* ↔ parity of the cumulative number of 1s.
=*BV* ↔ reverse parity of the cumulative number of 0s.

Identities

The following identities hold for any Boolean array *B*:

```
      <\B  ↔  ~≤\~B
      ≤\B  ↔  ~<\~B
      ≥\B  ↔  ~>\~B
      >\B  ↔  ~≥\~B
      =\B  ↔  ~≠\~B  ↔  ~2|+\~B
      ≠\B  ↔  ~=\~B  ↔  2|+\B
      ∨\B  ↔  ~∧\~B
      ∧\B  ↔  ~∨\~B
```

```
⍱\B  ←→  ~⍲\~B  ←→  (≥\B)=(∨\B)≤<\B
⍲\B  ←→  ~⍱\~B  ←→  (>\B)≠(∧\B)<≤\B
```

Applications

Remove leading blanks.

```
(∨\TXT≠' ')/TXT
```

Extract the first word.

```
A←TXT≠' ' ⋄  (A>∨\A<∨\A)/TXT
```

Determine if *V* is in increasing order.

```
∧/V=⌈\V
```

Determine if *V* contains correctly matched and nested parentheses.

```
∧/0=⌊\⌽+\-/V∘.='()'
```

Implementation Considerations

As noted previously, scan is defined as follows:

Let *result* `← f\V`.
Then, *result*`[I]  ←→  f/I↑V` for all `I ∊ ⍳⍴V` in origin 1.

For the associative functions + and ×, the following definition is used to reduce execution time. This definition is formally equivalent, but not always computationally equivalent, to the preceding one.

Let *result* `← f\V`.
Then, *result*`[1]  ←→  V[1]` and *result*`[I]  ←→` *result*`[I-1] f V[I]` for all `I ∊ 1↓⍳⍴V` in origin 1.

For arguments whose values differ significantly in magnitude, the two definitions may not return the same results. The following example shows that the two definitions may also differ from the exact answer.

```
Let V ← ¯1 1E20 ¯1E20 1
First definition:  +\V  ←→  ¯1  1E20 ¯1  ¯1
Second definition: +\V  ←→  ¯1  1E20   0   1
Exact definition:  +\V  ←→  ¯1  9.999...E19  ¯1  0
```

In this case, the exact answer cannot be returned because of the limited precision used within the computer.

For maximum-scan (`⌈\`) and minimum-scan (`⌊\`), the two definitions always produce the same results.

Matrix Division and Inversion

Syntax: *result* ← ⌹ *r*
result ← *l* ⌹ *r*

l a scalar, vector, or matrix
r a scalar, vector, or matrix

Either *l* or *r* is a scalar, or the first elements of the shapes of *l* and *r* must be equal.

For calculation purposes, matrix divide treats vector and scalar arguments as one-column matrix arguments. Conformability tests are based on the arguments treated this way, and a *LENGTH ERROR* occurs when the left and right arguments have an unequal number of rows.

The shape of the resulting matrix is determined by the shape of the arguments. For matrix inversion, it is the dimensions of the argument in reverse order:

```
ρ⌹A  ←→  ⌽ρA
```

For matrix division, the result has as many rows as the left argument had columns, and as many columns as were in the right argument.

```
ρB⌹A  ←→  (1↓ρA),(1↓ρB)
```

If the right argument is a scalar, a one-element vector, or a one-row by one-column matrix, matrix divide is equivalent to divide, except for minor differences in the shape of the result and except when both arguments are zero.

Matrix divide (dyadic domino) is used to solve matrix equations in much the same way that dyadic ÷ is used to solve scalar equations. It is primarily used to solve equations of the form MX=R (the matrix product MX is expressed in APL notation as *M*+.×*X*) where:

- M is a given matrix.
- R is a given vector (considered for matrix divide as a one-column matrix having the same number of rows as M).
- X is an unknown vector.

If such an equation has a unique solution X, then *X*←*R*⌹*M*. If it has more than one solution, then *R*⌹*M* will produce a *DOMAIN ERROR*. In fact, *R*⌹*M* will produce a *DOMAIN ERROR* whenever the matrix *M* is singular (a non-zero vector *V* exists for which *M*+.×*V* is the zero vector). If *M* has more rows than columns, is not singular, and the equation MX=R does not have a solution, then *R*⌹*M* yields the vector that most closely approximates the solution (the least squares approximation).

Matrix inverse (monadic domino) yields the inverse of a matrix *M* if *M* is non-singular and square. If *M* is non-singular and has more rows than columns, matrix inverse yields the least squares approximation to the right inverse of *M*.

Applications

The following examples show applications of ⌹.

Solving Linear Equations

Use ⌹ to solve a system of linear equations such as:

```
2x -  y + 3z =  12
-x + 4y - 2z = -11
3x +  y + 5z =  17
```

This system is equivalent to the matrix equation MX=R where M is the matrix of coefficients of the left side of the equation:

```
      M←3 3⍴2 ¯1 3 ¯1 4 ¯2 3 1 5 ⋄ M
 2 ¯1  3
¯1  4 ¯2
 3  1  5
```

X is the vector with elements x, y, z, and *R*←12 ¯11 17. Therefore, *X*←*R*⌹*M* will yield (the best approximation to) the solution of this system (since *M* is non-singular).

```
      X←R⌹M ⋄ X
1 ¯1 3
```

In fact, *R*⌹*M* yields the exact solution as shown by multiplying it back:

```
      M+.×X
12 ¯11 17
```

Fitting a Straight Line

Matrix divide can also be used in curve fitting. In many experiments, the object is to find a mathematical function that closely approximates empirical measurements. To find the straight line that comes closest to passing through a given set of points, you must find the values c and d so that the line with equation dx + c comes closest to the given values for x and y. For example, if we take the four points

(1.1, 2.3), (1.9, 4.0), (3.05, 6.3), and (4.1, 7.9)

and view them as points on our line, each point provides a value for x and a value for y to substitute in our general equation, giving us a system of four equations representing these data points:

$1.1d + c = 2.3$
$1.9d + c = 4.0$
$3.05d + c = 6.3$

4.1d + c = 7.9

As in the previous example, the closest possible least squares solution for such a system of equations is *C*←*Y*⌹*M*, where C contains the values of d and c, *Y* is the vector of y coordinates of the points, and *M* is the matrix *M*←*X*∘.*1 0 where *X* is the vector of x coordinates of the points. Applying this to the equation yields:

```
      Y←2.3 4.0 6.3 7.9
      X←1.1 1.9 3.05 4.1
      M←X∘.*1 0 ◊ M
1.1   1
1.9   1
3.05  1
4.1   1
```

Using matrix division to find the solution yields:

```
      C←Y⌹M ◊ C
1.876856212 0.3624773633
```

These results indicate that the linear equation which best approximates these points is

1.876856212x + 0.3624773633 = y

Fitting a Polynomial Curve

Similarly, the coefficients of the polynomial of degree *D* that most closely fit a set of data points can be obtained using the formula *C*←*Y*⌹*M*←*X*∘.*⌽0,⍳*D* (in origin 1). Applying this to our original data yields the coefficients *C* of the polynomial of degree 2 that best approximate them.

```
      C←Y⌹M←X∘.*2 1 0 ◊ C
¯0.153408846 2.676735268 ¯0.480885961
```

To see how closely the polynomial with these coefficients approximates our data points, we evaluate it for x = 3.05, using the polynomial evaluation function (⊥):

```
      3.05⊥C
6.256070817
```

This result is very close to the y value of 6.3. To see how closely this comes to all our data points, we use the polynomial evaluation function ⊥ again:

```
      (4 1⍴X)⊥C
2.27789813 4.051105114 6.256070817
7.914925938
```

Computational Accuracy and Efficiency

Although X←R⌹M and X←(⌹M)+.×R are equivalent APL statements, they will generally yield slightly different results when computed because of roundoff errors. The expression X←R⌹M will produce faster and more accurate results. Similarly, when solving several equations with the same coefficient matrix, such as

```
      X1←R1⌹M ⋄ X2←R2⌹M ⋄ X3←R3⌹M
```

it is more efficient to solve the single equation X←R⌹M where R is the matrix whose columns are R1, R2, and R3; and X is the matrix with columns X1, X2, and X3.

Sorting with the Grade Up and Grade Down Functions

Monadic grade up and grade down provide permutation vectors to sort only numeric data along the first coordinate. Dyadic grade up and grade down arrange only character data, but allow for arbitrary collating sequences. They are discussed separately below.

Monadic Grade

Syntax: *result* ← ⍋*data*
result ← ⍒*data*

data any non-scalar numeric array

The grade up and grade down monadic primitives arrange the indices of numeric data in ascending or descending order.

The *result* is always a numeric vector whose length is the same as the first dimension of the argument. For vector arguments, the result can be used as a subscript vector to arrange the argument into ascending (for grade up) or descending (for grade down) order. Duplicate values will retain their original relative positions.

In the case of two-dimensional (matrix) arguments, the result is formed by considering one column at a time, working from left to right. An initial ordering is generated by considering the leftmost column as a vector. If the vector has no duplicate values, the initial ordering becomes the result. If the vector does have duplicate values, then data from the next column to the right is used in an attempt to resolve the duplications. This process continues until either all duplications are resolved or all columns are used.

Arguments of more than two dimensions are treated as matrices, retaining the original first dimension and combining all the other dimensions into a single second dimension. In effect, the data is treated as being reshaped as follows:

```
        ((1↑ρA),×/1↓ρA)ρA
```

Some examples of monadic grade follow.

```
        ⎕IO←1
        ⍋ 17 2 14
2 3 1
        17 2 14[2 3 1]          Increasing sort.
2 14 17

        ⍒⎕← 3 4ρ 1 4 9 2 1 7 7 6 1 9 3 0
1 4 9 2
1 7 7 6
1 9 3 0
3 2 1
```

Dyadic grade up and grade down

Syntax: *result* ← *order* ⍋ *data*
result ← *order* ⍒ *data*

data	a character array
order	a character array used to establish the relative ordering of the characters in *data*

The grade up and grade down dyadic primitives arrange character data in ascending or descending order. Both arguments must be non-scalar arrays.

The left argument associates numeric values with each character in the right argument. The rules of monadic grade up or grade down are then applied to the associated numeric values to produce the result.

If the left argument is a vector, then the associated numeric values are equivalent to those produced by dyadic iota. Specifically, *V*⍋*A* is equivalent to ⍋*V*⍳*A*.

For left arguments of rank 2 or greater, each dimension is used independently, working from the last to the first. The numeric ordering value for any given character of the right argument with respect to a specified dimension of the left argument requires consideration of all occurrences of the characters in the left argument. The ordering value is taken as the minimum of the coordinate value along the specified dimension for these occurrences. If a character does not appear in the left argument, its ordering value is determined much like that of dyadic iota.

Ordering values are initially determined with respect to the last dimension of the left argument. The rules of monadic grade are then applied to the associated values, including duplications, to produce an ordering. If this ordering contains no duplications, or if no further dimensions of the left argument remain to apply, the process is complete. Otherwise, the ordering values are recalculated with respect to the next higher dimension, and the resolution process is reinvoked starting with the first column of the right argument. This process continues until either all duplications are resolved, or until all dimensions of the left argument have been exhausted.

Suppose the following matrix is used as the left argument (on some terminals the underscored letters are displayed as lowercase letters):

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
```

The initial ordering using the last dimension will result in *A* and *a* coming before *B* and *b*, and so on. If both *A* and *a* appear in the right argument, they will appear as duplications since they have identical coordinate values (and ordering values) along the last dimension. A second evaluation will then occur using the first dimension. This will give a further reordering placing *A* before *a*.

In the next example, three collating sequences (each starting with a blank) are used to produce the three different results shown in the following table.

Collating Sequence 1:

```
 abcdefghijklmnopqrstuvwxyzABCDEFGHIJ
KLMNOPQRSTUVWXYZ
```

Collating Sequence 2:

```
 aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrR
sStTuUvVwWxXyYzZ
```

Collating Sequence 3:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Original Data	**Sort with Collating Sequence 1**	**Sort with Collating Sequence 2**	**Sort with Collating Sequence 3**
Ama	acid	acid	acid
YMCA	ama	ama	ama
Trudgen	ammonia	ammonia	Ama
Tektite	pavilion	Ama	AMA
pi	phosphate	AMA	ammonia
pavilion	pi	NSPF	NSPF
piping	piping	pavilion	pavilion
pump	pump	phosphate	pH
underwater	pH	pH	Philodendron
tsunami	trudgen	pi	phosphate
NSPF	tsunami	piping	pi

larg the left argument
rarg the right argument
res the explicit result

Functions

Conjugate **Return the value of a nu**
res ← + *arg*
arg : any numeric array
res : same as 0+*arg*

```
      +¯27.34 18
¯27.34 18 6
```

Plus **Add two numbers**
res ← *larg* + *rarg*
larg, *rarg* : any numeric ar
res : each item of *larg* add of *rarg*

```
      ¯2 2 2 + 3
1.5 3 0
```

Negate **Change the sign of a num**
res ← − *arg*
arg : any numeric array
res : each item of *arg* subtr

```
      − 2 ¯2 1.5
¯2 2 ¯1.5
```

Minus **Subtract two numbers**
res ← *larg* − *rarg*
larg, *rarg* : any numeric arr
res : each item of *rarg* subt corresponding item of

```
      ¯2 2 2 − 3.
¯5.5 1 4
```

Tsunami	underwater	pump	piping
trudgen	Ama	Philodendron	pump
pH	AMA	trudgen	Tektite
phosphate	NSPF	tsunami	trudgen
ammonia	Philodendron	Tektite	Trudgen
AMA	Tektite	Trudgen	tsunami
Philodendron	Trudgen	Tsunami	Tsunami
acid	Tsunami	underwater	underwater
ama	YMCA	YMCA	YMCA

Note: The above examples all use dyadic ⍋; if dyadic ⍒ had been used, the order of the results would have been exactly reversed. Although *CM*[⎕*AV*⍒*CM*;] and ⊖*CM*[⎕*AV*⍋*CM*;] are equivalent, that ⎕*AV*⍒*XM* and ⌽⎕*AV*⍋*CM* are not identical unless there are no duplicates.

1-10 Primitive Function and Operator Reference

This section summarizes the APL primitive functions and operators. Each function and operator is listed with its syntax, a brief description, and one or more examples. In some examples a variable or result is shown in "display" form (Section 1-1) rather than the standard output typically generated by the system. This display form graphically illustrates the data structures and is produced by ⎕*SHOW* on some systems and by a display function on others. Recall that an array can be classified as a scalar, vector, matrix, or *n*-dimensional.

The following abbreviations are used throughout this section:

arg	the argument
conforming	the left and right arguments must have the same type and shape
ext	external factor that affects the result of this operation (e.g. ⎕*CT*, ⎕*RL* , ⎕*IO*)
f∘*g*	any dyadic function, whether a primitive function (+,−,×,÷, etc.), a system (e.g. ⎕*FREAD*), or a user-defined function.
i	positive integer scalar
idx	index or variable with valid indices

larg	the left argument
rarg	the right argument
res	the explicit result

Arithmetic Functions

+ Conjugate

Return the value of a number

res ← + *arg*

arg : any numeric array

res : same as `0+`*arg*

```
      +¯27.34 18 6
¯27.34 18 6
```

+ Plus

Add two numbers

res ← *larg* + *rarg*

larg, *rarg* : any numeric array (conforming)

res : each item of *larg* added to corresponding item of *rarg*

```
      ¯2 2 2 + 3.5 1 ¯2
1.5 3 0
```

− Negate

Change the sign of a number

res ← − *arg*

arg : any numeric array

res : each item of *arg* subtracted from zero

```
      - 2 ¯2 1.5
¯2 2 ¯1.5
```

− Minus

Subtract two numbers

res ← *larg* − *rarg*

larg, *rarg* : any numeric array (conforming)

res : each item of *rarg* subtracted from corresponding item of *larg*

```
      ¯2 2 2 - 3.5 1 ¯2
¯5.5 1 4
```

× **Signum** — **Determine the sign of a number**

res ← × *arg*
arg : any numeric array
res : ¯1 if *arg* is negative, 0 if *arg* is zero, and 1 if *arg* is positive.

```
      × 3 0 ¯0.5
1 0 ¯1
```

× **Times** — **Multiply two numbers**

res ← *larg* × *rarg*
larg, *rarg* : any numeric array (conforming)
res : each item of *larg* multiplied with corresponding item of *rarg*

```
      ¯2 2 2 × 3.5 0 2
¯7 0 4
```

÷ **Reciprocal** — **Find the reciprocal of a number**

res ← ÷ *arg*
arg : any non-zero numeric array
res : one divided by each item of *arg*

```
      ÷ 2 ¯1 ¯0.5
0.5 ¯1 ¯2
```

÷ **Divide** — **Divide two numbers**

res ← *larg* ÷ *rarg*
larg : any numeric array
rarg : any numeric array (conforming)
res : each item of *larg* divided by corresponding item of *rarg*

```
      2 ¯3 0 ÷ 1 3 0
2 ¯1 1
      0÷0
1
```

⋆ Exponential

Raise e to a power

res ← ⋆ *arg*
arg : any numeric array
res : e (2.71828...) raised to the power specified by each item of *arg*

```
      ⋆ 1 ¯1 0
2.718281828 0.3678794412 1
```

⋆ Power

Raise a number to a specific power

res ← *larg* ⋆ *rarg*
larg, *rarg* : any numeric array (conforming)
res : *arg* raised to the corresponding *rarg* power

```
      2 49 4 0 ⋆ 3 0.5 ¯1 40
8 7 0.25 0
```

⌈ Ceiling

Round up to the nearest integer

res ← ⌈ *arg*
arg : any numeric array
res : smallest integer greater than or equal to *arg*
ext : ⎕*CT*

```
      ⌈ 3.1416 ¯1.5 6
4 ¯1 6
```

⌈ Maximum

Select the greater of two numbers

res ← *larg* ⌈ *rarg*
larg, *rarg* : any numeric array (conforming)
res : the larger of each corresponding pair of numbers in *larg* and *rarg*

```
      ¯3.2 ¯4.1 ⌈ 7 ¯4.2
7 ¯4.1
```

⌊	**Floor**	**Round down to the nearest integer**

res ← ⌊ *arg*
arg : any numeric array
res : largest integer less than or equal to *arg*
ext : ⎕CT

```
      ⌊ 3.1416 ¯1.5 6
3 ¯2 6
```

⌊	**Minimum**	**Select the lesser of two numbers**

res ← *larg* ⌊ *rarg*
larg, *rarg* : any numeric array (conforming)
res : the lesser of each corresponding pair of numbers in *larg* and *rarg*

```
      ¯3.2 ¯4.1 ⌊ 7 ¯4.2
¯3.2 ¯4.2
```

|	**Magnitude**	**Compute the absolute value of a number**

res ← | *arg*
arg : any numeric array
res : the absolute value (or magnitude) of each element of *arg*

```
      | 2 0 ¯1.6
2 0 1.6
```

|	**Residue**	**Find the remainder after the division of two numbers**

res ← *larg* | *rarg*
larg, *rarg* : any numeric array (conforming)
res : the remainder after dividing each corresponding item of *rarg* by *larg*
rarg - (⌊*rarg*÷*larg*) ×*larg*

```
      2 ¯2 1 | 3 3 3.14159
1 ¯1 0.14159
```

⍟ Natural Logarithm

Compute the natural logarithm of a number

res ← ⍟ *arg*

arg : any positive numeric array

res : the logarithm (base *e*) applied to each item of *arg*

```
      ⍟ 1 10 2.7182818284
0 2.302585093 1
```

⍟ Logarithm

Compute the logarithm of a number

res ← *larg* ⍟ *rarg*

larg, *rarg* : any positive numeric array (conforming)

res : the logarithm of each element of *rarg* to the corresponding base in *larg*

```
      2 49 4 ⍟ 8 7 0.25
3 0.5 ¯1
```

○ Pi times

Multiply a number by Pi

res ← ○ *arg*

arg : any numeric array

res : *arg* multiplied by Pi (3.141592...)

```
      ○ 1 2 0
3.141592654 6.283185307 0
```

○ Trigonometric functions

Compute a Trigonometric function for a number

res ← *larg* ○ *rarg*

larg : any array of integers in the range -7 to +7

rarg : any valid numeric array (conforming)

res : the trigonometric function selected by *larg* applied to each corresponding item in *rarg*

Note: all arguments and results are in radians.

larg	*function*	*larg*	*function*
¯7	*ARCTANH*	7	*TANH*
¯6	*ARCCOSH*	6	*COSH*

? Roll

Select a random

res ← ? *arg*

arg : any positive

res : an integer p numbers gi random num ⎕IO ≤ re

ext : ⎕IO, ⎕RL

```
      ? 200
1969 2 23
```

? Deal

Select a set of uni

res ← *larg* ? *rarg*

larg, *rarg* : a posit

res : *arg* unique ra possible posi

ext : ⎕IO, ⎕RL

```
      8 ? 1
1 5 3 4 9 6
```

⌹ Matrix Inverse

Calculate the inver

res ← ⌹ *arg*

arg : numeric scala

res : inverse of *arg* square. If *arg* (must have m result is the le the inverse of

```
      ⌹ 2 2
 3 ¯1
¯2  1
```

```
¯5 ARCSINH            5  SINH
¯4 (¯1+rarg*2)*.5     4  (1+rarg*2)*.5
¯3 ARCTAN             3  TAN
¯2 ARCCOS             2  COS
¯1 ARCSIN             1  SIN
 0 (1-rarg*2)*.5
```

```
      0 ○ .6
0.8

      2 ○ 3.14159
¯1

      ¯3 ○ 0 1 2
0 0.7853981634 1.107148718
```

! **Factorial** — **Compute the Factorial of a number**

res ← ! *arg*

arg : any numeric array

res : if *arg* is a positive integer, *res* is the product of all positive integers from 1 through *arg*. If *arg* is zero, *res* is 1. All other numbers except negative integers are computed using the gamma function on *arg*+1; the function is undefined for negative integers.

```
      ! 0 4 2.5
1 24 3.32335097
```

! **Binomial** — **Find the number of permutations for a set of objects**

res ← *larg* ! *rarg*

larg, *rarg* : any positive numeric array (conforming)

res : the number of permutations of selecting *larg* objects at a time from *rarg* objects, for each corresponding *larg*, *rarg* pair of numbers

```
      1 2 5 ! 5 4 5
5 6 1
```

? **Roll**

Select a random integer

res ← ? *arg*

arg : any positive integer array

res : an integer picked at random from the set of numbers given by ⍳*arg*[*n*]; *res* contains a random number for each element of *arg* where ⎕IO ≤ *res* ≤ *arg*[*n*]

ext : ⎕IO, ⎕RL

```
      ? 2000 12 30
1969 2 23
```

? **Deal**

Select a set of unique random integers

res ← *larg* ? *rarg*

larg, *rarg* : a positive integer scalar

res : *arg* unique random integers selected from *rarg* possible positive integers (i.e. ⍳ *rarg*)

ext : ⎕IO, ⎕RL

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

⌹ **Matrix Inverse**

Calculate the inverse of a matrix

res ← ⌹ *arg*

arg : numeric scalar, vector or matrix

res : inverse of *arg* if *arg* is non-singular and square. If *arg* is non-singular but not square (must have more rows than columns) the result is the least squares approximation to the inverse of *arg*.

```
      ⌹ 2 2 ρ 1 1 2 3
 3 ¯1
¯2  1
```

⌹ Matrix Divide

Solve a set of simultaneous equations

res ← *larg* ⌹ *rarg*

larg, *rarg* : numeric scalar, vector or matrix; rank of *rarg* must equal or exceed rank of *larg;* if *rarg* is a matrix, last dimension must not exceed the first

res : the exact solution (or a least squares approximation if *rarg* has more rows than columns) of the matrix equation *rarg* • X = *larg* (see Section 1-9 for more details)

```
      14 26 ⌹ 2 2⍴1 3 4 2
5 3

      14 26 7 ⌹ 3 2⍴1 3 4 2 1 1
4.981481481  2.944444444
```

⊤ Representation

Find the representation of a number in another radix

res ← *larg* ⊤ *rarg*

larg, *rarg* : any numeric array

res : the expression of each element of *rarg* represented in a number system described by *larg*

```
      10 10 10 10 ⊤ 1776
1 7 7 6

      2 2 2 ⊤ 5
1 0 1

      7 24 60 ⊤ 5090
3 12 50

      7 24 60 ⊤ 5090 6666
 3    4
12   15
50    6
```

⊥ **Base Value**

Find the base value of a number

res ← *larg* ⊥ *rarg*
larg, *rarg* : any numeric array
res : the expression of *rarg* in radix *larg*

```
      10 ⊥ 1 7 7 6
1776

      10 3 2 10 ⊥ 1 7 7 6
276

      2⊥1 0 1 0
10

      7 24 60 ⊥ 3 12 50
5090
```

Logical Functions

< **Less than**

Compare two numeric arrays

res ← *larg* < *rarg*
larg, *rarg* : any numeric array (conforming)
res : 1 for each pair of corresponding values where *larg* is less than *rarg*; 0 otherwise
ext : ⎕CT

```
      1 2 3 < 2 1 3
1 0 0
```

≤ **Less than or equal**

Compare two numeric arrays

res ← *larg* ≤ *rarg*
larg, *rarg* : any numeric array (conforming)
res : 1 for each pair of corresponding values where *larg* is less than or equal to *rarg*; 0 otherwise
ext : ⎕CT

```
      1 2 3 ≤ 2 1 3
1 0 1
```

= **Equal** **Compare two arrays for equality**

res ← *larg* = *rarg*

larg, *rarg* : any array (conforming)

res : 1 for each corresponding value of *larg* and *rarg* that is equal; 0 otherwise

ext : ⎕CT

```
      'S'='STSC'
1 0 1 0
```

≥ **Greater than or equal** **Compare two numeric arrays**

res ← *larg* ≥ *rarg*

larg, *rarg* : any numeric array (conforming)

res : 1 if the corresponding value of *larg* is greater than or equal to *rarg*; 0 otherwise

ext : ⎕CT

```
      1 2 3 ≥ 2 1 3
0 1 1
```

> **Greater than** **Compare two numeric arrays**

res ← *larg* > *rarg*

larg, *rarg* : any numeric array (conforming)

res : 1 if the corresponding value of *larg* is greater than *rarg*; 0 otherwise

ext : ⎕CT

```
      1 2 3 > 2 1 3
0 1 0
```

≠ **Not equal** **Compare arrays for inequality**

res ← *larg* ≠ *rarg*

larg, *rarg* : any array (conforming)

res : 1 for each corresponding value of *larg* and *rarg* that are not equal; 0 otherwise

ext : ⎕CT

```
      1 2 3 ≠ 2 1 3
1 1 0
```

~ **Not** — **Negate a Boolean array**

res ← ~ *arg*

arg : any Boolean array

res : 1 for each item of *arg* that is 0; 0 for each item that is 1

```
      ~ 0 1
1 0
```

∨ **Or** — **Logical OR of two Boolean arrays**

res ← *larg* ∨ *rarg*

larg, *rarg* : any Boolean array (conforming)

res : 1 if either *larg* or *rarg* is 1; 0 otherwise

```
      0 0 1 1 ∨ 0 1 0 1
0 1 1 1
```

∧ **And** — **Logical AND of two Boolean arrays**

res ← *larg* ∧ *rarg*

larg, *rarg* : any Boolean array (conforming)

res : 1 if both *larg* and *rarg* are 1; 0 otherwise

```
      0 0 1 1 ∧ 0 1 0 1
0 0 0 1
```

⍱ **Nor** — **Logical NOR of two Boolean arrays**

res ← *larg* ⍱ *rarg*

larg, *rarg* : any Boolean array (conforming)

res : 1 if both *larg* and *rarg* are 0; 0 otherwise
equivalent to ~ (*larg* ∨ *rarg*)

```
      0 0 1 1 ⍱ 0 1 0 1
1 0 0 0
```

⍲ **Nand** **Logical NAND of two Boolean arrays**

res ← *larg* ⍲ *rarg*

larg, *rarg* : any Boolean array (conforming)

res : 0 if both *larg* and *rarg* are 1; 1 otherwise
equivalent to ~ (*larg* ∧ *rarg*)

```
      0 0 1 1 ⍲ 0 1 0 1
1 1 1 0
```

≡ **Match** **Compare the equivalence of two arrays**

res ← *larg* ≡ *rarg*

larg, *rarg* : any array

res : 1 if both *larg* and *rarg* have the same rank, shape, and values; 0 otherwise

ext : ⎕CT

```
      'XYZZY' ≡ 1 5⍴'XYZZY'
0
      0 ≡ ,0
0

      A←2 3⍴⍳4
      A ≡ A
1
      A ≡ 'A'
0
```

Location Describers and Modifiers

⍳ **Index generator** **Return a set of consecutive integers**

res ← ⍳ *arg*

arg : positive integer scalar

res : a vector of *arg* integers from the sequence
⎕IO, ⎕IO+1, ⎕IO+2,...

ext : ⎕IO

```
      ⍳5
1 2 3 4 5
```

ι **Index of** **Find location of items in an array**

res ← *larg* ι *rarg*

larg : any vector

rarg : any array

res : the index location of the first occurence of the items specified in *rarg* in the *larg* array. For elements of *rarg* that do not occur in *larg*, the result is 1 + ρ*larg* (⎕IO←1).

ext : ⎕IO

```
      A←3 4 7 3 8
      A ι 7 4 3 12
3 2 1 6
```

[ ] **Index into** **Select a subset of elements from an array**

res ←*arg* [idx_1 ; idx_2 ; ...]

arg : any non-scalar array

idx_n : any integer array. There must be one index per axis of arg. Indices are, separated by ";". Missing indices such as in *A* [ ] or *B* [; *J*] indicate that the entire axis should be selected.

res : the portion of the *arg* array specified by *idx*

ext : ⎕IO

```
      A←3 4 7 3 8
      A[Aι7 4 3]
7 4 3

      'ABCD'[3 2]
CB

      (3 4ρι12)[;3]
3 7 11
      'ABC'[4 4ρι3]

ABCA
BCAB
CABC
ABCA
```

∊ **Member of**

Compare contents of two arrays

res ← *larg* ∊ *rarg*

larg , *rarg* : any array

res : the same size as *larg* and contains a 1 if the *larg* item is found anywhere in *rarg*; 0 otherwise

ext : ⎕*CT*

```
      2 5 ∊ 1 2 3 4
1 0
```

↑ **Take**

Select a set of elements from an array

res ← *larg* ↑ *rarg*

larg : any integer scalar or vector with one element per dimension of *rarg*

rarg : any array

res : the subset of *rarg* items. The shape of *res* is specified by *larg*. If *larg* is negative, the selection starts from the end rather than the beginning; *res* is padded with the fill item (The fill item is ∊⊃ *arg* and is blank or zero for simple arrays) if *larg* specifies an array larger than *rarg*.

```
      2 ↑ 3 6 2
3 6

      5 ↑ 3 6 2
3 6 2 0 0

      ¯3 2 ↑ 2 3⍴1 2 3 4 5 6
0 0
1 2
4 5
```

↓ **Drop**

Exclude a set of elements from an array

res ← *larg* ↓ *rarg*

larg : any integer scalar or vector with one element per dimension of *rarg*

rarg : any array

res : all the items of *rarg* except the subset specified by *larg*. *larg* specifies the number of elements in each dimension that should be excluded from the result (starting from the end if *larg* is negative). If an element of *larg* is larger in magnitude than the corresponding dimension of *rarg*, *res* will be empty (have a dimension of zero) along the corresponding coordinate.

```
      5 ↓ 1 3 2 7 4 8
8

      A←2 3ρ1 2 3 4 5 6
      0 ¯1 ↓ A
1 2
4 5
```

⊂ **Enclose**

Create a nested scalar out of any array that is not a simple scalar

res ← ⊂*rarg*

rarg : any array

```
      C←'A' 'MM' 'SSS'

      ρC
3

      C[2]←⊂2 2 ρι4

      C
A    1 2  SSS
     3 4
```

```
        DISPLAY C
.→-------------.
|              |
|  .→--.  .→--.|
|A ↓1 2|  |SSS||
|- |3 4|  '---'|
|  '~--'       |
'∊-------------'
```

⊃ Pick

Select a portion of an array

res ← path ⊃ *arg*
arg : any array
path : positive integers describing now deep into *arg* to go to select an item
res : a subset of *arg* specified by *path*
ext : ⎕*IO*

```
        A←'ONE' (2 2ρι4) 'SIX'
        ρA
3

        DISPLAY A
.→-----------------.
|.→--.  .→--.  .→--.|
||ONE|  ↓1 2|  |SIX||
|'---'  |3 4|  '---'|
|       '~--'       |
'∊-----------------'
        2⊃A
1 2
3 4

        3 2⊃A
I

        (2 (2 1))⊃A
3

        2 ⊃'TEXT'
E
```

⊂ **Partitioned Enclose**

Build a non-simple vector from selected portions of an array

res←*larg* ⊂ *rarg* or *res*←*larg* ⊂[*i*] *rarg*

larg: Boolean vector with same length as selected coordinate of *rarg*
rarg: array of any rank
i: non-negative scalar indicating the dimension desired.
res: selected portions of *rarg*: *res* is a vector of length +/*larg*.

```
      A←0 0 1 0 1 1 0 0 ⊂ι8
      A
 3 4  5  6 7 8
      ρA
3
      DISPLAY A
.→-----------------------.
| .→--.  .→.  .→-----.  |
| |3 4|  |5|  |6 7 8|  |
| '~--'  '~'  '~-----'  |
'∊-----------------------'
```

⊃ **Disclose**

Retrieve the array stored as a nested scalar

res←⊃*rarg*

rarg : any array,
res: if *rarg* is a nested scalar, it will be expanded back to an array

```
      C←'ONE' (2 3 4 5)
      ρC
2
      ρ⊃C[2]
4
```

If *rarg* is an array rather than a nested scalar, the first item is selected and expanded into an array if it is a nested scalar. This is often called the "First" function.

```
      ⊃C
ONE
      ρ⊃C
3
      ⊃1 2 3
1
```

↑ **Mix**

Reduce one level of nesting.

res← ↑ *arg* or *res* ← ↑[*i*] *arg*

arg: any array with identically-shaped items.

i : non-negative scalar indicating the dimension desired

res: the shape is the shape of *arg* with the shape of the items inserted between the specified dimensions

```
      A←(1 2 3 4) (5 6 7 8)

      ρA
2

      ρ¨A
 4  4

      ρ↑A
2 4

      ↑A
1 2 3 4
5 6 7 8

      ↑[.5]A
1 5
2 6
3 7
4 8
```

↓ **Split**

Segment an array into a nested array

res←↓*arg* or *res*←↓ [*i*] *arg*

arg : any array

i : non-negative scalar indicating the dimension desired

res : the contents of *arg* in which the rank has been reducted by one by enclosing all items in the *i*th dimension into a nested scalar. For example, if *arg* is a matrix:

```
res[1]←⊂arg[1;]
res[2]←⊂arg[2;]
...
res[n]←⊂arg[n;]
```

```
      A←3 4⍴⍳12
      A
1  2  3  4
5  6  7  8
9 10 11 12

      ↓A
 1 2 3 4   5 6 7 8   9 10 11 12
      ⍴↓A
3
      DISPLAY ↓A
.→--------------------------------.
|.→-------. .→-------. .→----------.|
||1 2 3 4| |5 6 7 8| |9 10 11 12||
|'~-------' '~-------' '~----------'|
'∊--------------------------------'

      ↓[1]A
1 5 9  2 6 10  3 7 11  4 8 12
      DISPLAY ↓[1]A
.→----------------------------------.
|.→----. .→-----. .→-----. .→-----.|
||1 5 9| |2 6 10| |3 7 11| |4 8 12||
|'~----' '~-----' '~-----' '~-----'|
'∊----------------------------------'
```

⍋ **Numeric Grade Up**

Return ascending sort order of a numeric array

res ← ⍋ *arg*

arg : any numeric non-scalar array

res : the indices of *arg* that would arrange it in ascending numeric order

ext : ⎕IO

```
      A←5 2 8
      ⍋A
2 1 3
      A[⍋A]
2 5 8
```

⍋ **Character Grade Up**

Return ascending sort order of a character array

res ← *larg* ⍋ *rarg*

larg, *rarg* : any character non-scalar array

res : the indices of *rarg* required to arrange *rarg* in ascending order where *larg* specifies the collating sequences to be used

ext : ⎕IO

```
      'ABC' ⍋ 'CAB'
2 3 1

      A←3 4⍴'FOURFIVESIX '

      A
FOUR
FIVE
SIX

      ⎕AV⍋A
2 1 3

      A[⎕AV⍋A;]
FIVE
FOUR
SIX
```

⍒	**Numeric Grade Down**	**Return descending sort order of a numeric array**

res ← ⍒ *arg*

arg : any numeric non-scalar array

res : the indices of *arg* required to arrange *rarg* in descending numeric order

ext : ⎕IO

```
      A←37 9 18

      ⍒A
1 3 2

      A[⍒A]
37 18 9
```

⍒	**Character Grade Down**	**Return descending sort order of character array**

res ← *larg* ⍒ *rarg*

larg, *rarg* : any character non-scalar array

res : the indices of *rarg* required to arrange *rarg* in descending order where *larg* specifies the collating sequence

ext : ⎕IO

```
      ⎕AV ⍒ 'CAB'
1 3 2
```

Note: `B[A⍋B]←→⊖B[A⍒B]`

⌽	**Reverse**	**Reverse elements of an array**

res ← ⌽ *arg* or *res* ← ⌽[*i*] *arg*

arg : any array

i: non-negative scalar indicating the dimension desired

res : the items in *arg* reversed along the i^{th} dimension default is the last dimension.

ext : ⎕IO

```
      ⌽ 'TOVES'
SEVOT

      A←3 3⍴'ABCDEFGHI'
      A
ABC
DEF
GHI
      ⌽A
CBA
FED
IHG
      ⌽[1]A
GHI
DEF
ABC
```

Note: `⌽[1]A` ←→ `⊖A`

⊖ **Reverse**

Reverse elements of an array

res ← ⊖ *arg* or *res* ← ⊖[*i*] *arg*

arg : any array

i : non-negative scalar indicating the dimension desired

res : the order of the items in *arg* are reversed along the *i*th dimension. The default is the first dimension.

ext : ⎕IO

```
      ⊖ 3 3⍴'ABCD'
CDA
DAB
ABC
```

Note: `⌽A` ←→ `⊖[i]A` where *i*=⍴⍴*A* (the rank or the number of dimensions of *A*).

⌽ **Rotate**

Rotate elements of an array

res ← *larg* ⌽ *rarg* or *res* ← *larg* ⌽[*i*] *rarg*

res : the items in *arg* rotated *larg* places along the *i*th dimension (default is last dimension)

ext : ⎕IO

```
      2 ⌽ 'TODAY'
DAYTO

      A←3 4⍴⍳12

      A
1  2  3  4
5  6  7  8
9 10 11 12

      12 1 ⌽ A
 2  3  4 1
 6  7  8 5
10 11 12 9

      1 2 3 ⌽ [2]A
 2 3  4  1
 7 8  5  6
12 9 10 11
```

Note: `A⌽[1]B ←→ A⊖B`

⊖ **Rotate** **Rotate elements of an array**

res ← *larg* ⊖ *rarg* or *res* ← *larg* ⊖[*i*] *rarg*

larg : integer scalar or vector of length equal to chosen dimension of *rarg*

rarg : any array

i : non-negative scalar indicating the dimension desired

res : the items in *rarg* rotated *larg* places along the *i*th dimension. The default is the first dimension.

```
      1 ⊖ 3 3⍴'ABCD'
DAB
CDA
ABC
```

Note: `A⌽B ←→ A⊖[i]B` where `i=⍴⍴B` (the rank or number of dimensions of `B` , `⎕IO←1`)

⍉ **Transpose**

Reverse axes of an array

res ← ⍉ *arg*

arg : any array

res : *arg* with the dimensions interchanged

```
        A←3 4⍴⍳12
        A
1  2  3  4
5  6  7  8
9 10 11 12

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

        ⍴A
3 4

        ⍴⍉A
4 3
```

⍉ **Dyadic Transpose**

Select and optionally re-order axes of an array

res ← *larg* ⍉ *rarg*

larg : positive integer scalar or vector

rarg : any array

res : *rarg* with the dimensions interchanged in the order specified by *larg*

ext : ⎕IO

```
        A←2 3 4⍴⍳24

        ⍴1 3 2⍉A
2 4 3

        ⍴1 2 3⍉A
2 3 4

        ⍴3 2 1⍉A
4 3 2
```

```
      B←3 4ρ 'ABCDEFGHIJKL'

      1 1 ⍉ B
AFK
```

/ **Replicate (compress)**

Replicate items of an array

res ← *larg* / *rarg* or *res* ← *larg* / [*i*] *rarg*

larg : positive integer scalar or vector of length equal to the chosen dimension

rarg : any array

res : each item of *rarg* is replicated the number of times specified by the corresponding *larg* value

ext : ⎕IO

```
      0 1 2 / 'JMO'
MOO

      A←2 3ρ'ABCDEF'
      A
ABC
DEF

      1 2 3/A
ABBCCC
DEEFFF

      0 1/[1]A
DEF
```

Note: `A/[⎕IO]B` ←→ `A⌿B`

⌿ **Replicate (compress)**

Replicate items of an array

res ← *larg* ⌿ *rarg* or *res* ← *larg* ⌿ [*i*] *rarg*

larg : non-negative integer scalar or vector with length equal to first dimension of *rarg*

rarg : any array

i : non-negative scalar indicating the dimension desired

res : each item of *rarg* is replicated the number of times specified by the corresponding *larg* value along the the chosen dimension of *rarg*.

```
      1 0 2 ⌿ 3 4⍴⍳12
1   2   3   4
9  10  11  12
9  10  11  12
```

Note: *A/B* ←→ *A/[i]B* where
i=⍴⍴*B*(⎕*IO*←1)

**** **Expand**

Expand an array with fill items

res ← *larg* \ *rarg* or *res* ← *larg* \ [*i*] *rarg*

larg : boolean vector whose sum equals the length of the chosen dimension of *rarg*

rarg : any array

i : non-negative scalar indicating the dimension desired

res : the array *rarg* expanded by adding an additional fill item for each corresponding 1 in *larg*

ext : ⎕*IO*

```
      0 0 1 0 1 \ 7 8
0 0 7 0 8

      A←2 3⍴'ABCDEF'
      A
ABC
DEF

      1 0 1 0 1 0\A
A B C
D E F
```

⍀ **Expand**

Expand an array

res ← *larg* ⍀ *rarg* or *res* ← *larg* ⍀ [*i*] *rarg*

larg : Boolean vector whose sum equals the length of the chosen dimension of *rarg*

rarg : any array

res : the array *rarg* expanded by adding additional blanks or zeros for each corresponding 1 in *larg* along the first dimension of *rarg*.

```
      A←2 3 ⍴ 1 2 3 4 5 6
```

```
      A
1 2 3
4 5 6

      1 0 1 ⍀ A
1 2 3
0 0 0
4 5 6
```

Note: `A⍀B  ←→  A\[⎕IO]B`

Type Describers and Modifiers

← **Assign**

Store a value in a variable

name ← *arg*

name : a variable name

arg : any valid expression that returns a value

```
      V ← ⍳5
      V
1 2 3 4 5

      NEWNAME←V+2
      NEWNAME
3 4 5 6 7
```

[ ]← **Index Assignment**

Modify a subset of an array

name[idx_1;idx_2;...]←*arg*

name : a variable name

arg : any valid expression that returns a value

```
      V←2 3⍴⍳6
      V
1 2 3
4 5 6

      V[2;2 3]←7 8
      V
1 2 3
4 7 8
```

∊ Type

The datatype of an array

res← ∊ *arg*

arg: any array

res: zero for each numeric and blank for each character element of *arg*.

```
      ∊ 10 'A' 20 'B'
0   0

      0=∊10 'A' 20 'B'
1 0 1 0
```

⍎ Execute

Execute an APL expression

⍎ *expression* or *res* ← ⍎ *expression*

expression : character scalar or vector

res : the result generated by executing the expression (see Section 1-10 for more details on execute)

```
      ⍎ '2+3'
5
      N←7
      'V',(⍕N),'←10×⍳',⍕N
V7←10×⍳7    (string is displayed)

      V7
VALUE ERROR
      V7
       ∧

      ⍎'V',(⍕N),'←10×⍳'⍕N
             (string is executed)
      V7
10 20 30 40 50 60 70
```

⍕ Format

Convert numeric to character

res ← ⍕ *arg*

arg : any array

res : *arg* converted to character representation

ext : ⎕PP

```
      ⍕ 2 3⍴1 2 3 4 5 6
1 2 3
4 5 6
      ⍴ ⍕ 2 3⍴1 2 3 4 5 6
2 5

      'REDUNDANT'≡⍕'REDUNDANT'
1
      ⍴ ⍕ 1 2 3
5
```

⍕ Pattern Format — **Convert numeric to character**

res ← *pattern* ⍕ *rarg*

pattern : integer scalar or vector of pairs; a single pair is replicated as with scalar extension. The first number of each pair specifies the field width for the column; zero, requests a field large enough to accommodate the largest number. The second number specifies the number of decimal places. If the second number is negative, the result is fomatted in exponential notation. A pair of numbers for each column can specify different formatting for each column. If only one number is specified it is assumed to be the number of decimal places.

rarg : any numeric array

res : a character representation of *arg* formatted as specified by pattern.

```
      1 0 ⍕ 2 3 5
235
      1 ⍕ 2 3 5
 2.0 3.0 5.0

      1 0 4 1 6 2 ⍕ 2 3⍴⍳ 6
1 2.0  3.00
4 5.0  6.00
```

Shape Describers and Modifiers

ρ **Shape** **Return shape of an array**

res ← ρ *arg*

arg : any array

res : a vector containing the length of each dimension of *arg*

```
      ρ 2 3 5
3

      ρ 2 3 5 ρι30
2 3 5

      ρ 99

      ρρ 99
0
```

ρ **Reshape** **Create an array of specific shape**

res ← *larg* ρ *rarg*

larg : numeric scalar or vector

rarg : any array

res : the items of *rarg* selected in order and formed into the new shape specified by *larg*. Some *rarg* elements may be lost (*res* will have fewer items than *rarg*) or duplicated (*res* will have more items than *rarg*) as needed.

```
      3 ρ 99
99 99 99

      2 4 ρ 2 3 5
2 3 5 2
3 5 2 3

      2 3ρ1 1 2ρ7 8
7 8 7
8 7 8
```

, **Ravel** **Change an array into a vector**

res ← , *arg*

arg : any array

res : all the items of *arg* in the same order as *arg*, but as a vector

```
      , 99
99
      ρ , 99
1
      ,2 4ρ2 3 5
2 3 5 2 3 5 2 3
```

, **Catenate** **Join two arrays**

res ← *larg* , *rarg* or *res* ← *larg* , [*i*] *rarg*

larg, *rarg* : any arrays of like type and chosen dimensions (conforming)

i : non-negative scalar indicating the dimension desired

res : the two arrays are joined along the *i*th dimension (default is the last dimension). If *i* is fractional, a new dimension is added.

ext : ⎕IO

```
      2 3 5 , 99
2 3 5 99

      (2 3ρι6),2 2ρ33 333 66666

1     2     3   33  333
4     5     6   66  666

      B←'HOW' ,[.5] 'NOW'
      B
HOW
NOW

      'HOW' ,[1.5] 'NOW'
HN
OO
WW
```

≡ **Depth**

Levels of nesting in an array.

res ← *arg*

arg: any array

i : non-negative scalar indicating the dimension desired

res: the maximum number of times disclose (⊃) must be used to extract a simple scalar

```
      ≡3.3
0
      ≡1 2 3
1
      ≡''
1
      ≡(1 2) 'AB'
2
      ≡⊂⊂⊂⊂⊂12 12
6
      ≡(1 2)(2 3)(3 4)(5 6)
2
```

Operators

/ **Reduction operator**

Apply a specified function across an array, reducing its dimensions

res ← *f* / *arg* or *res* ← *f* / [*i*] *arg*

arg : any array

i: non-negative scalar indicating the dimension desired.

res : the function *f* is applied progressively across the array eliminating the *i*th dimension (the default is the last dimension) in the process

```
res[1]←arg[1;1] f(arg[1;2]..f arg[1;m])
res[2]←arg[2;1] f(arg[2;2]..f arg[2;m])
res[3]←arg[3;1] f(arg[3;2]..f arg[3;m])
...
res[n]←arg[n;1] f(arg[n;2]..f arg[n;m])
```

ext : ⎕IO

```
        +/ 2 3 5
10

        ×/ 2 3⍴1 2 3 4 5 6
6 120

        A←2 3 ⍴ 1 2 3 4 5 6
        A
1 2 3
4 5 6
        ×/[2]A
6 120
        ,/'ABC' 'DEF' 'GHI'
 ABCDEGFGHI
```

⌿ **Reduction Operator**

Apply a function across an array reducing the number of dimemsions

res ← *f* ⌿ *arg* or *res* ← *f* ⌿ [*i*] *arg*

arg : any array valid for *f*

i: non-negative scalar indicating the dimension desired.

res : the function *f* is applied progressively across the array eliminating the i^{th} dimension (the default is the first dimension) in the process

```
res[1]←arg[1;1]f(arg[2;1]...farg[n;1])
res[2]←arg[1;2]f(arg[2;2]...farg[n;2])
res[3]←arg[1;3]f(arg[2;3]...farg[n;3])
...
res[m]←arg[1;m]f(arg[2;m]...farg[n;m])
```

ext : ⎕IO

```
        A←2 3 ⍴ 1 2 3 4 5 6
        A
1 2 3
4 5 6
        ×⌿A
4 10 18
        ×/[1]A
4 10 18
```

Note: For functions other than scalar primitives, the general case of reduction is defined for vectors (recursively) as:

```
res←⊂(⊃arg)f⊃f/1↓arg
```

Example:

```
      ∊/('AE∘')('BUCKWHEAT')
 1 1 0
```

**** **Scan Operator** **Apply successive reductions to an array**

res ← *f* \ *arg* or *res* ← *f* \ [*i*] *arg*

res : the cumulative effect of successive applications of reduction to the i^{th} dimension (the default dimension is the last dimension) of *arg*

```
res[1]←(f/arg[1;1]),(f/arg[1;1 2])..f/arg[1;]
res[2]←(f/arg[2;1]),(f/arg[2;1 2])..f/arg[2;]
...
res[n]←(f/arg[n;1]),(f/arg[n;1 2])..f/arg[n;]
```

ext : ⎕IO

See Section 1-9 for more information.

```
      +\ 2 3 5
2 5 10

      ×\ 2 3ρ1 2 3 4 5 6
1   2   6
4  20 120

      ,\1 2 3
 1  1 2  1 2 3
```

⍀ **Scan Operator** **Apply a successive reduction to an array**

res ← *f* ⍀ *arg* or *res* ← *f* ⍀ [*i*] *arg*

arg : any array valid for *f*

res : the cumulative effect of successive applications of reduction to the i^{th} dimension (the default is the first dimension) of *arg*

```
res[1]←(f⍀arg[1;1]),(f⍀arg[2;1])..f⍀arg[;1]
res[2]←(f⍀arg[1;2]),(f⍀arg[2;2])..f⍀arg[;2]
...
res[m]←(f⍀arg[1;m]),(f⍀arg[2:m])..f⍀arg[1;m]
```

ext: ⎕IO

See Section 1-9 for more details

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

      ×⍀A
1    2    3
4   10   18

      ×\[1]A
1    2    6
4   10   18
```

f* . *g **Inner Product** — **Generalized Matrix Multiplication**

res ← *larg f* . *g rarg*

larg, *rarg* : conforming arrays valid for *f* and *g* where last dimension of *larg* is equal to first dimension of *rarg*

res : the application of function *g* between elements of the last dimension of *larg* and corresponding elements of the first dimension of *rarg* followed by reducing the result using function *f*. The shape of *res* is (¯1↓ρ*larg*),1↓ρ*rarg*. If *larg* is *n* by *k*, and *rarg* is *k* by *m*, then the *res* is:

```
res[1;1]←(f/larg[1;] g rarg[;1])
res[1;2]←(f/larg[1;] g rarg[;2])
   ...
res[2;1]←(f/larg[2;] g rarg[;1])
   ...
res[1;m]←(f/larg[1;] g rarg[;m])
   ...
```

```
res[n;1]←(f/larg[n;] g rarg[;1])
    ...
  res[n;m]←(f/larg[n;] g rarg[;m])
```

Note: For functions other than scalar primitives, inner product is defined only for vectors:

res ← f/ larg g arg

```
      2 3 5 +.× 2 3 5
38

      'SPORT'+.='SHOUT'
3

      (3 3ρ'ABCDEFGHI')∧.='DEF'
0 1 0


      M←2 3ρι6 ◊ N←3 4ρι12
      M+.×N  (matrix multiplication)
38 44  50  56
83 98 113 128

      N∧.=⍉N
1 0 0
0 1 0
0 0 1

      'BUCKWHEAT GROATS'+.∊'AEIOU'
5
```

∘.*f* Outer Product

Apply function between every item of two arrays

res ← larg ∘.f rarg

larg, rarg : any arrays valid for *f*

res : if *f* produces a result, *res* is an array of size ((ρ *larg*), ρ *rarg*) consisting of the result from applying *f* between each combination of *larg* and *rarg* items

If *f* does not produce a result, then ∘*f* will not return a result.

```
      2 3 5 ∘.* 0 1 2 3
1  2   4   8
1  3   9  27
1  5  25 125

      1 2 3 4 5 ∘.⌈ 1 2 3 4 5
1 2 3 4 5
2 2 3 4 5
3 3 3 4 5
4 4 4 4 5
5 5 5 5 5

      'ABC' ∘.= 'ABC'
1 0 0
0 1 0
0 0 1

      'ABC' ∘., '01'
 A0 A1
 B0 B1
 C0 C1
```

f¨ **Each**

Apply a function to each item

res ← *f*¨ *arg* or *larg f*¨*rarg*

rarg : any array with items valid for *f*

larg: any array with items valid, if any, for *f* (optional)

res : the collection of all results (each result is a single nested scalar) from applying *f* to each item of *arg* one at a time

```
      A←1 2 3ρ¨4 5 6
      A
 4  5 5  6 6 6
      ρA
3
```

This example reads the first five components of a file.

```
      ⎕←TN←99,¨ι5
99 1  99 2  99 3  99 4  99 5

      ρTN
5
```

```
      DISPLAY TN
.→---------------------------------.
|.→----..→----..→----..→----..→----.|
||99 1||99 2||99 3||99 4||99 5||
|'~----''~----''~----''~----''~----'|
'∊---------------------------------'

      FILE←⎕FREAD¨TN
```

) COMMANDS

Chapter 2
System Commands

System commands are instructions to the APL system rather than facilities of the APL language interpreter. System commands all begin with a right parenthesis,), to distinguish them from APL language statements. The commands are listed below by type.

- Active Workspace Environment

)FNS	Display function names
)HELP	Display online documentation
)RESET	Clear state indicator
)SI	Display state indicator
)SIC	Clear state indicator
)SINL	Display state indicator showing local names
)SYMBOLS	Display (or change) size of the symbol table
)VARS	Display variable names
)WSID	Display (or change) workspace name

- Workspace and File Management

)CLEAR	Clear active workspace
)DROP	Delete a saved workspace
)FILEHELPER	Help gain access to a file
)FLIB	Display list of component files
)LIB	Display list of all files
)LOAD	Load a saved workspace
)PSAVE	Protected save of a workspace
)SAVE	Save active workspace
)WSLIB	Display list of workspaces
)XLOAD	Load a workspace without executing ⎕*LX*

- Object Manipulation

)	Recall previous APL statements
)COPY	Copy from a saved workspace
)EDIT	Edit an object with full-screen editor
)ERASE	Erase objects in active workspace
)PCOPY	Protected copy from a saved workspace

- Operating Environment

)*CMD*	Execute DCL command
)*LIBS*	Display library to directory correspondence
)*OFF*	End APL session
)*PORTS*	List active users and ports

2-1 System Commands vs. System Functions

Some system functions and system variables provide basically the same capabilities as system commands; however these general differences should be noted:

- System variables can be referenced or assigned; system functions usually have arguments, even if empty. System commands report the current value; those that take an argument reset the value.

- System variables and system functions can be used in an APL statement as part of a defined function; system commands cannot.

- Results from system functions and variables can be captured by assignment to a variable; output from system commands cannot.

2-2 System Command Reference

On the following pages, all of the system commands are listed in alphabetical order and are discussed in detail. Each description contains the system command's name, purpose, syntax, arguments, and effect. One or more examples are also provided for clarity.

Note: Many of the system commands have workspace identifiers or file identifiers as arguments. They are referred to in the syntax as *wsid* and *fileid*, respectively.

A valid identifier consists of a workspace or file name preceded by a directory name. A directory name follows the operating system's convention and may also include a disk or network node identifier. For example, the following are valid workspace or file identifiers.

```
MYWORK
[APL.REL1]DATES
[STUART]TEMPWS
$DISK1:[APL.WS]TEMPWS
LABVAX1::$DD01:[USER1]UTIL
```

If the directory name is omitted, the current default directory is used.

To provide compatibility with other APL*PLUS Systems in a variety of operating systems, this APL*PLUS System also supports library mode. In library mode, a valid identifier consists of the workspace name optionally preceded by a valid library number. For example:

```
TEMPWS
101 DATES
```

The connection between library numbers and operating system directories are made with ⎕*LIBD* and reported with)*LIBS* or ⎕*LIBS*. The system is in directory mode by default unless ⎕*LIBD* is used to assign a library number to a directory. At that point the system is in library mode until all library-to-directory correspondences are removed. ⎕*LIBD* is also used to disolve a library-to-directory assignment.

The APL*PLUS System is in either directory mode or library mode. Some commands that are valid in directory mode will give *INCORRECT COMMAND* messages in library mode and vice versa. The definitive test for library mode is that ⎕*LIBS* has at least one entry:

```
0≠1↑ρ⎕LIBS
```

Workspace and file names themselves (not the directory or library prefix) are limited to a maximum length of eleven characters. Names must be composed entirely of alphabetic letters (A-Z, a-z) and digits (0-9). The first character of the name must be a letter.

Recall Previous APL Statement)

Purpose: Recall previous nonblank APL statement entered in immediate execution mode for re-use after editing.

Syntax:)

Effect: Recalls the previous line and displays it on the screen. The line can then be edited in the same manner as though it had just been typed in. When you press Enter, the current form of the line is executed.

Examples:

```
      1 2 3 + 4 5
LENGTH ERROR
      1 2 3 + 4 5
      ^       ^

      )                  (Recall last line, cursor at end.
      1 2 3 + 4 5_       Type a space and a 6, making it:
      1 2 3 + 4 5 6      and then press Enter.)
5 7 9
```

Purpose: Clear the active workspace.

Syntax:

```
)CLEAR
)CLEAR wssize
```

Argument: *wssize* new workspace size in bytes

wssize must be an integer number greater than 8192, but smaller than the operating system limit.

Effect: Discards the contents of the active workspace and resets the workspace-related system variables to their default values. (See Chapter 3 for the default values).

File ties and session-related system variables are unaffected by the)*CLEAR* operation.

The new size of the workspace may be larger or smaller than the present workspace size. If the workspace size requested exceeds the system configuration limit, the message *INSUFFICIENT SPACE FOR WS* is displayed and the workspace is cleared, but the workspace size is not changed.

The workspace can be cleared under program control by using:

```
      ⎕SA←'CLEAR' ◊ →
```

Example:

```
      )WSID
IS EXAMPLE

      ⎕WSSIZE,⎕WA
150000  116090

      ⎕PW←56
      ⎕IO←0
```

```
        )VARS
A       B       C       DAY     E
F       G       H       I

        )CLEAR 250000
CLEAR WS

        )VARS                        (The variables are deleted.)

        )WSID
IS CLEAR WS                          (EXAMPLE is deleted.)

        ⎕PW                          (Session-related system
56                                   variables remain.)

        ⎕IO                          (Workspace-related system
1                                    variables have been reset.)

        ⎕WSSIZE
250000
```

Purpose: Execute a VMS DCL command.

Syntax:)*CMD*
)*CMD command*

Argument: *command* DCL command to be executed

Effect: Temporarily exits APL (the contents of the workspace are preserved) and allows access to the operating system.

If *command* is not specified, you are in the operating system and may enter as many operating system commands as you wish. `Logoff` returns you to the APL session.

If *command* is specified, APL is again temporarily exited, but this time the operating system command is executed and control immediately passes back to APL.

The APL terminal exit string, if any, is written to the terminal before any non-APL output is produced, and the APL initialization string is written when control returns to APL. Output produced by the operating system is not part of the APL session; it cannot be scrolled back once it has disappeared from the terminal screen, and it will vanish if you press the Refresh key.

⎕*CMD* provides a similar capability and can be used under program control. In addition, ⎕*CMD* can be used to capture the output generated by the DCL command.

Examples:

```
      )CMD                  (Leave APL.)
type log to return to apl
$ show def
   $DISK1:[MYERS]
$ log                       (Return to APL.  Press
                            Refresh key to restore screen.)
      )CMD SHOW TIME
   31-AUG-1987 10:44:36
      2+2                   (Still in APL.)
4
```

Purpose: Copy APL functions and variables from a saved workspace to the active workspace.

Syntax:
```
)COPY wsid
)COPY wsid objlist
```

Arguments:

wsid	workspace identifier (see section 2-2)
objlist	list of functions or variables to be copied

Effect: Copies objects from the saved workspace (*wsid*) into the active workspace and displays a *SAVED* message with the time and date that *wsid* was saved. Identically named objects already in the active workspace will be replaced.

If *objlist* is not specified, all APL variables and functions in the saved workspace are copied into the active workspace.

If copying cannot be completed because an object is too large to fit into the active workspace, a *NOT COPIED:* message is displayed along with the names of the objects that could not be copied. If an object is not found in the specified workspace, a message *NOT FOUND:* is displayed along with the names of the objects that could not be found. In both cases, copying continues with the remaining objects in the list.

If the free space in the active workspace is insufficient for the copy process, one of the following messages may be displayed:

```
WS FULL
WS TOO LARGE
```

If *)COPY* is unable to create a temporary file used in the copy process, one of the following mesages may be displayed:

```
CANNOT CREATE TEMPORARY COPY FILE
ERROR WRITING TEMPORARY COPY FILE
```

Copying a function copies only the source form of the function; any intermediate code normally saved to improve that function's performance is not copied. All ⎕*STOP* and ⎕*TRACE* settings in effect for a copied function are also discarded during the copy process.

⎕*COPY* provides a similar capability and can be used under program control.

Example:

```
      MATRIX
VALUE ERROR
      MATRIX
      ^
      )SI
THREE[7]*

      )COPY OTHERWS ONE TWO THREE FOUR
SAVED 14:19:10   07/02/85
NOT COPIED: TWO
NOT FOUND: FOUR
```

Delete a Saved Workspace)DROP

Purpose: Erase a saved workspace from disk storage.

Syntax: *)DROP wsid*

Argument: *wsid* workspace identifier (see section 2-2)

Effect: Deletes the named workspace (*wsid*) from storage and displays the timestamp of the operation. The active workspace is not affected.

If the workspace does not exist you receive a *WS NOT FOUND* message. If you do not have permission from the operating system to delete this file, a *WS ACCESS ERROR* is displayed. If the library number is undefined (see ⎕*LIBS*), the message *LIBRARY NOT FOUND* is displayed.

The combined use of ⎕*NTIE* and ⎕*NERASE* provide the same capability and can be used under program control.

Examples:

```
      )DROP TEMPWS
12:17:13 05/25/87
```

(In directory mode.)

```
      )DROP [JGW.WSS]OLDWS
10:50:51 05/24/87
```

(In library mode.)

```
      )DROP 101 OLDWS
10:50:51 05/24/87
```

Purpose: Modify or create a function or character variable.

Syntax:)*EDIT object*

Argument: *object* name of the function or character variable to be edited

Effect: Activates the full-screen editor with a new copy of the contents of the named object as an image in the edit ring. If the object exists, it must either be an unlocked function or a simple character variable whose rank is two or less (a vector or matrix). If no object with the specified name exists, it is assumed to be the name of a new function to be created.

The)*EDIT* command can only be used from immediate execution mode. Attempts to use it from ⎕ or function definition mode produces a *NOT IN DEFN OR QUAD* message.

The system function ⎕*EDIT* and special keyboard keystrokes provide a similar capability. ⎕*EDIT* can be used under program control.

For details on the use of the full-screen editor, see Chapter 2 of the *APL*PLUS System User's Manual*.

Examples:

```
)EDIT CUSTOMERLIST

)EDIT PROGRAM
```

Erase Objects in Workspace ***)ERASE***

Purpose: Erase functions and variables from the active workspace.

Syntax: *)ERASE objlist*

Argument: *objlist* list of functions or variables to be erased

Effect: Erases the specified objects from the active workspace. If any of them cannot be erased, the system displays the message *NOT ERASED:* followed by the names of the objects that were not erased.

Functions that are suspended or pending can be erased, but the storage they occupy will not be reclaimed until execution is completed or the stack is cleared (see *)SIC*)

⎕EX and *⎕ERASE* provide a similar capability and can be used under program control.

Examples:

```
      )ERASE JANDATA TRIALFN NOSUCH
NOT ERASED: NOSUCH
```

Purpose: Allow access to a file without adherance to passnumber or access matrix constraints. Useful when you are accidentally locked out of a file.

Syntax: *)FILEHELPER fileid*

Effect: Discards the access matrix for the file specified by *fileid*. ⎕*FHIST* information is updated and you are reflected as the current owner of the file and the last person to change the access matrix. You must be the owner of the file at the VMS level in order to use *)FILEHELPER*.

Examples:

```
      'LOCKEDFILE' ⎕FSTIE 1
FILE ACCESS ERROR

      )FILEHELPER LOCKEDFILE
      'LOCKEDFILE' ⎕FSTIE 1     (Now works.)
```

Purpose: List the names of the APL component files in a library or directory.

Syntax:
```
)FLIB
)FLIB dir
)FLIB lib
```

Arguments:

dir directory to be searched

lib library number of the directory to be searched

Effect: Lists all component files stored in the specified directory or library, even if the user has no access to them. If no library number or directory name is specified, the current working directory is searched.

A directory name (*dir*) can be specified even when the system is in library mode. A library number (*lib*) can only be used when in library mode.

⎕FLIB provides a similar capability and can be used under program control.

Examples:
```
      )FLIB
DATEBOOK   TAXDATA

      ⎕LIBD '213 [APL.WS]'
      )FLIB 213
ORACLE     REPORTS

      )FLIB [APL.REL1]
DATES      INPUT      SERXFER
```

Purpose: List the names of all user-defined functions in the active workspace.

Syntax:

```
)FNS
)FNS start
```

Argument: *start* starting letter or character string

Effect: Displays a list, in alphabetic order, of the user-defined functions in the active workspace. Specifying the optional *start* string begins the list with the functions whose names are alphabetically equal or subsequent to the *start* string.

⎕*NL* and ⎕*IDLIST* provide a similar capability and can be used under program control.

Examples:

```
      )FNS
ADDITEM     PROCESS     TOTALSBYMONTH
CHANGE      RANGECHECK
FILEUPDATE  RESTART

      )FNS P
PROCESS     RESTART
RANGECHECK  TOTALSBYMONTH
```

Purpose: Provide information on the editing commands available in the full-screen editor.

Syntax:)*HELP*

Effect: Displays the contents of the editor help file on the screen. The default help file *HELP.HLP* provided with the system contains a summary of the editing commands available for the terminal chosen when APL was loaded. A different help file may be used, depending on the type of terminal being used.

If the file contains more lines than can be displayed at once, the user can browse through the file by using the U and D keys to move up and down through the file. The help screen remains active until the user presses the Q key.

A different file can be used as the help file if specified by the APL session parameter `help=`. See Chapter 1 in the *APL*PLUS System User's Manual.*

Examples:

)*HELP*	(The system displays the contents of the Help file.)

Purpose: List every workspace and file (including native files) in a library.

Syntax:

```
)LIB
)LIB dir
)LIB lib
```

Arguments:

dir directory to be searched

lib library number where files and workspaces are located

Effect: Lists the files stored in the specified directory. If no directory is specified, the files in the current working directory are listed.

The APL*PLUS System uses extension .WS for saved APL workspaces and .VF for APL component files.

A directory name (*dir*) can be specified even when the system is in library mode. A library number (*libno*) can only be used when in library mode.

⎕LIB provides a similar capability and can be used under program control.

Examples:

```
      )LIB
DATES.WS     TEST.VF
```

(Switch to library mode.)

```
      ⎕LIBD '123 [APL.WS]'
      )LIB 123
JUNK.VF      TEST.WS
```

(Search another directory.)

```
      )LIB [APL.REL1]
ADDSUB.C     DEMO.WS      MOVEFILE.WS
APL          FORMAT.WS    XDEMO.VF
CORE         MAKEFILE
```

Purpose: Display the definitions of the APL libraries in use during this session.

Syntax: *)LIBS*

Effect: Displays the APL library definitions in use during this session. For an explanation of APL libraries, see the *APL *PLUS System User's Manual*. If there is no output from *)LIBS* (indicating that no library numbers are defined), then APL is in directory mode. Library numbers cannot be used when APL is in directory mode.

If any library numbers have been assigned to directory names, then APL is in library mode, and *)LIBS* will list the library-to-directory correspondences. When APL is in library mode, library numbers can be used as a substitute for the directory name.

⎕LIBS provides a similar capability and can be used under program control.

Examples:

```
      )LIBS            (Directory mode; no libraries defined.)

      )LIBS            (Library mode.)
666 [APL.OLD]             11   [STSC.UTIL]
  1 [GROUP.DIR]     12345678   [APL.WS]
```

Purpose: Activate a saved workspace by replacing the current workspace with a copy of a workspace stored on disk.

Syntax: *)LOAD wsid*

Argument: *wsid* workspace identifier

Effect: Replaces the active workspace with a copy of the specified saved workspace (*wsid*) and displays the time and date that the workspace was saved. Once loaded, the latent expression (⎕*LX*) is automatically executed. In a workspace saved with a non-empty state indicator, ⎕*LX* could be a localized latent expression.

The workspace can be in any directory. If a directory is not specified, the current directory is assumed. If the specified workspace is not located in the specified directory, the system displays a *WS NOT FOUND* message. If you do not have read privilege for the file that contains the saved workspace, the system displays a *HOST ACCESS ERROR*. If you load a workspace that was saved by a previous version of APL, you may see the message

```
OBSOLETE WS STRUCTURE UPDATED.
PLEASE RESAVE WS
```

This means that APL has automatically updated the active workspace to accommodate changes to the workspace structure needed for the new version.

If you attempt to load a workspace when the version of APL you are running is older than the version used to save the workspace, the message *INCOMPATIBLE WS* is displayed and the workspace is not loaded.

File ties and session-related system variables are not affected by the *)LOAD* operation.

⎕*LOAD* provides the same capability and can be used under program control.

Examples:

```
      )LOAD [APL.REL1]SCRT     (Directory mode.)
[APL.REL1]SCRT SAVED 14:53:17 05/14/87

      )LOAD STARTWS
STARTWS SAVED 17:20:42 03/17/87
CORPORATE FORECASTING SYSTEM READY
FILES LAST USED ON 8/15/1987 AT 5:35
PM

NEW, MODIFY, DELETE, END [N,M,D,E]:

      ⎕LIBD '123 [APL.WS]'     (Library mode.)
      )LOAD 123 FREQ
123 FREQ SAVED 11:15:59 01/20/59
```

Purpose: End the current APL session.

Syntax: *)OFF*

Effect: Terminates an APL session and returns you to the operating system. The contents of the active workspace are not preserved and any files that were tied are automatically untied.

⎕*SA* provides a similar capability and can be used under program control (⎕*SA*←'*OFF*' ◊ →).

Examples:

```
      )OFF
$
```

Protected Copy)PCOPY

Purpose: Copy APL functions and variables from a saved workspace into the active workspace provided the copy does not replace any objects in the active workspace.

Syntax:

)*PCOPY wsid*
)*PCOPY wsid objlist*

Arguments:

wsid	workspace from which to copy (see section 2-2)
objlist	list of functions or variables to copy

Effect: Copies objects from the saved workspace (*wsid*) into the active workspace and displays a *SAVED* message.

Objects that do not exist in the saved workspace will be listed after a *NOT FOUND:* message. If no objects are specified (*objlist* is omitted), then all variables and functions are copied. Identically named objects already in the active workspace will not be replaced.

Objects that were found but not copied are flagged with a *NOT COPIED* message. This could be due to the workspace containing an existing object by the same name or insufficient space in the workspace to store the object. Copying continues with the remaining objects on the list.

Examples:

```
      )VARS
SIX  THREE

      )PCOPY OTHERWS ONE TWO THREE
SAVED 14:19:10  07/02/85
NOT COPIED: THREE

      )VARS
ONE SIX THREE TWO
```


Purpose: List users signed on to the operating system and the port numbers to which they are attached.

Syntax:)*PORTS*

Effect: Lists the users presently logged on to the VMS operating system and which ports they are using. All active users are listed, whether or not they are presently using APL. The information reported is derived from the VMS command `show users`.

Examples:

```
        )PORTS
STUART:TXA0     SYSTEM          LLG:TXA6
MRVN:TXA3       MLO:TXA4        RIK:TXA5
JGW:TXA9        LINDA:TXA8
```


Protected Save of a Workspace — *)PSAVE*

Purpose: Save a copy of the current workspace on disk under the specified name only if the workspace does not already exist.

Syntax:

```
)PSAVE
)PSAVE wsid
```

Argument: *wsid* — workspace identifier (see section 2-2)

wsid is optional and, if omitted, the name of the active workspace is used.

Effect: Creates a new file on disk containing the active workspace with a name of "*wsid*.WS". If the directory name or library name is included the workspace, the workspace is saved in the specified directory. Otherwise, it is saved in the current directory.

)PSAVE changes the name of the active workspace (*⎕WSID*) to match that of the new saved workspace and updates the values of *⎕WSTS* and *⎕WSOWNER*.

If you attempt to *)PSAVE* a workspace that already exists in the specified library or directory, the system will generate a *WS NAME ERROR* message.

)PSAVE is a more restrictive variant of *)SAVE*.

Example s:

```
      )WSLIB
ACCOUNT        MAILBOX

      )PSAVE PRINTFILE
19.16.34 12/14/86

      )WSLIB
ACCOUNT        MAILBOX      PRINTFILE

      )PSAVE PRINTFILE
WS NAME ERROR
```

Purpose: Clear the state indicator of the active workspace.

Syntax:

```
)RESET
)RESET n
```

Argument: *n* number of suspensions to clear from the state indicator

Effect: Clears the state indicator completely, as opposed to → which clears only the most recent suspension.

If *n* is specified, the state indicator is cleared for *n* suspensions.

⎕*SA* provides a similar capability and can be used under program control (⎕*SA*←'*RESET*').

Examples:

```
          )SI
SUBFN[6]*
STARTUP[2]
SUBFN[5]*
STARTUP[2]
SUBFN[4]*
STARTUP[2]          (Two functions are suspended.)

          → 0

          )SI
SUBFN[5]*
STARTUP[2]
SUBFN[4]*
STARTUP[2]          (One suspension has been cleared.)

          )RESET
          )SI       (All functions have been cleared.)
```

Purpose: Save a copy of the active workspace on disk under the specified name.

Syntax:)*SAVE*
)*SAVE wsid*

Argument: *wsid* workspace identifier (see section 2-2)

Effect: Creates a copy of the active workspace as a file on disk with a name of "*wsid*.WS". If the directory name or library number is also supplied, the file is saved in the specified directory, otherwise it is saved in the current directory.

If no *wsid* is given, the system uses the current active workspace identification (⎕*WSID*), including its library number or directory name. You cannot save a clear workspace; you must first name it.

If *wsid* is different from the workspace name,)*SAVE* changes the name of the workspace (⎕*WSID*) to match that of the saved workspace. If the current workspace name is different from *wsid* and a workspace is already saved on disk with a name of *wsid*, a *NOT SAVED THIS WS IS...* message is displayed. If the save is successful, ⎕*WSID*, ⎕*WSTS*, and ⎕*WSOWNER* are updated to match that of the saved workspace.

For maximum safety during the)*SAVE* operation, the new workspace file is first built as a temporary file *WSSAV.TMPWS.WS*. After the entire workspace is successfully saved in the temporary file, the old workspace file is erased and the temporary file is renamed. If a disk error or system crash occurs during the save process, the original version of the saved workspace remains intact on the disk.

⎕*SAVE* provides a similar capability and can be used under program control.

Examples:

```
      )WSLIB
MAINTGAME    TEST

      )WSID
IS MAINTGAME

      )SAVE
MAINTGAME SAVED 11:03:56 08/05/87

      )SAVE PRODGAMES
PRODGAMES SAVED 11:53:14 08/05/87

      )WSLIB
MAINTGAME    TEST    PRODGAMES
```

Purpose: Display the state indicator of the active workspace, showing which functions are pendent or suspended.

Syntax:)*SI*

Effect: Displays the state indicator starting with the most recent entry. The state indicator includes the status of suspended and pendent functions, executes (⍎), and evaluated input (⎕) calls. The list shows the name of the function and the number of the statement at which execution was suspended.

⎕*SI* provides the same capability under program control.

Example:

```
      )SI
SUBFN[7]*
REPORT[3]
SUBFN[7]*
STARTUP[11]
⍎
```

Purpose: Clear the state indicator of the active workspace.

Syntax:)*SIC*

Effect: Clears the state indicator completely, as opposed to → which clears only the most recent suspension. The system command)*RESET* performs the same function as)*SIC*.

⎕*SA* provides a similar capability and can be used under program control (⎕*SA*←'*RESET*').

Examples:

```
      )SI
SUBFN[6]*
STARTUP[2]
SUBFN[5]*
STARTUP[2]
SUBFN[4]*
STARTUP[2]
```

(There are three suspended function executions.)

```
      →
      )SI
SUBFN[5]*
STARTUP[2]
SUBFN[4]*
STARTUP[2]
```

(Only the topmost suspension, *SUBFN*[6], has been cleared.)

```
      )SIC

      )SI
```

(The state indicator is empty. All suspensions have been cleared.)

Purpose: Display the state indicator of the active workspace, showing which functions are pendent or suspended and which names are localized within each function.

Syntax: *)SINL*

Effect: Displays the same information as *)SI* with the addition of localized names at each level of the stack.

Example:

```
      )COPY UTILITY SUBFN
SI DAMAGE
SAVED 13:03:11   05/10/87

      )SINL
SUBFN[¯1]* L1 L2 X ⎕IO
REPORT[¯1] X Y ⎕ELX
SUBFN[¯1]* L1 L2 X ⎕IO
STARTUP[¯1] RESULT MORE DONE
```


Purpose: Display and optionally change the number of symbol table entries for which there is space reserved in the active workspace.

Syntax:

```
)SYMBOLS
)SYMBOLS n
```

Argument: *n* maximum number of objects allowed in the symbol table

n must be a positive integer greater than 16 or the number of symbols currently in use, whichever is larger.

Effect: Used alone, *)SYMBOLS* reports the maximum number of entries possible in the symbol table of the active workspace and the number in use.

When *n* is provided, *)SYMBOLS* resets the symbol table size to the specified number of entries.

In this APL⋆PLUS System, the symbol table can be enlarged or reduced at any time, not just in a clear workspace. In addition, the system automatically enlarges the symbol table when additional symbol space is required.

⎕SYMB provides the same reporting capability and can be used under program control.

Example:

```
      )CLEAR
CLEAR WS

      )SYMBOLS
IS 500; 0 IN USE

      A←B←C←5
      )SYMBOLS
IS 500; 3 IN USE

      )SYMBOLS 1024
WAS 500
```

Purpose: List the names of the variables in the active workspace.

Syntax:)*VARS*
)*VARS start*

Argument: *start* starting letter or character string

Effect: Displays a list, in alphabetic order, of the variables currently in the local environment of the active workspace. Specifying the optional *start* string begins the list with variables whose names are alphabetically equal or subsequent to the *start* string.

⎕*NL* and ⎕*IDLIST* provide a similar capability and can be used under program control.

Examples:

```
      A←1  ⋄  B←2  ⋄  C←3  ⋄  D←4

      )VARS
A       B       C       D

      )VARS  C
C       D
```

Purpose: Display or reset the name associated with the active workspace.

Syntax:)*WSID*
)*WSID wsid*

Argument: *wsid* workspace identifier (see section 2-2)

Effect: Displays the workspace identification without changing it.

When used with *wsid*,)*WSID* sets the name of the active workspace to the workspace identification provided.

⎕*WSID* provides a similar capability and can be used under program control.

Examples:

```
      )WSID
IS [APL.REL1]MYWS

      )WSID TUESDAY
WAS [APL.REL1]MYWS
```

Display List of Workspaces)WSLIB

Purpose: List the names of the workspaces in a library or directory.

Syntax:

```
)WSLIB
)WSLIB dir
)WSLIB lib
```

Arguments:

dir	directory name
lib	library number

Effect: Lists the workspaces in either the specified directory (*dir*) or library (*lib*) or the user's default directory. The workspaces are listed in alphabetic order. If *lib* or *dir* is omitted, your current default directory is assumed.

⎕WSLIB provides a similar capability and can be used under program control.

Examples:

```
      )WSLIB
GAMES MONTHS UTILITY

      )WSLIB [APL.REL1]
DATES

                                  (Change to library mode.)
      ⎕LIBD '105 [APL.WS]'

      )WSLIB 105
GRAPH PRINT
```


Load a Workspace, Suppressing Execution of the Latent Expression **)XLOAD**

Purpose: Retrieve a saved workspace without executing its latent expression.

Syntax: *)XLOAD wsid*

Argument: *wsid* workspace identifier (see section 2-2)

Effect: Replaces the active workspace with the specified saved workspace and displays the time and date that the workspace was saved, but does not execute the latent expression (*⎕LX*). In a workspace saved with a non-empty state indicator, *⎕LX* could be a localized latent expression.

If the specified workspace is not located, the system displays a *WS NOT FOUND* message.

File ties and session-related system variables are not affected by the *)XLOAD* operation.

The system function *⎕XLOAD* provides the same capability and can be used under program control.

Caution: In this APL*PLUS System, anyone can *)XLOAD* a workspace. Other APL*PLUS Systems and future versions of this system may restrict use of *)XLOAD* to the workspace owner.

Example:

```
      )XLOAD MYWS
SAVED 10:26:22 13/11/86

      ⎕LX
'BOO HOO'                    (Did not execute ⎕LX.)
```

⎕FNS, ⎕VARS

Chapter 3

System Functions, Variables, and Constants

This chapter describes in detail each of the system functions, system variables, and system constants in the APL⋆PLUS System. Their names always begin with a quad (⎕) symbol so that you can easily recognize them (that is, ⎕*LOAD* and ⎕*AV*). System functions, variables, and constants are features that are always available in any workspace. They are listed below by type.

- Workspace Information (active workspace)

⎕*DM*	⎕*WA*
⎕*IDLIST*	⎕*WSID*
⎕*IDLOC*	⎕*WSOWNER*
⎕*IO*	⎕*WSSIZE*
⎕*SI*	⎕*WSTS*
⎕*SYMB*	

- Workspace and File Management

⎕*COPY*	⎕*PSAVE*
⎕*LIB*	⎕*QLOAD*
⎕*LIBD*	⎕*SAVE*
⎕*LIBS*	⎕*WSLIB*
⎕*LOAD*	⎕*XLOAD*
⎕*PCOPY*	

- Function/Object Information and Manipulation

⎕*CR*	⎕*FMT*
⎕*CRL*	⎕*FX*
⎕*CRLPC*	⎕*LOCK*
⎕*DEF*	⎕*MF*
⎕*DEFL*	⎕*NC*
⎕*DR*	⎕*NL*
⎕*EDIT*	⎕*SIZE*
⎕*ERASE*	⎕*SS*
⎕*EX*	⎕*VI*
⎕*FI*	⎕*VR*

- Execution Related

⎕*ALX*	⎕*LC*
⎕*DL*	⎕*LX*
⎕*DM*	⎕*SA*
⎕*ELX*	⎕*SI*
⎕*ERROR*	⎕*STOP*
⎕*IO*	⎕*TRACE*

- Component File Functions

⎕*FAPPEND*	⎕*FRDCI*
⎕*FAVAIL*	⎕*FREAD*
⎕*FCREATE*	⎕*FRENAME*
⎕*FDROP*	⎕*FREPLACE*
⎕*FDUP*	⎕*FRESIZE*
⎕*FERASE*	⎕*FSIZE*
⎕*FHIST*	⎕*FSTAC*
⎕*FHOLD*	⎕*FSTIE*
⎕*FLIB*	⎕*FTIE*
⎕*FNAMES*	⎕*FUNTIE*
⎕*FNUMS*	⎕*LIBD*
⎕*FRDAC*	⎕*LIBS*

- Native File Functions

⎕*LIBS*	⎕*NREAD*
⎕*NAPPEND*	⎕*NRENAME*
⎕*NCREATE*	⎕*NREPLACE*
⎕*NERASE*	⎕*NSIZE*
⎕*NNAMES*	⎕*NSTAC*
⎕*NNUMS*	⎕*NTIE*
⎕*NRDAC*	⎕*NUNTIE*

- Input/Output Management

⎕*ARBIN*	⎕*PP*
⎕*ARBOUT*	⎕*PR*
⎕*CURSOR*	⎕*PW*
⎕*EDIT*	⎕*WGET*
⎕*INKEY*	⎕*WINDOW*
⎕*PFKEY*	⎕*WPUT*

- Interface to Operating System and Non-APL Programs

⎕*CHDIR*	⎕*NA*
⎕*CMD*	⎕*XPn*
⎕*DR*	

- Other Functions

⎕*AI*	⎕*TCESC*
⎕*AV*	⎕*TCFF*
⎕*CT*	⎕*TCLF*
⎕*RL*	⎕*TCNL*
⎕*SYSID*	⎕*TCNUL*
⎕*SYSVER*	⎕*TS*
⎕*TCBEL*	⎕*UL*
⎕*TCBS*	⎕*USERID*
⎕*TCDEL*	

3-1 System Functions

System functions share many of the properties of APL primitive functions:

- They are always available for use in any workspace.
- They can be incorporated into user-defined functions.
- Some have both monadic and dyadic definitions.
- Most return an explicit result that can be used in subsequent operations.

System functions can be niladic (no arguments), monadic (1 argument), dyadic (2 arguments), or ambivalent (1 or 2 arguments). Typically, they.

- provide information about the session, the active workspace, and the objects in it

- retrieve other objects or workspaces

- assist in debugging programs

- produce an effect on or indicate the status of the relevant environment

- provide access to files

- provide an interface to the operating system or non-APL programs.

3-2 System Variables

System variables, a special class of APL variables, are used to manage the interaction between the APL processor and the active workspace.

System variables provide a means of holding information that you, your programs, or the system can always find in any workspace. To you, system variables behave like ordinary variables with some restrictions on domain and shape; to the system, they are a set of parameters controlling the interface with you.

System variables are always available. You cannot erase or copy them. You can reference them, assign values to them, and localize them in functions. They are similar to other localized variables in functions except in the following respects:

- Names of system variables cannot be used as function names or as names of labels, arguments, or the results.
- When a session-related system variable is no longer shadowed (upon returning from function execution or loading a workspace), it takes on the global value associated with the session.
- When execution depends upon a system variable that is localized but has no assigned value, it assumes the value that the variable had at a previous level. This is referred to as pass-through localization.

System variables are classified as session-related or workspace-related. Session-related system variables are not saved with any workspace except where they are localized in pendent or executing functions. No primitive functions depend upon the values of these variables. Workspace-related system variables are stored with the workspace and, therefore, may change value after a)*LOAD* or ⎕*LOAD*.

Session-Related Variables

The default value of session-related system variables is established at the start of each APL session and remains in effect until a new value is assigned. Loading a workspace does not affect the global value of these variables for the session. The value of a localized session variable temporarily supersedes the global value. When a session-related system variable is no longer shadowed (upon return from function execution), the variable takes on the global value associated with the session. The following table summarizes session-related system variables.

Session-Related System Variables

Name	Meaning	Acceptable Values	Default Value
⎕*WINDOW*	Terminal window size and location	Not assignable	0 0 24 80
⎕*PW*	Printing Width	An integer from 30 through 255	80
⎕*CURSOR*	Cursor location	Any screen position	0 0

Workspace-Related Variables

Workspace-related system variables are stored with the workspace and are possibly altered whenever a workspace is loaded. Various primitive functions depend upon the value of one or more of these variables. Workspace-related system variables are summarized in the **Workspace-Related System Variables** table.

The default value of workspace-related system variables is established in a clear workspace and its current value is the value (possibly localized) associated with the active workspace. As with user-defined variables that are localized, when a workspace-related system variable is no longer shadowed (upon return from function execution) it takes on the global value associated with the current state of the workspace.

Workspace-Related System Variables

Name	Meaning	Acceptable Values	Default Value
⎕*ALX*	Attention Latent Expression	Character vector or singleton	'⎕*DM*'
⎕*CT*	Comparison Tolerance	0≤⎕*CT*≤1*E*¯10	1*E*¯13
⎕*ELX*	Error Latent Expression	Character vector or singleton	'⎕*DM*'
⎕*IO*	Index Origin	0 or 1	1
⎕*LX*	Latent Expression	Character vector or singleton	''
⎕*PP*	Printing Precision	Integer from 3 to 18	10
⎕*PR*	Prompt Replacement	Character singleton	' '
⎕*RL*	Random Link	1 to ¯2+2*31	16807
⎕*SA*	Stop Action	'' '*CLEAR*' '*EXIT*' or '*OFF*'	
⎕*WSID* (Clear workspace)	Workspace Identification	Any valid workspace name	''

For example:

```
       ∇ FOO;⎕PW
[1]      ⎕PW←30
[2]      GOO
       ∇

       ∇ GOO;⎕PW
[1]      ⎕PW←77
[2]      ⎕PW
       ∇

         ⎕PW←60

         FOO
77
         ⎕PW
60
```

System Constants

System constants are values that are available in any workspace and do not change within a given APL system. They include the following:

```
⎕AV         ⎕TCESC
⎕FAVAIL     ⎕TCFF
⎕SYSID      ⎕TCLF
⎕TCBEL      ⎕TCNL
⎕TCBS       ⎕TCNUL
⎕TCDEL
```

3-3 Details of System Functions, Variables, and Constants

On the following pages, all of the system functions, variables, and constants are listed in alphabetic order and are discussed in detail. Each description contains the name, syntax, effect, and one or more examples.

Note: Some of the system functions have workspace or file identifiers as arguments. They are referred to as *wsid* and *fileid*, respectively. See section 2-2 for a discussion on identifier names.

Accounting Information ⎕*AI*

Purpose: Return current accounting information.

Syntax: *result* ← ⎕*AI*

Result: *result* is an eight-element numeric vector containing:

[1] Your account number (identification code)

[2] Cumulative amount of CPU time used by this APL session

[3] The elapsed time since the start of the APL session

[4] 0

Although all time is expressed in milliseconds, ⎕*AI* relies on the operating system clock for time measurement. This limits resolution to 1/60th of a second. ⎕*AI*[3] has a one-second resolution.

Caution: ⎕*AI* as described here is specific to this APL*PLUS System. The length and definition of each item of *result* may be different from other APL*PLUS Systems or future releases of this system.

Errors: *WS FULL*

Example: The following expression provides the hours, minutes, seconds, and milliseconds since starting the APL session:

```
      ⎕IO←1
      0 60 60 1000 ⊤ ⎕AI[3]
0 6 24 0
```

Attention Latent Expression ⎕*ALX*

Purpose: Contain the APL expression to be executed in the event of an attention exception.

Syntax:

value ← ⎕*ALX*
⎕*ALX* ← *statement*

Arguments:

value	character vector or singleton
statement	APL expression to replace the current value

Default: '⎕*DM*' in a clear workspace

Effect: When an attention exception occurs during the execution of an APL statement or function, the most local value of the statement stored in ⎕*ALX* is executed (⍎⎕*LX*).

An attention exception occurs whenever execution suspends at the start of a function line because of a weak interrupt. A weak interrupt is usually generated by pressing the Break key once. It is interpreted by the system as a request to stop execution as soon as it has finished executing the current line.

A strong interrupt is usually generated by pressing the Break key twice in rapid succession and is interpreted by the system as a request to stop execution immediately. Note that a strong interrupt does not trigger an attention exception whereas a weak interrupt does.

Errors:

DOMAIN ERROR
RANK ERROR

In addition, any APL error can occur during execution of ⎕*ALX*.

Example: In the first example, ⎕*ALX* is used to protect a critical function from suspension when an interrupt has been signalled by automatically restarting the function. Note that ⎕*LC* has no element corresponding to the ⍎ that would show in the state indicator (see ⎕*SI* or)*SI*) during the execution of the statement ⍎⎕*ALX*.

```
    ∇ SAMPLE1;⎕ALX              ∇ SAMPLE2;⎕ALX
[1] ⎕ALX←'→⎕LC'             [1] ⎕ALX←'⎕ERROR ''ATTN'''
    .                           .
    .                           .
    .                           .
    ∇                           ∇
```

The function *SAMPLE2* uses ⎕*ALX* to pass a special error exception to the calling function so that ⎕*ELX* can be used to handle both errors and attentions. The calling function can then determine that the error resulted from an attention exception and take appropriate action.

Purpose: Perform input and output of data for various physical devices with optional built-in translation.

For example, ⎕*ARBIN* can be used to communicate with a remote computer, a printer, or a native file.

Syntax:

result ← ⎕*ARBIN data*

result ← *out in trans proto wait limit term* ⎕*ARBIN data*

Arguments:

out	output device
in	input device
trans	translation option
proto	protocol option
wait	seconds to wait while collecting the result from *in*
limit	maximum number of bytes of input expected from *in*
term	list of terminator codes
result	data received from the device
data	data sent to the device

The right argument, *data* is either character or numeric data to be sent to the device. If *data* is a matrix or array of higher rank, it is raveled (, *data*) before being transmitted.

The left argument is an integer vector or singleton of transmission options.

out The destination to which the right argument (*data*) is sent, identified by a number. A 1 (the default) specifies the terminal for the APL process; 0 specifies no output. A negative value of *out* indicates the tie number of a native file to which output is appended.

in The source from which data is to be received, identified by a number. A 1 (the default) selects the terminal for the APL process; 0 specifies no input and causes ⎕*ARBIN* to return an empty vector (' ') immediately after *data* has been transmitted even if *wait* or *limit* has not been satisfied. A negative value for *in* indicates the tie number of a native file from which input is read.

trans The way *data* is to be translated before being written and the way *result* is translated after being read.

If *data* is in integer form, it is treated as raw numeric codes and never translated.

If the translation specification is 0 or 1, *data*, in character form, has overstrikes expanded and is translated to typewriter-paired or bit-paired codes, respectively. If the specification is 3, 2 or ¯1, *data* (character form) is transmitted without translation or expansion of overstrikes.

When not explicitly specified, the *trans* is 0 for dyadic use of ⎕*ARBIN* and ¯1 for monadic use.

result is translated in one of four ways.

Trans	Description
¯1	raw untranslated numeric codes, one for each character received.
0	translated according to the APL-ASCII typewriter-pairing overlay. Overstrikes formed with the Backspace character are combined into single APL characters.
1	translated according to the APL-ASCII bit-pairing overlay. Overstrikes formed with the Backspace character are combined into single APL characters.
2	untranslated 7-bit characters. The high (parity) bit is set to 0.
3	untranslated 8-bit characters with the high-order bit preserved.

proto specifies other aspects of the operation.

Proto	Description
0	(Default.)
1	(Reserved.)
2	Echo each character read from inport to outport.

wait The maximum number of elapsed seconds to wait for data (a dead-man timer). If this time limit is reached before any data is received, or since the last data was received or successfully sent, control returns to the calling program. A negative value selects no timeout (an infinite wait). The effect of a zero wait value may be changed in a future release; a zero *limit* should be used when no input is desired.

The default *wait* value, if none is specified, is ¯1.

limit The maximum number of characters of input desired.

Execution of ⎕*ARBIN* terminates when this number of characters has been received. A value of 0 indicates that no response is expected at this time, causing an empty result to be returned immediately.

The default *limit* value, if none is specified, is 400 characters. Since the result of ⎕*ARBIN* always contains a trailing termination code, the minimum value for *limit* is 2.

term A list (possibly empty) of termination codes. Execution of ⎕*ARBIN* terminates when one of these codes is received. For character to numeric equivalents, see Appendix B of the *APL*PLUS System User's Manual.*

The default terminator list, if none is specified, is 13 (the newline character). If ¯1 is supplied as *term*, no termination character is used.

Effect: ⎕*ARBIN* transmits data to the specified port and waits for as long as dictated in the left argument for a response before returning its explicit result. If a wait is dictated, the explicit result is the response received up to termination. If no wait is specified (by a 0 value for *wait* or *limit*), an empty explicit result is returned immediately, allowing local processing to resume at once. Concurrent gathering of a response is still possible during such processing. Note, however, that buffering of input depends upon the capabilities of the operating system version being used. Input may be lost if system buffers overflow.

⎕*ARBIN* can also be used with regular native files, where its overstrike-handling capability is sometimes useful (for example, output to be printed on a printer).

Result: *result* is either a character or numeric vector (depending on translation).

When input is requested, the result of ⎕*ARBIN* is a character or numeric vector as specified in the translation.

If the translation value is 0 or 1, incoming sequences will be resolved as appropriate into overstruck characters, regardless of the order in which they are received. (This process depends on the received characters not causing the cursor to backspace beyond the beginning of the text.) Undefined overstrikes are resolved into an undefined character (⎕*AV*[255+⎕*IO*]).

If the received sequence contains tab characters (ASCII HT), they are represented in *result* as ⎕*AV*(9+⎕*IO*) and are not resolved into spaces. This allows user-programming to determine how they will be treated, even permitting simulation of variable tab positions. Users who do not want to provide interpretation for tab characters can instruct the device not to use them.

The last element of *result* is the terminator character and identifies the cause of ⎕*ARBIN* termination.

¯1↑*result*	**Termination**
⎕*AV*[129+⎕*IO*]	Time out
⎕*AV*[130+⎕*IO*]	Character limit
⎕*AV*[131+⎕*IO*]	Break termination character
⎕*AV*[132+⎕*IO*]	End of file (for native files)
⎕*AV*[*term*]	User supplied termination character

Caution: ⎕*ARBIN* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors: *DOMAIN ERROR*
RANK ERROR
WS FULL

Arbitrary Output to Terminal ⎕*ARBOUT*

Purpose: Permit the transmission of arbitrary transmission codes to a terminal or other remote device.

Syntax: ⎕*ARBOUT codes*

Argument: *codes* set of codes to be transmitted

The argument is an integer array with values from 0 to 255 inclusive. The argument can be of any rank; it is raveled before being displayed. It can also be of any length; it is not limited by the value of ⎕*PW*.

Examples:

```
      ⎕ARBOUT 7
```

(Ring the bell on the terminal.)

Purpose: Return a vector of all possible character values.

Syntax: *result* ← ⎕*AV*

Result: *result* is a 256-element vector of all possible character values.

Caution: Avoid relying heavily on the order in which the character set is mapped onto the elements in ⎕*AV* since this is not the same in all APL⋆PLUS Systems. However, all possible characters are represented somewhere in ⎕*AV* -- even those not available directly from the keyboard. The explicit result can be indexed and the results stored in variables. Throughout this manual, all subscripts into ⎕*AV* are shown in index origin 0.

Note that the entire result of ⎕*AV* cannot be visually displayed since several of its elements are terminal control characters. See Appendix B of the *APL⋆PLUS System User's Manual* for a display of the entire ⎕*AV*. This ⎕*AV* has the same composition as the APL⋆PLUS System for the PC although not all characters can be visually distinguished on most terminals.

Errors: *WS FULL*

Example:

```
      ⎕IO←0
      ⎕AV ⍳'ABC'
65 66 67

      ⎕AV[65 66 67]
ABC

      OLD←'abc'
      ALLCAPS←⎕AV
      IX←(⍳26)+⎕AV⍳'a'
      ALPHA←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      ALLCAPS[IX]←ALPHA
      NEW←ALLCAPS[⎕AV⍳OLD]

      NEW
ABC
```

The last example translates character values. *NEW* becomes a revised version of *OLD* in which all lowercase letters are converted to uppercase letters. A translate table *ALLCAPS* has been formed to do the translation.

Change Working Directory ⎕*CHDIR*

Purpose: Change the default directory.

Syntax: *result* ← ⎕*CHDIR dir*

Argument: *dir* directory name

dir is a character scalar or vector containing a valid directory name or an empty vector (`''`) that returns the name of the current default directory.

Result: *result* is the old current working directory name.

Effect: Changes the working directory to the directory specified. Since the old directory name is returned as *result*, ⎕*CHDIR* `''` can be used to query the current directory.

Errors: *DOMAIN ERROR*
RANK ERROR

Caution: ⎕*CHDIR* as described here is specific to this APL*PLUS System. It may be different or absent in other APL*PLUS Systems.

Examples:

```
      ⎕CHDIR ''                 (Query current directory.)
[STUART]
      ⎕CHDIR '[LINDA.TEST]'     (Change.)
[STUART]
```

Execute DCL Command ⎕*CMD*

Purpose: Execute a VMS DCL command.

Syntax:

```
result ←   ⎕CMD command
result ← 1 ⎕CMD command
         0 ⎕CMD command
```

Argument: *command* DCL command

command is a character vector or singleton containing the DCL command to be executed. It may be empty.

Result: If ⎕*CMD* is used monadically, *result* is an integer scalar containing the return code for the operation. If ⎕*CMD* is used dyadically, *result* is a character vector containing the output generated by executing the DCL command.

Effect: If *command* is empty, APL is temporarily exited, the contents of the workspace are preserved. You are then returned to the operating system and may enter as many operating system commands as you wish. Logoff returns you to the APL session and execution continues with the next statement.

If *command* is a non-empty character vector, APL is termporarily exited, the operating system command is executed, and control immediately passes back to APL.

If ⎕*CMD* is used monadically (only a right argument), the APL terminal exit string, if any, is written to the terminal before any non-APL output is produced and the APL initialization string is written when control returns to APL. Output produced by the system is not part of the session. It cannot be called back once it has disappeared from the session screen and it will vanish if you press the Refresh key.

If ⎕*CMD* is used dyadically with 1 as the left argument, the output is captured and returned as a result. The terminal is not reset. If 0 is the left argument, no result is produced.

Monadic ⎕*CMD* is best used for situations where the execution of the DCL command requires control of the terminal. Dyadic ⎕*CMD* is recommended when the DCL command does not need control of the terminal since all output can be captured by the APL session.

Caution: Do not use dyadic ⎕*CMD* to run an interactive application since you will not receive any output until the program terminates.

⎕*CMD* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors: *DOMAIN ERROR*

Examples:

```
      0⍴⎕CMD ''
$ show time
  5-AUG-1987 14:15:41
$ log
```

(Back in APL.)

```
      RES←1 ⎕CMD 'SHOW DEF'
      ⍴RES
17
      RES

$DISK1:[MYERS]
```

Copy From Saved Workspace ⎕COPY

Purpose: Copy APL functions and variables from a saved workspace into the active workspace.

Syntax:

```
result ← ⎕COPY wsid
result ← objlist ⎕COPY wsid
```

Arguments:

wsid	workspace name (see section 2-2)
objlist	list of functions and variables to copy

objlist can be either a character matrix of object names, one name per row, or a character vector with each name seperated by one or more blanks.

Result: *result* is an integer vector representing the success or failure of ⎕COPY. If *objlist* is specified, *result* contains a response code for each object in *objlist*.

Response Code	**Explanation**
2	A variable was copied successfully.
1	A function was copied successfully.
0	No objects copied; none found with the supplied name.
¯2	The object was too large to copy given the available free workspace.
¯3	The name is defined as a label and cannot be changed.
¯4	There is insufficient space in the symbol table to copy this object.
¯6	The amount of workspace available is too small to perform the copy.

If ⎕COPY is used without specifying *objlist*, then *result* is empty if all objects of *wsid* were copied successfully. If one or more objects to be copied from *wsid* are suspended or pendent functions in the current workspace, *result* is a numeric vector containing an appropriate response code for each object that is not copied. If an unanticipated error occurs, no result is returned.

Effect: Copies objects from the specified workspace (*wsid*) into the local environment of the active workspace replacing any objects by the same name. See description of ⎕*PCOPY* for a way to prevent replacement of existing objects.

Copying a function only copies its source form; all compiled code is discarded and ⎕*STOP* and ⎕*TRACE* settings are cleared in the active workspace.

Errors:

```
DOMAIN ERROR
INSUFFICIENT MEMORY
LENGTH ERROR
RANK ERROR
WS ARGUMENT
WS DAMAGED
WS FULL
WS NOT COMPATIBLE
WS NOT FOUND
```

Example:

```
      )VARS
MT

      MT
1 2
3 4

      )SI
SUSPENDED[3]*

      'MT XXX DATA SUSPENDED' ⎕COPY 'WS3'
2 0 2 ¯3

      )VARS                         (Value of MT has changed.)
DATA MT

      MT
CAT
DOG
RAT
```

Purpose: Return the canonical representation of a function.

Syntax: *result* ← ⎕*CR* *fnname*

Argument: *fnname* function name

fnname is a character singleton or vector containing the name of a function.

Result: *result* is a character matrix containing the canonical representation of the most local definition of the function. Each line of the function (including the header) is left-justified and all lines (except the longest line) are padded on the right with blanks.

If *fnname* is not the name of an unlocked function, *result* is an empty matrix (shape 0 0).

The result of ⎕*CR* can be assigned to a variable and used as the argument to ⎕*DEF* or ⎕*FX* to redefine the original function.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Example:

```
      ∇  TRI N;A
[1]      ⎕←A←,1
[2]     L1:→(N<ρA)ρ0 ◊ ⎕←A←(0,A)+A,0
[3]      →L1
      ∇

         ρQ←⎕CR 'TRI'
4 25

         Q
TRI N;A
⎕←A←,1
L1:→(N<ρA)ρ0 ◊ ⎕←A←(0,A)+A,0
L1

         ⎕FX Q
TRI
```

Canonical Representation of a Single Function Line ⎕*CRL*

Purpose: Return a character vector containing the canonical representation of a single line of a function.

Syntax: *result* ← ⎕*CRL* '*fnname*[*n*]'

Arguments: *fnname* function name
n line number

The argument to ⎕*CRL* is a character singleton or vector. *fnname* is the name of a valid function and *n* is a non-negative integer representing a line number in the function.

Result: *result* is the canonical representation of line *n* of function *fnname* with a length matching that of line *n* (generally shorter than the width of ⎕*CR* '*fnname*'). If *n* is zero, the result is the header of the function.

If *fnname* is a locked function or if *n* is greater than the number of lines in the function, the result is an empty vector.

result is also an empty vector if the argument is ill-formed or the function does not exist.

If *n* is not given, the result of ⎕*CRL* is 1⍴' '.

Errors: *DOMAIN ERROR*
RANK ERROR
WS FULL

Examples:

```
      ∇  FOO
[1]      ⎕←'THIS IS A TEST'
[2]      A←⍳12
[3]      ⎕←A×3
      ∇
```

```
      ⎕CRL 'FOO'

      ρ⎕CRL 'FOO'
0

      ⎕CRL 'FOO[2]'
A←⍳12

      DD←⎕CRL 'FOO[1]'
      DD
⎕←'THIS IS A TEST'

      ⍎DD
THIS IS A TEST
```

Purpose: Retrieve the public comment from a single line of a function. A public comment begins with ⍝∇ and can occur after executable code on a given line. ⎕*CRLPC* also operates on locked functions, allowing even locked functions to have imbedded documentation retrievable by the user.

Syntax: *result* ← ⎕*CRLPC* '*fnname*[*n*]'

Arguments:

fnname	function name
n	line number

Result: *result* is the public comment for line *n* of function *fnname*.

If line *n* has no public comment or if *n* is greater than the number of lines in the function, *result* is an empty vector. It is also an empty vector if the argument is ill-formed or the function does not exist.

Errors:

DOMAIN ERROR
RANK ERROR
WS FULL

Example: ⎕*CRLPC* can be used to identify different versions of the same locked function; the version number can be documented in a public comment.

```
      ⎕CRLPC 'LOCKEDFN[1]'
⍝∇ VERSION 4 REVISED 10/15/86 BY SAM
```

Comparison Tolerance ⎕*CT*

Purpose: Specify the maximum relative difference allowed between two numbers for them to be considered equal.

Syntax:

```
value ← ⎕CT
⎕CT ← value
```

Domain: *value* is any single numeric value between 0 and `1E¯10`. In a clear workspace, the default value is `1E¯13`. `⎕CT`, when referenced, is always a numeric scalar.

Effect: Overcomes the problems of inexact internal representation and cumulative rounding errors that are inherent in computer arithmetic on noninteger values. Comparison tolerance is a means of ignoring small differences between two numbers that are likely to come from inexact representation or rounding.

Two numbers are considered equal if their relative difference is less than or equal to `⎕CT`. Other comparisons are derived from that property. This means that *A* and *B* are considered equal if:

```
(|A-B)≤⎕CT×(|A)⌈|B.
```

If `⎕CT` is 0, all comparisons are exact. Furthermore, all comparisons with the number 0 are exact and are independent of `⎕CT`. Setting `⎕CT` to `0` may produce counter-intuitive results from floating-point calculations on real numbers due to the way numbers are stored internally (see **Caution:** below).

The value of `⎕CT` is used when computing the result of any of the following primitive functions using floating-point data:

- floor (`⌊`)
- ceiling (`⌈`)
- residue (`|`)
- match (`≡`)
- membership (`∊`)
- index of (`⍳`)
- numeric relation (`> ≥ = ≤ <`)

Caution: Only in special cases should ⎕*CT* be set to zero. The examples presented below illustrate the shortcomings of exact comparisions when performing arithmetic on non-integer numbers that experience rounding.

The following chart shows how the results of some simple expressions depend upon the value of ⎕*CT*.

Effect of ⎕*CT* on Numeric Operations

```
EPS ← 1E¯15
A   ← 0 0 1 1
B   ← (0+EPS), (0-EPS),(1+EPS),(1-EPS)

⎕CT ← 0          ⌊B   ⟷  0  ¯1   1   0
⎕CT ← 10×EPS     ⌊B   ⟷  0   0   1   1

⎕CT ← 0          ⌈B   ⟷  1   0   2   1
⎕CT ← 10×EPS     ⌈B   ⟷  0   0   1   1

⎕CT ← 0          A=B  ⟷  0   0   0   0
⎕CT ← 10×EPS     A=B  ⟷  0   0   1   1

⎕CT ← 0          A<B  ⟷  1   0   1   0
⎕CT ← 10×EPS     A<B  ⟷  1   0   0   0

⎕CT ← 0          A⍳B  ⟷  5   5   5   5
⎕CT ← 10×EPS     A⍳B  ⟷  5   5   3   3

⎕CT ← 0          A∊B  ⟷  0   0   0   0
⎕CT ← 10×EPS     A∊B  ⟷  0   0   1   1
```

Errors:

```
DOMAIN ERROR
RANK ERROR
```

Examples:

```
      )WSID
IS CLEAR WS

      ⎕CT
1.0E¯13

      3=3+.000000000001
0
      ⎕CT←.00000000001
      3=3+.000000000001
1
```

Purpose: Query or set the cursor location on the screen.

Syntax:

```
pair ← ⎕CURSOR
⎕CURSOR ← pair
```

Domain: Integer vector (2 elements) containing the row and column of the cursor position relative to the upper-left corner of the window (in origin 0). The default value is 0 0 and is reset each time the window is cleared.

Effect: The value of ⎕*CURSOR* is the cursor location at the time the statement is executed (not its position before the line was executed, which may be the line above).

Assigning a new value to ⎕*CURSOR* moves the cursor to the new position. *pair* must be a valid cursor position or a *DOMAIN ERROR* is produced.

Caution: ⎕*CURSOR* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

DOMAIN ERROR
LENGTH ERROR
RANK ERROR

Examples:

```
      ⎕CURSOR
22 0
```

(The cursor was on line 22 in column 0 of the current window when ⎕*CURSOR* was executed.)

```
      ⎕CURSOR ← 0 0 ◊ 'A'
```

(Move the cursor to the upper-left corner of the current window and display an "*A*".)

Purpose: Define a function from a character representation.

Syntax: *result* ← ⎕*DEF fnrep*

Argument: *fnrep* character representation of a function

If *fnrep* is a character vector whose first non-blank character is ∇ or ⍫, it is assumed to represent a function in ⎕*VR* form. Otherwise, a character vector will be taken to be a vector version of a function in ⎕*CR* form (that is, without ∇'s and line numbers). If *fnrep* is a character matrix, the function is assumed to be in ⎕*CR* form. *fnrep* may contain superfluous blanks in the same way that function definition (∇-editor or)*EDIT*) allows them.

Result: If the function definition is successful, *result* is the name of the defined function.

If the function definition is not successful, *result* is a two-element numeric vector containing information about the error (see **Errors:** below).

Effect: Defines a function of the appropriate name in the active workspace unless an error condition occurs. The amount of available workspace area and the number of symbols may change. If *fnrep* contains a leading or trailing ⍫, the function will be locked after it is defined.

If the name of the function defined corresponds to a local identifier in a currently executing, pendent, or suspended function, the newly defined function is local to that function and is erased when the function in which it is localized completes execution.

If the name of the function defined corresponds to the name of an existing function, the existing function is replaced and any ⎕*STOP* or ⎕*TRACE* settings in the function are removed.

Example:

```
      M
TRI N;A
⎕←A←,1
L1:→(N<ρA)ρ0 ◊ ⎕←(0,A)+A,0 ◊ →L1

      M←⎕CR 'TRI'
      M[1;]←(1↓ρM)↑'TRIANGLE N;A'

      ⎕DEF M
TRIANGLE
```

Notes: *⎕DEF* and *⎕FX* provide similar capabilities. *⎕DEF* is a more powerful and general case of *⎕FX*. The differences are outlined below:

- *⎕DEF* accepts both canonical (matrix) and visual (vector) representations of a function; *⎕FX* accepts only the canonical representation.
- *⎕DEF* can create a function as a locked function; *⎕FX* cannot.
- *⎕DEF* indicates both the cause and the location of an error; *⎕FX* indicates only the location.
- *⎕DEF* indicates the *SYMBOL TABLE FULL* or *WS FULL* conditions via error codes without halting execution. *⎕FX* halts execution.

Errors: If the system recognizes an error condition during analysis of a character vector or matrix argument, the function is not defined, but no explicit error is reported. Instead, the result is a two-element integer vector containing information about the error. The first element is the type of error that occurred; the second element indicates the row of the function representation where the error begins. The index returned depends on the current setting of *⎕IO*.

The following error types are indicated by the first element of the result:

⎕*DEF* Error Codes

Code	Explanation
1	*WS FULL*; the function definition requires more workspace storage than is available.
2	*DEFN ERROR* • the function or header is ill-formed • the function name is already in use as a variable or label • the function is executing, pendent, suspended, or waiting • the first character in a line of code is a right parenthesis, right bracket, or left bracket (not including line numbers)
3	Reserved.
4	*SYMBOL TABLE FULL*; creating the function requires more symbol table entries than are available in the active workspace.
5-9	Reserved.

Purpose: Edit a single line of the most local definition of an unlocked function.

Syntax:

```
result ← ⎕DEFL 'fnname[n]line'
result ← ⎕DEFL 'fnname[~n]'
```

Arguments:

fnname	function name
`[n]`	line number
line	text of the line to be inserted or replaced
`[~n]`	line number or numbers to be deleted

The argument must be a character scalar or vector.

To **replace** an existing line in the function named *fnname*, specify the line number *n* in brackets followed by the replacement text (*line*).

To **insert** a new line into the function named *fnname*, specify *n* as a decimal fraction between two existing lines, such as `[3.5]`. In such a case, `⎕DEFL` will insert *line* between lines 3 and 4. If *n* is greater than the number of lines in the function, *line* will be inserted at the end of the function.

To **delete** a line from the function named *fnname*, specify a tilde (~) before *n* and omit *line*. Multiple lines can be deleted by specifying *n* as a vector, as in `[~3 4 5]`.

Result: If the operation is successful, *result* is a character vector containing the name of the function. If the name of the function changes as a result of replacing line 0 of the function, the result is the name of the new function.

If the operation is not successful, *result* is a numeric scalar containing information about the error (see **Errors:** below).

Effect: Inserts or deletes the lines as requested by the syntax. All lines following the point of insertion or deletion are automatically renumbered.

Note that the form of the argument to ⎕*DEFL* is the same for insertion and replacement. The effect depends upon the value of *n* relative to the line numbers of the function. In this sense, the behavior of ⎕*DEFL* is similar to other function editing capabilities in the APL*PLUS System.

Errors: If an error condition occurs during analysis of argument values by the system, no explicit error is reported. Instead, the result is an integer scalar indicating the type of error. Note that if one of the listed errors occurs, the function is not changed.

⎕*DEFL* Error Codes

Code	Explanation
1	*WS FULL*; the function definition requires more workspace storage than is available.
2	*DEFN ERROR* • the argument is ill-formed • *fnname* is the name of a locked, suspended, pendent, or non-existent function • the new name of the function is currently defined or you tried to delete line 0 • the first nonblank character in *line* is a) or] • *n* is negative or greater than 9999.9999
3	Reserved.
4	*SYMBOL TABLE FULL*; creating the function requires more symbol table entries than are available in the active workspace.
5-9	Reserved.

Example:

```
      ⎕VR 'TRI'
    ∇ TRI N;A
[1]   ⎕←A←,1
[2]  L1:→(N<ρA)ρ0 ◊ ⎕←A←(0,A)+A,0 ◊ →L1
    ∇
      ⎕DEFL 'TRI[1] A←,1'
TRI
      ⎕VR 'TRI'
    ∇ TRI N;A
[1]   A←,1
[2]  L1→(N<ρA)ρ0 ◊ ⎕←A←(0,A)+A,0 ◊ →L1
    ∇
```

Purpose: Delay execution.

Syntax: *result* ← ⎕*DL seconds*

Argument: *seconds* length of the delay in seconds

seconds is a positive numeric singleton (possibly fractional).

Result: *result* is the actual delay in seconds; it may vary each time ⎕*DL* is used.

Effect: Using the system clock, ⎕*DL* delays execution for the time requested. The delay can be aborted by a weak interrupt in which case *result* may be substantially less than *seconds.*

Errors:

```
DOMAIN ERROR
LENGTH ERROR
WS FULL
```

Example:

```
      ⎕DL 5
5
```

Diagnostic Message ⎕*DM*

Purpose: Return the last diagnostic message recorded in the workspace. A diagnostic message is produced for any event that halts execution such as an APL error or a user interrupt.

Syntax: *result* ← ⎕*DM*

Result: *result* is a character vector containing the diagnostic message associated with the last error or interrupt that occurred.

Effect: Displays the diagnostic message associated with the last weak interrupt, strong interrupt, or trapped error that occurred in the workspace. Except for *INTERRUPT*, ⎕*DM* does not reflect the diagnostic message displayed after an untrapped error or attention. For more information on exceptions, see ⎕*ALX*, ⎕*ELX*, and ⎕*ERROR* in this chapter.

The diagnostic message reported by ⎕*DM* is saved when the workspace is saved.

If there is not enough workspace storage available when an error or attention occurs, the system displays *NO SPACE FOR* ⎕*DM* followed by the diagnostic message. ⎕*DM* is empty after a *NO SPACE FOR* ⎕*DM* error.

Caution: System-produced diagnostic messages may be altered or extended in the future. Applications that analyze the result of ⎕*DM* should, therefore, be designed to allow easy modification. One such technique is to use the same function for analyzing the diagnostic message throughout an application.

Examples:

```
      )CLEAR              (⎕DM is empty in a clear workspace.)
CLEAR WS

      ρ⎕DM
0
```

```
      3+A               (An APL error is generated; the normal
VALUE ERROR             diagnostic message displays since
      3+A               ⎕ELX←'⎕DM'.)
        ^

      ⍴⎕DM ⋄ ⎕DM        (⎕DM now returns the diagnostic
32                      message associated with the last error
VALUE ERROR             exception.)
      3+A
        ^

      )SAVE TEMP        (The workspace is saved, then cleared.)
TEMP SAVED 7:19:00 05/27/87

      )CLEAR
CLEAR WS

      ⍴⎕DM
0

      )LOAD TEMP
TEMP SAVED 7:19:00 05/27/87

      ⎕DM               (⎕DM was saved with the workspace.)
VALUE ERROR
      3+A
        ^

      ⎕ELX ← ''         (⎕ELX is set to do nothing; no error
      5÷0               message is displayed on obvious APL
                        errors.)
      'A' + 1
      2 3 × 9 10 11

      ⎕DM               (Last error message is in ⎕DM.)
LENGTH ERROR
      2 3 × 9 10 11
      ^   ^
```

```
      ⎕ELX ← '⎕ERROR '' '''
                    (Result is even less revealing; ⎕DM is
                    reset, removing the error message.)

      2 3 × 9 10 11 (Same statement causes error but an
                     empty line displays.)
      ⎕DM

                    (⎕DM contains a single space.)
      ⎕ELX ← '⎕DM' (After experimenting, reset ⎕ELX.)

      2 3 × 9 10 11
LENGTH ERROR
      2 3 × 9 10 11
      ^   ^
```

Purpose: Report the internal datatype of the argument.

Syntax: *result* ← ⎕*DR data*

Argument: *data* any APL array

Result: *result* is the datatype code for *data*. The last digit of the result (10 | *result*) indicates the data format used while the other digits (⌊*result*÷10) indicate the number of bits per element with which the data is represented. The following are the datatype codes for this APL⋆PLUS System:

Code	Datatype	
11	Boolean	(1 bit per element)
82	character	(8 bits per element)
323	integer	(32 bits per element)
644	floating point	(64-bit VAX format)
326	nested	(32-bit pointer)
807	heterogeneous	(10-byte structure)

Caution: More datatype codes may be added in future releases. The datatype codes specified here are not necessarily the same datatype codes on other APL⋆PLUS Systems on other computers.

⎕*DR* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples:

```
      ⎕DR 'X'
82

      ⎕DR 'A',1
807

      ⎕DR ⊂⍳5
326

      ⎕DR¨5,(C⊂⍳5),'C',(1∧1)
323 326 82 11
```

Edit an Image of Named Object from Active Workspace ⎕*EDIT*

Purpose: Edit a character vector, matrix, or function.

Syntax: ⎕*EDIT object*

Argument: *object* name of the object to be edited

object is a character vector, one-row matrix, or scalar containing the name of the object to be edited.

Effect: A new edit session is created in the session manager and the function or variable specified by *object* is copied into it. The session name is updated to reflect the object's name and the session manager is initialized to edit the copy of the object. (The details on editing operations are described in Chapter 2 of the *APL⋆PLUS System User's Manual.*)

Upon return to your APL session, the cursor is restored to the same position it was in before the statement was executed.

If the variable named in the argument contains numeric or nested data or the argument is of rank greater than 2, a *NONCE ERROR* is produced. If the object does not exist, a new object is created and given the specified name.

Errors:

DOMAIN ERROR
NONCE ERROR
SYMBOL TABLE FULL
WS FULL

Caution: ⎕*EDIT* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples:

```
⎕EDIT 'CUSTOMERLIST'

⎕EDIT 'PROGRAM'
```

Purpose: Contain the APL expression to be executed in the event of an error exception.

Syntax:

statement ← ⎕*ELX*
⎕*ELX* ← *statement*

Domain: Character vector or singleton containing an APL expression. The default value of ⎕*EX* is '⎕*DM*' in a clear workspace.

Effect: Whenever a trapped error (see definition below) occurs during execution of an APL expression or function, the statement stored in the most local value of ⎕*ELX* is executed. Thus, if ⎕*ELX* has its default value ('⎕*DM*') when an error occurs, the system simply displays the diagnostic message (see ⎕*DM*).

If an error occurs during execution of the actual statement in ⎕*ELX*, the system displays the diagnostic message and returns to immediate execution input. If, however, the error handler calls a function, errors signalled within that function trigger execution of ⎕*ELX*.

If an error occurs while the system is evaluating ⎕ input, the diagnostic message associated with the error is displayed and the user is prompted again for input; ⎕*DM* is not changed and ⎕*ELX* is not executed. Note that if a function call is entered in ⎕ input, errors occurring within the called function do trigger execution of ⎕*ELX*.

APL Errors Handled by ⎕*ELX*:

The following errors are trapped (trigger execution of ⎕*ELX*) except when caused by a system command. Any error exceptions signalled by ⎕*ERROR* are also trapped.

AXIS ERROR
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE DAMAGED

FILE FULL
FILE INDEX ERROR
FILE NAME ERROR
FILE NOT FOUND
FILE SIZE ERROR
FILE TIE ERROR
FILE TIE QUOTA EXCEEDED
FILE TIED
FORMAT ERROR
HOST ACCESS ERROR
INDEX ERROR
LENGTH ERROR
LIBRARY NOT FOUND
LIMIT ERROR
NONCE ERROR
RANK ERROR
SYMBOL TABLE FULL
SYNTAX ERROR
VALUE ERROR
WS ARGUMENT ERROR
WS FULL
WS NOT COMPATIBLE
WS NOT FOUND
WS TOO LARGE

Errors that are **not** trapped are:

- input errors (including errors in expressions evaluated for ⎕ input)
- errors resulting from system commands
- errors signaled by an ill-formed statement in ⎕*ELX*
- system errors (internal errors in the APL⋆PLUS System itself)

Errors: *DOMAIN ERROR*
RANK ERROR

In addition, any APL error can occur during the execution of ⎕*ELX*.

Examples: In the function *SAMPLE*1, ⎕*ELX* is used to branch to the error-processing part of the function if an error occurs.

```
     ∇ SAMPLE1;⎕ELX
[1]    ⎕ELX←' →ERR '
.
.
.
[n]    ERR:
.
.
.
     ∇
```

This next function uses ⎕*ELX* to invoke an error in the function that called it.

```
     ∇ SAMPLE2;⎕ELX
[1]
⎕ELX←'⎕ERROR((⎕DM⍳⎕TCNL)-⎕IO)↑⎕DM'
.
.
     ∇
```

Erase Objects ⎕*ERASE*

Purpose: Erase, if possible, objects in the workspace while under program control.

Syntax: *result* ← ⎕*ERASE objlist*

Argument: *objlist* list of function or variable names

objlist can be a character vector containing one or more object names separated by one or more blanks, or it can be a character matrix with one identifier in each row.

Result: *result* is a character matrix with each row containing the name of an object that was not erased. Objects that are undefined are **not** included in *result*.

If all objects in *objlist* are erased, *result* is an empty matrix.

Effect: Erases objects specified in *objlist*. ⎕*ERASE* does not erase the definitions of identifiers representing labels, system functions, or system variables. An object might not be erased because the name is ill-formed or because it is a suspended or executing function.

In this version of the APL⋆PLUS System, ⎕*ERASE* can erase a suspended or exectuing function. In fact, a function can even erase itself. The name association with the function is broken, but the executing function does not actually disappear until it completes execution or is cleared from the)*SI* stack.

Note: ⎕*ERASE* and ⎕*EX* provide similar capabilities. For maximum portability to other APL Systems, use ⎕*EX* rather than ⎕*ERASE*.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Example:

```
      ρ⎕←⎕ERASE 'MYPROGRAM'
0 0
      ρ⎕VR 'MYPROGRAM'
0
```

Purpose: Generate a user-defined error exception.

Syntax: ⎕*ERROR message*

Argument: *message* diagnostic message

message is a character singleton or vector containing the first line of the diagnostic message associated with the resulting error exception.

Effect: ⎕*ERROR* provides two facilities:

- the ability of a function to signal an exception to the program from which it was called
- the ability to signal user-defined error exceptions.

When ⎕*ERROR* is executed, the state indicator stack is returned to the environment from which the function executing ⎕*ERROR* was called. If the state indicator is empty or contains only one function when ⎕*ERROR* is executed, the error exception is signalled in the global environment.

If *message* is empty (' '), no exception is signaled, which permits conditional signaling of error exceptions with a statement of the form ⎕*ERROR condition* / '*message*'.

Errors:

```
DOMAIN ERROR
NO SPACE FOR ⎕DM
RANK ERROR
WS FULL
```

Examples: In the function *SQRT* below, ⎕*ERROR* signals an error in the environment from which *SQRT* is called instead of within *SQRT* itself.

```
     ∇   R←SQRT A;⎕ELX
[1]      ⎕ELX←'⎕ERROR((⎕DM⍳⎕TCNL)-⎕IO)↑⎕DM'
[2]      R←A*0.5
     ∇

         SQRT ¯1
DOMAIN ERROR
         SQRT ¯1
         ^
```

In the next example, *SQRT* is modified to detect a negative argument and generate an error message that is more informative than the *DOMAIN ERROR* report normally produced by the system.

```
     ∇   R←SQRT A;⎕ELX
[1]      ⎕ELX←'⎕ERROR ((⎕DM⍳⎕TCNL)-⎕IO)↑⎕DM'
[2]      ⎕ERROR (∨/,A<0)/'ARGUMENT NEGATIVE'
[3]      R←A*0.5
     ∇

         SQRT ¯1
ARGUMENT NEGATIVE
         SQRT ¯1
         ^
```

If *SQRT* is called from another function and a negative argument is supplied to *SQRT*, an error is signalled in the calling function.

```
     ∇   R←M RELMASS V;C
[1]      ⍝ COMPUTES RELATIVISTIC MASS
[2]      ⍝ OF A MOVING OBJECT
[3]      ⍝ M ←→ REST MASS; V ←→ VELOCITY
[4]      ⍝ C ←→ SPEED OF LIGHT IN METERS/SEC
[5]      C←300000000
[6]      R←M÷SQRT 1-(V*2)÷C*2
     ∇

         1 RELMASS 2.9E8
3.905667329

         1 RELMASS 3.5E8           (Uses a velocity greater
ARGUMENT NEGATIVE                   than the speed of light.)
RELMASS[5] R←M÷SQRT 1-(V*2)÷C*2
                ^
```

The following technique can be used to clear the result of ⎕*DM*, provided the state indicator is clear and ⎕*ELX* does not call ⎕*ERROR*.

```
        ⎕ERROR ' '
```

Since ⎕*ERROR* reduces the state indicator stack by one function call, it can be used to move one level up in the state indicator for debugging purposes; for example:

```
        DRIVER
LENGTH ERROR
SUBROUTINE[1]  Z←A+B×0,1↓A
                        ∧

        )SI
SUBROUTINE[1]  *
PROCESS[7]
MAINFN[3]
DRIVER[5]

        ⎕ERROR 'POP'
POP
PROCESS[7]  SUBROUTINE
            ∧

        )SI
PROCESS[7]  *
MAINFN[3]
DRIVER[5]
```

The argument (*B*) to *SUBROUTINE* can now be corrected and execution can resume.

```
B←(ρA)↑B  ◊   →⎕LC
```

Erase Objects ⎕*EX*

Purpose: Erase, if possible, the most local version of one or more objects in the active workspace while under program control.

Syntax: *result* ← ⎕*EX objlist*

Argument: *objlist* list of zero or more functions or variable names

objlist can be a character vector containing one or more object names separated by one or more blanks, or it can be a character matrix with one identifier in each row.

If ⎕*EX* produces a *WS FULL* or *DOMAIN ERROR*, nothing has been erased.

Result: *result* is a Boolean vector with one element for each name provided in *objlist*. The result is 1 if the object was erased or undefined; the result is 0 if the object was not erased. An object might not be erased because the name is ill-formed or because it is a suspended or executing function.

Effect: Erases objects specified in *objlist*. ⎕*EX* does not erase an identifier if it is a label, system function, or system variable.

Caution: Some APL systems may restrict *objlist* to a character matrix.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
      ⎕EX 'TRI'
1

      TRI
VALUE ERROR
      TRI
      ^

      ⎕EX ⎕AI
DOMAIN ERROR
      ⎕EX ⎕AI
      ^
```

Purpose: Append a value to the end of a component file by adding a new component.

Syntax:
result ← *value* ⎕*FAPPEND tieno*
result ← *value* ⎕*FAPPEND tieno pass*

Arguments:

value	variable (or value) to be appended to the file
tieno	file tie number
pass	passnumber

value can have any rank, shape, or data type.

The right argument must be an integer-valued singleton or two-element vector with a valid tie number (*tieno*) and optional valid passnumber.

If the passnumber is omitted, it is assumed to be zero.

Result: *result* is the number of the new component.

Effect: Appends a new data component to the file along with component information (⎕*FRDCI*). This process increases the disk space occupied by the file.

Access: The file must be tied, the passnumber must match the one in effect, and you must have append access. The access code for ⎕*FAPPEND* is 8.

Errors:
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE FULL
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL

Examples: The first example places the visual representation of *TRI* in the next component of the file tied to 27 and captures the component number in the variable *COMP*.

```
        COMP←(⎕VR 'TRI') ⎕FAPPEND 27
```

The next example appends the variables *JANSALES* and *FEBSALES* at the end of the file tied to 33.

```
        ⎕FSIZE 33
1 20 36412 100000

        JANSALES←48032
        JANSALES ⎕FAPPEND 33
20

        ⎕FSIZE 33
1 21 36432 100000
```

Purpose: Indicate availability of the component file system.

Syntax: *result* ← ⎕*FAVAIL*

Result: *result* is 1 if the component file system is available for use, 0 if it is not.

Note: On this APL★PLUS System, the file system is always available. ⎕*FAVAIL* is included for compatibility with other APL★PLUS Systems in which the file system is not always available.

Errors: *WS FULL*

Purpose: Create a new component file.

Syntax:

'*fileid*' ⎕*FCREATE tieno*
'*fileid size*' ⎕*FCREATE tieno*
'*fileid size/comp*' ⎕*FCREATE tieno*

Arguments:

fileid	file identifier (see section 2-2)
size	file size limit in bytes
comp	starting component number
tieno	file tie number

The left argument must be a character scalar or vector designating the file to create. It contains the file identifier (*fileid*) and, optionally, the file size unit (*size*) and starting component number (*comp*). The file name must be different from any others in that directory or library.

The optional *size* specifies a limit on the amount of space the file can occupy on disk. If omitted, the default is 0, meaning the file has no limit on its size. *size* is specified in bytes and must be an integer value. The file size limit can be changed later by ⎕*FRENAME* or ⎕*FRESIZE*.

The optional *comp* specifies the starting component number for the new file. It must be integer-valued and follow a slash (/) in the argument. If omitted, the starting component number is 1.

The file tie number (*tieno*) must be a positive integer-valued singleton. You must have no other file currently tied with this number.

Effect: Creates a new file and ties it to the tie number specified.

Access: No file access code is required for ⎕*FCREATE*. However, you must be authorized to create files in the specified or default directory or library.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE NAME ERROR
FILE TIE ERROR
LIBRARY NOT FOUND
RANK ERROR
WS FULL
```

Examples:

```
'TEXTFILE' ⎕FCREATE 27

'PRINTFILE 225000' ⎕FCREATE 1

'[MYERS]D87 0/11001' ⎕FCREATE 99

⎕LIBD '12 [MYERS]'
'12 DATA88' ⎕FCREATE 98
```

File Drop of Components ⎕*FDROP*

Purpose: Drop components from either end of a component file.

Syntax:
⎕*FDROP tieno n*
⎕*FDROP tieno n passno*

Arguments:

tieno	file tie number
n	number of components to drop
passno	pass number

The argument must be a two- or three-element integer vector which designates the file by tie number (*tieno*), the components to drop, and an optional passnumber. If the passnumber is not specified, it is assumed to be zero.

Effect: Drops components from a file. If ***n*** is positive, ***n*** components are dropped starting from the beginning of the file. If ***n*** is negative, (| *n*) components are dropped from the end of the file. If *n* is zero, no components are dropped.

Access: The file must be tied, the passnumber must match the one in effect, and the user must have drop access. The access code for ⎕*FDROP* is 32.

Errors:
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE INDEX ERROR
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR

Examples:

```
      ⎕FSIZE 27
1 10 7424 0

      ⎕FDROP 27 2 ◊ ⎕FSIZE 27
3 10 7424 0

      ⎕FDROP 27 ¯3 ◊ ⎕FSIZE 27
3 7 2536 0
```

Purpose: Create an exact copy of a file with a new name and compact it, if possible, to occupy less disk space.

Syntax:

'fileid' ⎕*FDUP tieno*

'fileid size/comp' ⎕*FDUP tieno passno*

Arguments:

fileid	file identification (see section 2-2)
size	file size limit in bytes
comp	initial component number
tieno	file tie number
passno	file passnumber

The left argument must be a character scalar or vector designating the new file to create. It contains the file identifier (*fileid*) and, optionally, the file size limit (*size*) and starting component (*comp*). The fileid must be different from any others in that directory or library.

The optional *size* specifies a limit on the amount of storage a file can occupy on disk. If omitted, the default is 0, meaning the file has no limit on its size. *size* is specified in bytes and must be integer-valued.

comp specifies the starting component number for the new file. It, too, must be integer-valued and must follow a slash (/) in the argument. If omitted, the starting component number is 1.

The file tie number (*tieno*) must be a positive integer-valued singleton. You must have no other file currently tied with this number.

Effect: ⎕*FDUP* creates a new file with the specified name (*fileid*) and copies all the data from the file specified by *tieno* into it. Unused space created by replacing records with a different sized component is retrieved in the process, potentially allowing the new file to occupy less disk space than the original file. The old file remains unchanged.

Caution: ⎕*FDUP* as described here is specific to this APL★PLUS System. It may be different or absent in other APL★PLUS Systems. In particular, the APL★PLUS System for the PC allows ⎕*FDUP* to duplicate the file onto itself; this implementation does not. Note also that ⎕*FDUP* does not preserve the component information (⎕*FRDCI*) of the old file. This behavior may change in a future release and may be different on other APL★PLUS Systems.

Access: The file to be duplicated must be tied, the passnumber must match the one in effect, and you must have both duplicate access and the authority to create files in the specified (or default) directory or library. The access code for ⎕*FDUP* is 16384.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE TIE ERROR
HOST ACCESS ERROR
LIBRARY NOT FOUND
WS FULL
```

Examples:

```
      ⎕FLIB ''
LISTINGS

      'LISTINGS' ⎕FTIE 10

      ⎕FNAMES
LISTINGS

      'LEANINGS' ⎕FDUP 10

      ⎕FNAMES
LISTINGS

      ⎕FLIB ''
LEANINGS
LISTINGS
```

Purpose: Erase a tied component file.

Syntax:

```
'fileid' ⎕FERASE tieno
'fileid' ⎕FERASE tieno pass
```

Arguments:

fileid file identifier (see section 2-2)
tieno file tie number
pass file passnumber

The left and right arguments designate the same file. The left argument is a character vector or scalar containing the file identification (*fileid*).

The right argument must be a integer-valued singleton or two element vector designating the file by tie number (*tieno*) and, optionally, the passnumber. If the passnumber is not specified, it is assumed to be zero.

Effect: Unties a file and erases it from the directory or library. All of the data in the file is destroyed.

Access: A file must be tied. The passnumber must match the one in effect and you must have erase access. The access code for ⎕*FERASE* is 4. The file cannot be erased if any other user also has it tied.

Errors:

```
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE NAMES ERROR
FILE TIE ERROR
FILE TIED
HOST ACCESS ERROR
LENGTH ERROR
LIBRARY NOT FOUND
RANK ERROR
WS FULL
```

Examples:

```
      'TEXTFILE' ⎕FTIE 10
      'TEXTFILE' ⎕FERASE 10

      'PRTFILE' ⎕FSTIE 33 707
      'PRTFILE' ⎕FERASE 33 707
```

APL Component File History ⎕*FHIST*

Purpose: Provide historical information about an APL component file.

Syntax: *result* ← ⎕*FHIST tieno*

Argument: *tieno* file tie number

tieno must be a scalar or one-element vector containing a valid file tie number.

Result: *result* is a three-row integer matrix containing information about the history of the file. Row 1 contains the user number of the file owner and the timestamp of the file's creation in both packed form and ⎕*TS* form. Row 2 contains the user number and timestamp associated with the most recent change to the file. Row 3 contains the user number and timestamp associated with the most recent setting of the file access matrix.

Access: The file must be tied and the passnumber must match the one in effect. In addition, the operating system must allow you to read the file. If not, a *HOST ACCESS ERROR* results.

Warning: ⎕*FHIST* is experimental in this release of this APL⋆PLUS System. This feature may change or be removed in a future release.

Example:

```
                    'TESTFILE' ⎕FTIE 1 ◊ ⎕FHIST 1
(Created)       103 448289548   1984 3 16 12 52 29 0
(Last change)   199 449334082   1984 4  1  9 19 34 0
(Access set)    103 448443819   1984 3 18 17 56 53 0
```

Purpose: Synchronize file operations in shared file systems.

Syntax: ⎕*FHOLD tieno*
⎕*FHOLD tieno pass*

Argument:

tieno file tie numbers
pass file passnumbers

The argument designates the files (by file tie numbers) and the passnumbers. If a passnumber is not specified, it is assumed to be zero. The argument must be an integer array consisting of one of the following:

- a scalar, vector, or one-row matrix of file tie numbers
- a two-row matrix whose first row contains file tie numbers and whose second row contains corresponding passnumbers.

Effect: Provides an interlock by which multiple users can synchronize file updates. Only one user can have the interlock at any one time. Each user executing ⎕*FHOLD* waits in a queue until his turn comes to have the interlock (Note: ⎕*FHOLD* does not lock files).

⎕*FHOLD* first releases any current interlocks and then, when it's your turn, sets an interlock on each designated file. No interlocks are set while another user has an interlock set on any of the designated files; ⎕*FHOLD* execution waits until all such other interlocks have been released. While an interlock is set, other users are delayed in turn from completing execution of their ⎕*FHOLD* operations but not from executing other file operations.

All interlocks are released when the user who set them executes another ⎕*FHOLD*, exits APL, enters immediate execution mode, or signals a strong interrupt. The interlock on an individual file can be released without affecting other interlocks by untying or retying the file.

File interlocks are not released when a program stops for ⎕ or ⍞ input. Stopping for input when files are held can impose long delays on other users and should be avoided except when necessary.

File tie numbers must be distinct, and they must designate tied files. An empty vector or a one- or two-row, zero-column matrix releases all interlocks and does not set any.

Access: The file must be tied, the passnumber must match the one in effect, and you must have hold access. The access code for *⎕FHOLD* is 2048.

Errors:

DOMAIN ERROR
FILE ACCESS ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR

Example: The following example holds a file while an update is performed:

```
        ⎕FHOLD 2 2⍴27 33 0 ¯317232

    ∇ FOO
.
.
.
[5]     ⍝ UPDATE DIRECTORY
[6]     ⎕FHOLD TN
[7]     ENTRY←((⎕FREAD TN,1),[1] NEW)
[8]     ENTRY ⎕FREPLACE TN,1
[9]     ⎕FHOLD ⍳0
.
.
.
    ∇
```

Purpose: Convert a character string to numeric values.

Syntax: *result* ← ⎕*FI data*

Argument: *data* character string to convert

data is a character singleton or vector.

Result: *result* is a numeric vector formed by taking *data* and converting it to numbers. The conversion process uses the same rules as when numbers are entered from the keyboard in immediate execution mode. Groups of characters that are invalid numbers appear as zeros in *result* .

Errors:

```
DOMAIN ERROR
LIMIT ERROR
RANK ERROR
WS FULL
```

Examples:

```
      A←'666 ¯1.20  .1  314159E¯5'
      ⎕FI A
666 ¯1.2 0.1 3.14159

      ⎕FI '  2  '
2

      ρ⎕FI '  2  '
1

      ρ⎕FI ' '
0

      ⎕FI 'ANSWER: 666'
0 666

      B←'ANSWER IS 666 LBS.'
      ⎕FI B
0 0 666 0

      C←'   .25 -6.25  8,9,10'
      ⎕FI C
0.25 0 0
```

Purpose: Produce a character matrix of all the component files in a library or directory.

Syntax:

result ← ⎕*FLIB* ''
result ← ⎕*FLIB dir*
result ← ⎕*FLIB lib*

Arguments:

dir directory name
lib library number

If the system is in directory mode, the argument, if supplied, must be a character vector or scalar representing a valid directory name (*dir*).

If the system is in library mode, the argument, if supplied, must be a positive integer singleton that has been associated with a directory with ⎕*LIBD* or a startup parameter.

An empty character or numeric vector argument indicates the user's default directory or library.

Result: The form of *result* depends on the argument supplied and the system mode (library or directory).

If the system is in directory mode (the default) and no argument or directory name is supplied, *result* is a character matrix of file names, left justified; the number of columns is the length of the longest file name in the list (the directory prefix and file suffix (.VF) are omitted from the list).

If the system is in library mode, the result is a 22-column character matrix containing one file identification per row. The columns in the result are defined as follows:

Column 1-10	Library number, right justified
Column 11	Space
Column 12-22	File name, left justified

When the system is in library mode, you can still supply a directory name as an argument to ⎕*FLIB*. The result is a library-style display of file names with 1↑⎕*AI* used as the library number.

)*FLIB* produces the same list of files formatted in multiple columns and without library numbers for convenient viewing on the terminal.

In all modes, the files are listed in alphabetic order.

Errors:

```
DOMAIN ERROR
LENGTH ERROR
LIBRARY NOT FOUND
WS FULL
```

Examples:

```
      ⎕FLIB '[APL.REL1]'  (Directory mode.)
CONVERT
DATES
SERXFER

      ρ⎕FLIB '[APL.REL1]'
3 7
                                        (Switch to library mode.)
      ⎕LIBD '123 [APL.REL1]'
      ⎕FLIB 123
 123 CONVERT
 123 DATES
 123 SERXFER

      ρ⎕FLIB 123
3 22
```

Purpose: Format character and numeric data into a character matrix with advanced formatting features. ⎕*FMT* is described in detail with many examples in Chapter 4 of the *APL ⋆PLUS System User's Manual.*

Syntax:

result ← *formatstring* ⎕*FMT data*

result ← *formatstring* ⎕*FMT* (*data1*;*data2*;...;*datan*)

result ← *formatstring* ⎕*FMT* (⊂*data1*), (⊂*data2*) ... ⊂*datan*

Arguments:

data, datan	APL arrays
formatstring data2,	format phrases to be applied to *data, data1,* and so on

formatstring is a character vector that contains combinations of editing and positioning format phrases separated by commas. These phrases control the editing and display of *data* in the right argument.

Format Phrases

rmAw	Character
rmEw.s	Exponential
rmFw.d	Fixed point
rmG <*pattern*>	Pattern
rmIw	Integer
Tp or *T*	Absolute tab
rXp	Relative tab
r <*text*>	Text insertion

where:

d	Decimal position parameter (*F*)
m	Optional Modifier
p	Position parameter (*T*, *X*)
pattern	Pattern text parameter (*G*)
r	Optional repetition factor
s	Significant digits parameter (*E*)
w	Field width parameter (*A*, *E*, *F*, *I*)

Any combination of the following modifiers can be used with the phrases shown in parentheses:

Format Phrase Modifiers

B	Blank if zero (*F*,*I*)
C	Comma insertion (*F*,*I*)
K i	Scale argument by 10⋆*i* (*E*,*F*,*G*,*I*)
L	Left justify (*F*,*I*)
M<*text*>	Negative left decoration (*F*,*G*,*I*)
N<*text*>	Negative right decoration (*F*,*G*,*I*)
O<*text*>	Format zeros as text (*F*,*G*,*I*)
P<*text*>	Positive or zero left decoration (*F*,*G*,*I*)
Q<*text*>	Positive or zero right decoration (*F*,*G*,*I*)
R<*text*>	Background fill (*A*,*E*,*F*,*G*,*I*)
S<*symbolpairs*>	Symbol substitution (*F*,*G*,*I*)
Z	Zero fill (*F*,*I*)

The text in the decorations, background fill, symbol substitution, and text insertion can be delimited by any of the following pairs of symbols:

<	>
⊂	⊃
¨	¨
⎕	⎕
⍞	⍞
/	/

Multiple format phrases for individual data columns are separated by commas within *formatstring*. A group of format phrases can be repeated by enclosing it in a pair of parentheses and preceding the left parenthesis with a repetition factor.

The right argument can contain any numeric or character array. It can also be a strand (a vector of enclosed arrays).

Result: *result* is a character matrix of the data formatted as specified.

Caution: Older APL⋆PLUS Systems use a special list (*data1*;*data2*) to format multiple arrays of different types. This system supports this form for compatibility, but a nested vector or a strand can be also used, perhaps more conveniently. For example, the following expressions produce the same result:

```
CHAR←3 3ρ'ONE TWO SIX'
NUM←1000×23

'3A1,I5' ⎕FMT(CHAR;NUM) (old way)
'3A1,I5' ⎕FMT CHAR NUM  (new way)
```

Examples:

```
      'I5,2F8.1,E9.3' ⎕FMT 3 4ρι12
    1     2.0     3.0   4.00E0
    5     6.0     7.0   8.00E0
    9    10.0    11.0   1.20E0

      'G<(999) 999-9999' ⎕FMT 3019845000
(301) 984-5000

      FSTR←'3A1,<*PLUS >,6A1'
      FSTR ⎕FMT 1 9ρ'APLSYSTEM'
APL*PLUS SYSTEM
```

Purpose: Return the file identifications of all tied component files (files tied with *⎕FTIE* or *⎕FSTIE*).

Syntax: *result* ← *⎕FNAMES*

Result: *result* is a character matrix of file identifications. The form and shape of *result* depends on whether the system is in library or directory mode. The rows of *result* have the same order as *⎕FNUMS*.

In directory mode (the default) *⎕FNAMES* formats *result* to be as wide as needed to contain the directory path and file name in the same form as supplied when the file was tied.

In library mode, the result is 22 columns wide formatted as follows:

Columns	1-10	Library number
Column	11	Blank
Columns	12-22	Filename

Errors: *WS FULL*

Examples:

```
      ⎕FNAMES                (In directory mode.)
[APL.WSS]CHAPTER1
TEMP
PRINTFILE

      ⎕FNAMES                (In library mode).
        76 CHAPTER1
       101 TEMP
       101 PRINTFILE
```

Purpose: Display the tie numbers of all tied component files (files tied with ⎕*FTIE* or ⎕*FSTIE*).

Syntax: *result* ← ⎕*FNUMS*

Result: *result* is a numeric vector of file tie numbers. The tie numbers are in the same order as the file names reported by ⎕*FNAMES*, which is the order in which they were tied.

Errors: *WS FULL*

Examples:

```
      ⎕FNUMS
27 33 17

      ⎕FUNTIE ⎕FNUMS  (Untie all tied files at one time.)

      ρ⎕FNUMS
0
```

Purpose: Report the current access matrix for an APL component file.

Syntax:

result ← ⎕*FRDAC tieno*
result ← ⎕*FRDAC tieno pass*

Arguments:

tieno file tie number
pass passnumber

The right argument is an integer-valued singleton or two-element vector designating the file (by tie number) and optionally the passnumber. If the passnumber is omitted, it is assumed to be zero.

Result: *result* is a three-column numeric matrix containing the access matrix of the file. A newly created file has an access matrix with no rows.

Access: The file must be tied, the passnumber must match the one in effect, and you must have the authority to read the access matrix. The access code for ⎕*FRDAC* is 4096.

Errors:

DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE DAMAGED
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL

Examples:

```
      ⍴⎕FRDAC 27        (File with empty access matrix.)
0 3

      ⎕FRDAC 33 7655
12304   16059   7566
23405   16063      0
```

File Read of Component Information ⎕*FRDCI*

Purpose: Return information about one component of a file.

Syntax: *result* ← ⎕*FRDCI tieno comp*
result ← ⎕*FRDCI tieno comp pass*

Arguments:

tieno	file tie number
comp	component number
pass	passnumber

The right argument must be an integer-valued, two- or three-element vector. If the passnumber is omitted, it is assumed to be zero.

Result: *result* is a ten-element numeric vector containing the following information:

- the workspace storage needed to hold the component, in bytes.
- the account number of the user who most recently executed ⎕*FAPPEND* or ⎕*FREPLACE* on the component.
- the timestamp, in ⎕*WSTS* format (microseconds since 00:00 on January 1, 1900) , when the component was last written to file. Use the *TIME* function in the workspace *FILEAID* (see Chapter 4, Supplied Functions) to interpret the timestamp. The microsecond resolution is maintained for compatibility with other APL*PLUS Systems. The clock accuracy, however, is one second.

Access: The file must be tied, the passnumber must match the one in effect, and you must have the authority to read the access matrix. The access code for ⎕*FRDCI* is 512.

Errors:

DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE DAMAGED
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL

Example:

```
      )COPY DATES FTIMEFMT
SAVED  17:00:46  01/26/86

      FTIMEFMT (⎕FRDCI 27 1)[3]
7/14/87 15:14:00.000
```

Purpose: Read a component of a file and make it available in the workspace as a variable.

Syntax:
result ← ⎕*FREAD tieno comp*
result ← ⎕*FREAD tieno comp pass*

Arguments:

tieno	file tie number
comp	component number
pass	passnumber

The argument is an integer-valued two- or three-element vector that designates the data to be returned by file tie number (*tieno*), the component number (*comp*), and the passnumber. If the passnumber is omitted, it is assumed to be zero.

Result: *result* is the actual value stored in the file component.

Access: The file must be tied, the passnumber must match the one in effect, and *comp* must be a valid component number. The access code for ⎕*FREAD* is 1.

Errors:
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE DAMAGED
FILE DATA ERROR
FILE INDEX ERROR
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL

Examples:

```
      ⎕FREAD 27 1
THIS FILE CONTAINS DATA FOR 1982
CREATED 26 JANUARY 1987.
```

```
      ⎕FREAD 27 2
SMALLS, BARRY T. 4856739 6/30/85

      A←⎕FREAD 27 3
      A
SMITH, KAREN M. 3847384 3/01/86
      ⍴A
40
```

Purpose: Change the name of a file.

Syntax:
fileid ⎕*FRENAME tieno*
fileid size ⎕*FRENAME tieno pass*

Arguments:

fileid	file identification (see section 2-2)
pass	passnumber
size	file size limit
tieno	file tie number

The left argument, a character scalar or vector, designates the new file identification and, optionally, the new size limit. The new file name must not already exist in the library. The *fileid* must be specified consistent with the mode selected (directory or library).

If a directory name or library number is specified, it must designate a library in which you are allowed to own files. If the directory or library number is omitted, your default library is assumed.

The right argument, an integer-valued singleton or two-element vector, designates the old file identification by tie number and optional passnumber. If the passnumber is not specified, it is assumed to be zero.

Effect: ⎕*FRENAME* changes the file name to the one specified in the left argument, potentially moving it to a different directory. If the file name already exists, the system signals a *FILE NAME ERROR*.

The result of ⎕*FNAMES* will reflect the new file identification. The user who renames the file becomes the new file owner.

⎕*FRENAME* can be applied to a file that is share tied. Other users do not become aware of the name change until the next time they attempt to tie the file. If ownership of the file is changed, the former owner will lose all access to the file except that which is explicitly granted by the access matrix.

Access: The file must be tied, the passnumber must match the one in effect, and you must have rename access. You must be authorized to own files in the designated directory and must have a sufficient user storage limit to accommodate the present space needed by the file. The access code for *⎕FRENAME* is 128.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE NAME ERROR
FILE SIZE ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR
```

Examples:

```
      ⎕FLIB ''
PRIMES

      'PRIMES' ⎕FTIE 10

      'PRIMENUMBERS' ⎕FRENAME 10

      ⎕FLIB ''
PRIMENUMBERS
```

(Directory mode.)

```
      'NEWNAME' ⎕FRENAME 10
```

(Library mode.)

```
      ⎕LIBD '101 [MLO]'
      '101 NEWNAME' ⎕FRENAME 10
```

Replace Component — *⎕FREPLACE*

Purpose: Change the value of an existing component of a file.

Syntax:
value ⎕FREPLACE tieno comp
value ⎕FREPLACE tieno comp pass

Arguments:

value	any APL object
tieno	file tie number
comp	component number
pass	passnumber

value is the value to be stored in the file. It can have any rank, shape, or datatype.

The right argument, a two- or three-element integer vector, designates where to store the data by file tie number (*tieno*) and, optionally, by passnumber (*pass*). If the passnumber is omitted, it is assumed to be zero.

Effect: Replaces the designated component of the file with a new value. It also updates the component information (*⎕FRDCI*). Replacing a component with a smaller or larger value may change the file size.

Access: The file must be tied, the passnumber must match the one in effect, and you must have append access. The access code for *⎕FAPPEND* is 16.

Errors:
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE FULL
FILE INDEX ERROR
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL

Examples:

```
LIBRARY←⎕FREAD 33 10
LIBRARY←LIBRARY, ⎕USERID
LIBRARY ⎕FREPLACE 33 10
```

Purpose: Reset the file size limit of a component file.

Syntax: *size* ⎕*FRESIZE tieno*
size ⎕*FRESIZE tieno pass*

Arguments:

size	file size limit in bytes
tieno	file tie number
pass	passnumber

size is the new file size limit in bytes. It must be a positive integer scalar or one-element vector greater than or equal to the current size of the file. *size* may also be zero, meaning that the file has no size limit.

The right argument, a singleton or two-element integer vector, designates the file by tie number (*tieno*) and optional passnumber (*pass*). If the passnumber is omitted, it is assumed to be zero.

Effect: Changes the file size limit to the specified value. If *size* is zero (the default for a new file), the file has no size limit, meaning that it can grow as large as needed.

Access: The file may be tied, the passnumber must match the one in effect, and the user must have resize access. The access code for ⎕*FRESIZE* is 1024.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE SIZE ERROR
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL
```

Example:

```
      ⎕FSIZE 27
1 50 94560 100000
      2600000 ⎕FRESIZE 27
      ⎕FSIZE 27
1 50 94560 2600000
```

Purpose: Return size limits of a component file.

Syntax:

result ← ⎕*FSIZE tieno*
result ← ⎕*FSIZE tieno pass*

Arguments:

tieno file tie number
pass passnumber

The argument, an integer scalar or two-element vector, designates the file by tie number (*tieno*) and optional passnumber (*pass*). If the passnumber is omitted, it is assumed to be zero.

Result: *result* is a four-element numeric vector with the following information:

[1] the number of the first component in the file

[2] the next available component

[3] the physical storage (in bytes) used by the file, including data, overhead, and access matrix

[4] the size limit for the file as set by the user (a value of zero means no upper limit)

Errors:

DOMAIN ERROR
FILE ACCESS ERROR
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL

Examples:

```
      'PRIMES' ⎕FSTIE 37
      ⎕FSIZE 37
7 53 28672 100000

      'NEWFILE' ⎕FCREATE 13
      ⎕FSIZE 13
1 1 2048 10
```

Purpose: Set the access matrix of a component file.

Syntax:
access ⎕*FSTAC tieno*
access ⎕*FSTAC tieno pass*

Arguments:

access	access matrix
tieno	file tie number
pass	passnumber

access is the new access matrix. It is a three-column integer matrix or a three-element vector. See Chapter 3 of the *APL ⋆PLUS System User's Guide* for more information on access matrices.

The right argument, an integer scalar or one- or two-element vector, designates the file by tie number (*tieno*) and optional passnumber (*pass*). If the passnumber is omitted, it is assumed to be zero.

Effect: Replaces the access matrix for the file. The new access restrictions are imposed on a user the next time the file is tied by that user. ⎕*FSTAC* may increase the amount of disk storage occupied by the file.

Access: The file must be tied, the passnumber must match the one in effect, and the user must have the authority to change the access matrix. The access code for ⎕*FSTAC* is 8192.

Errors:
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE FULL
FILE TIE ERROR
HOST ACCESS ERROR
LENGTH ERROR
RANK ERROR
WS FULL

Example:

```
MAT←2 3ρ4772490 2 666 1000 ¯1 0

MAT ⎕FSTAC 33

⎕FRDAC 33
4772490       2       666
   1000      ¯1         0
```

File Share Tie ⎕*FSTIE*

Purpose: Tie a component file for shared use.

Syntax: *fileid* ⎕*FSTIE tieno*
fileid ⎕*FSTIE tieno pass*

Arguments:

fileid	file identification (see section 2-2)
tieno	file tie number
pass	optional passnumber

fileid must be a character vector or singleton containing the file identification of an existing file. If the directory or library number is not specified, the default library is assumed.

The right argument, an integer scalar or one- or two-element vector, designates the file tie number (*tieno*) and optional passnumber (*pass*). If the passnumber is omitted, it is assumed to be zero.

Effect: The file is share tied. File ties are "slippery;" that is, if a file is already tied to one tie number, ⎕*FSTIE* can tie the file to the same number or to another unused tie number without requiring the file to first be untied.

Access: The file must exist and must not be exclusively tied (⎕*FTIE*) by anyone, although it can be share tied by others. The user must have some form of access to the file, and the passnumber must match the one in the access matrix.

Note: More than one user can simultaneously update a file when ⎕*FSTIE* is used (see ⎕*FHOLD* , ⎕*FTIE*).

Errors:

DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE NAME TABLE FULL
FILE NOT FUOND
FILE TIE ERROR
FILE TIE QUOTA EXCEEDED
FILE TIED
HOST ACCESS ERROR
LENGTH ERROR
LIBRARY NOT FOUND
RANK ERROR

Examples:

```
'PRIMES' ⎕FSTIE 37
```

(Directory mode.)

```
'[APL.REL1]MYFILE' ⎕FSTIE 22
```

(Switch to library mode.)

```
⎕LIBD '12345 [APL.WSS]'
'12345 PRINTOUT' ⎕FSTIE 1 666
```

File Tie ⎕*FTIE*

Purpose: Tie a component file for exclusive (non-shared) use.

Syntax: *fileid* ⎕*FTIE* *tieno*
fileid ⎕*FTIE* *tieno pass*

Arguments:

fileid	file identification (see section 2-2)
tieno	available positive file tie number
pass	optional integer passnumber

fileid must be a character vector or singleton containing the file identification of an existing file. If the directory name or library number is not specified, the default directory is assumed.

The right argument, an integer scalar or one- or two-element vector, designates the file tie number (*tieno*) and optional passnumber (*pass*). If the passnumber is omitted, it is assumed to be zero.

Effect: The file is exclusively tied. No other user will be able to tie the file as long as it remains exclusively tied.

File ties are "slippery;" that is, if a file is already tied to one tie number, ⎕*FTIE* will allow you to tie the file to the same number or to another unused tie number without requiring the file to first be untied.

Access: The file must exist, it must not be tied by anyone else, the user must have the authority to exclusively tie the file, and the passnumber must match the one in the access matrix of the file. The access code for ⎕*FTIE* is 2 (see ⎕*FSTAC*).

Note: Only one user can update a file when ⎕*FTIE* is used (see ⎕*FHOLD*, ⎕*FSTIE*).

Examples:

```
'PRIMES' ⎕FTIE 37
```

(Directory mode.)

```
'[APL.REL1]MYFILE' ⎕FTIE 2
```

(Switch to library mode.)

```
⎕LIBD '12345 [APL.REL1]'
'12345 MYFILE' ⎕FTIE 1
```

File Untie ⎕*FUNTIE*

Purpose: Untie one or more component files.

Syntax: ⎕*FUNTIE* *tieno1 tieno2 tieno3 . . . tieno*

Argument: *tieno1 tieno2 tieno3 . . . tieno* file tie numbers of files to be untied

The argument is an integer scalar or vector of possible file tie numbers. Elements of the argument need not be in use as file tie numbers. An empty vector is permitted as an argument and does not affect any file ties.

Effect: The files tied to any of the tie numbers in the argument are untied. This frees the file tie slot for possible re-use with another file. Any file holds in effect are released.

Errors:
DOMAIN ERROR
RANK ERROR
WS FULL

Examples:

```
      ⎕FUNTIE 33
      ⎕FUNTIE ⎕FNUMS        (Unties all current ties.)
```

Purpose: Define (fix) a function from a character matrix (canonical) representation of the function (see also ⎕*CR* and ⎕*DEF*).

Syntax: *result* ← ⎕*FX fnrep*

Argument: *fnrep* function representation

fnrep contains the canonical representation of a function (the result of ⎕*CR*) as a character matrix. The lines of the matrix should not contain bracketed line numbers, nor should they contain ∇ or ⍫ other than in comments or character constants. Blanks that would be superfluous in function definition mode are ignored by ⎕*FX*.

Result: If the function definition is successful, *result* is a character vector containing the name of the function defined.

If the function definition is not successful, *result* is a numeric scalar containing the index of the matrix argument where the first fault was found. *result* depends on the index origin (⎕*IO*).

Effect: Defines the specified function in the active workspace unless an error condition occurs. The amount of available workspace area and the number of symbols may change.

If the name of the function that has been defined corresponds to a local identifier in a currently executing, pendent, or suspended function, the newly-defined function is local to that function and is erased when the function in which it is localized completes execution.

If the name of the function that has been defined corresponds to the name of an existing function, the existing function is replaced and any ⎕*STOP* or ⎕*TRACE* settings in the function are removed.

Notes: ⎕*DEF* and ⎕*FX* provide similar capabilities. ⎕*DEF* is a more powerful and general case of ⎕*FX*. The differences are outlined below:

- ⎕*DEF* accepts both canonical (matrix) and visual (vector) representations of a function; ⎕*FX* accepts only the canonical representation.
- ⎕*DEF* can create a function as a locked function; ⎕*FX* cannot.
- ⎕*DEF* indicates both the cause and the location of an error; ⎕*FX* indicates only the location.
- ⎕*DEF* indicates the *SYMBOL TABLE FULL* or *WS FULL* conditions via error codes without halting execution. ⎕*FX* halts execution.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Example:

```
      ⎕FX 3 5⍴'ABC  DEFG HIJKL'
ABC
      ⎕VR 'ABC'
    ∇ ABC
[1]   DEFG
[2]   HIJKL
    ∇
```

Purpose: Return a character matrix of identifiers (names). The list can be restricted to those that begin with designated letters.

Syntax:

result ← ⎕*IDLIST class*
result ← *letters* ⎕*IDLIST class*

Arguments:

class	the classification of identifiers to be included in *result*
letters	an optional character scalar or vector specifying the first
letters	of identifiers to be selected

The right argument*class* is the sum of one or more of these values:

Value	**Identifier**
1	functions
2	variables
8	labels

To obtain a combination of identifier types, the sum of the appropriate values is used.

letters restricts the names included in *result* to those whose first letter occurs in *letters*. If *letters* is not specified, all identifiers of the specified types are produced.

Result: *result* is a character matrix of identifiers. The rows are in alphabetic order.

Note: ⎕*IDLIST* and ⎕*NL* provide similar capabilities, but they use different classification codes and arguments. In addition, ⎕*IDLIST* accepts an argument consistent with the result of ⎕*IDLOC*; ⎕*NL* accepts an argument consistent with the result of ⎕*NC*. For maximum portability to other APL systems, use ⎕*NL* rather than ⎕*IDLIST*.

Errors:

DOMAIN ERROR
LENGTH ERROR
WS FULL

Example: List all functions that begin with T, U, or V.

```
        'TUV' ⎕IDLIST 1
TRI
VALIDATE
```

Purpose: Return the local and global classifications of a list of identifiers.

Syntax: *result* ← ⎕*IDLOC idlist*

Argument: *idlist* list of identifiers

idlist contains a list of zero or more identifiers. It can be represented as a character vector containing two or more identifiers separated by one or more blanks or a character matrix with one identifier in each row.

Result: *result* is a numeric matrix with each row corresponding to an identifier named in *idlist*. The matrix has one column for each function in the state indicator, progressing from the most local to the most global in increasing column order. The last column contains the global definitions.

Values that may be returned are shown in the following table. The values in the last column are always non-negative.

Value	Classification
¯1	Not localized at this level
0	Localized with no assigned value at this level or globally undefined
1	System or user-defined function
2	System or user-defined variable with value
8	Label

Note: ⎕*IDLOC* and ⎕*NC* provide similar capabilities, but they use different classification codes and arguments. Other differences include:

- ⎕*IDLOC* returns all local and global classifications; ⎕*NC* returns only the locally active classifications of the identifier.
- ⎕*IDLOC* is more informative than ⎕*NC*. Different numeric codes are used by each; ⎕*NC* returns a less specific classification code.

- ⎕*IDLOC* accepts either a character matrix or character vector; ⎕*NC* accepts only a character matrix as an argument.

- ⎕*IDLIST* returns a result consistent with ⎕*IDLOC*; ⎕*NL* returns a result consistent with ⎕*NC*.

- ⎕*NC* accepts an ill-formed identifier name; ⎕*IDLOC* produces a *DOMAIN ERROR*

For maximum portability to other APL systems, use ⎕*NC* rather than ⎕*IDLOC* when appropriate.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Example:

```
      )SINL
TRI[1]     *        N       A
TEST[1]             A

      ⎕IDLOC 'A N TRI'
 0  8  0
 2 ¯1  0
¯1 ¯1  1
```

This example shows that *A* is undefined (0) in the most local environment (*TRI*), where it is localized but has not been defined by assigning it a value. In the environment of *TEST*, *A* is defined as a label (8). *A* has no global definition (0).

Purpose: Read one keystroke at a time from the terminal.

Syntax: *result* ← ⎕INKEY

Result: *result* is a character scalar containing the first key typed at the terminal or the first key in the type-ahead buffer.

Effect: Waits for a single character of keyboard input. The input is not displayed on the screen when it is typed, but instead returned as *result*.

Multiple keystrokes typed by the user are buffered and only the first character is returned. The remaining characters can be read by further use of ⎕INKEY. Logical function keys are returned as a single character; that is, they are not expanded into the multiple keystroke definition specified by ⎕PFKEY.

If Ctrl-C (interrupt) is pressed, ⎕INKEY returns a Ctrl-C (⎕AV[3+⎕IO]) and signals a weak interrupt.

Caution: ⎕INKEY as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Example:

```
      'Q'=⎕INKEY
1                                   (User pressed a "Q".)
```

Index Origin ⎕*IO*

Purpose: Set or retrieve the value of the index origin. The value of ⎕*IO* is used in the definition of several APL functions.

Syntax:

```
value ← ⎕IO
⎕IO ← value
```

Domain: *value* can be either 0 or 1. In a clear workspace, the default value for ⎕*IO* is 1.

Effect: When generating or referencing index values, the system assumes that indices are numbered starting at ⎕*IO*.

The value of ⎕*IO* is used in connection with:

- computing the result of index generator (monadic ⍳) and index of (dyadic ⍳)
- computing the result of roll (monadic ?) and deal (dyadic ?)
- computing the result of grade up (⍋) and grade down (⍒)
- indexing applied to an array (*A*[...])
- applying the axis operator to a primitive function (⌽[...]*A*)
- interpreting the left argument to dyadic transpose (...⍉*A*)
- computing the result of ⎕*DEF* and ⎕*FX* when an invalid argument is used

Errors: *DOMIAN ERROR*
RANK ERROR

Example: The columns below show the effect of ⎕*IO* on various operations.

```
    ⎕IO←1                            ⎕IO←0

          ⍳5                               ⍳5
1 2 3 4 5                        0 1 2 3 4

          X←5+⍳5                           X←5+⍳5
          X                                X
6 7 8 9 10                       5 6 7 8 9

          X[3]                             X[3]
8                                8
```

```
      X[5]                      X[5]
10                        INDEX ERROR
                                X[5]
                                ^^

      X[0]                        X[0]
INDEX ERROR                 5
      X[0]
      ^^

      1 2 3 4 [3]                 1 2 3 4 [3]
3                           4

      'ABCDEF'[2+⍳3]        'ABCDEF'[2+⍳3]
CDE                         CDE

      V←6 23 11 4 ¯6              V←6 23 11 4 ¯6
      ⍋V                          ⍋V
5 4 1 3 2                   4 3 0 2 1

      X,[0.5] V                   X,[0.5] V
 6  7  8  9 10               5  6
 6 23 11  4 ¯6               6 23
                             7 11
                             8  4
                             9 ¯6

      3?3                         3?3
3 1 2                       2 0 1
```

Line Counter ⎕*LC*

Purpose: Return the current value of the execution line counter.

Syntax: *result* ← ⎕*LC*

Result: *result* is a numeric vector of line numbers from the state indicator beginning with the most local. It does not include any values corresponding to ⍎ or ⎕ symbols appearing in ⎕*SI* or)*SI*.

Effect: While ⎕*LC* just returns the line numbers, it can be used in the expression to resume a stopped or interrupted execution.

Errors: *WS FULL*

Example:

```
      ⎕SI
TRI[2]*
⍎
EXAMPLE[3]

      ⎕LC
2 3
      →⎕LC                    (Restart execution.)
```

Purpose: Return a character matrix of file names in the specified library.

Syntax:
result ← ⎕*LIB dir*
result ← ⎕*LIB lib*

Arguments:
dir directory name (see section 2-2)
lib library number

If the system is in directory mode (the default), the right argument is a character vector or scalar containing the directory name (*dir*) to be searched for files. If the system is in library mode, the right argument is a library number (*lib*).

Result: *result* is a character matrix containing one file identification in each row. The number of columns in *result* is determined by the longest file name in the list. The columns are arranged in alphabetic order.

If an argument is not specified, *result* contains the file identification for your default working directory or library.

Caution: ⎕*LIB*, as described here, is specific to this APL⋆PLUS System. It may be different or absent in other systems.

Errors:
DISK ERROR
DOMAIN ERROR
LENGTH ERROR
LIBRARY NOT FOUND
WS FULL

Examples:

```
      ⎕LIB ''
TEMP.SF
DATA87.SF
      ⎕LIB '[LLG]'
DATES.C
SERHOST
UTILITY
      ⎕LIBD '12 [JGW]'
      ⎕LIB 12
DATES
SERHOST
UTILITY
```

Purpose: Associates a library number with a directory.

Syntax: ⎕*LIBD libdefn*

Argument: *libdefn* library number and the name of a directory

libdefn must be a character vector containing both the library number and the directory name separated by at least one space. The library number should be an integer number (in character form) and the directory name a valid, existing directory.

Effect: Equates the library number with the directory in the argument. The result of ⎕*LIBS* changes accordingly; the number can be used in workspace and file names, and the number can be used to query the contents of the directory. If the library number was defined previously, the new definition replaces the previous one.

No test is made of the validity of the directory name or of the existence of a directory by the given name. If the name is ill-formed or the library does not exist, a *LIBRARY NOT FOUND* message will be produced when you attempt to use the library definition.

Errors: *DOMAIN ERROR*
RANK ERROR

Caution: ⎕*LIBD* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples:

```
      ⎕LIBS
1 [APL.REL1]

      ⎕LIBD '11 [APL.WS]' ◊ ⎕LIBS
 1 [APL.REL1]
11 [APL.WS]
```

Purpose: List the defined APL libraries and the directories to which they correspond.

Syntax: *result* ← ⎕*LIBS*

Result: *result* is a character matrix with one row for each defined APL library. Each row shows the library number and the associated directory to which it corresponds.

The association of a library number and directory can be made when entering APL by a line in the form "`library=`" or in the APL configuration file. Associations between libraries and directories can also be made under program control using ⎕*LIBD*. In the absence of any library definitions, APL is in directory mode, meaning that no libraries are defined. Directories other than the current working directory are referenced by explicitly specifying the directory name.

If no libraries are defined, the result is a zero-row matrix. Thus, the expression 0=1↑ρ⎕*LIBS* is true if and only if the system is in directory mode. This is the definitive test for distinguishing directory mode from library mode under program control.

The libraries listed in ⎕*LIBS* are not guaranteed to exist. Attempts to access or create a file or workspace in a library corresponding to a directory that cannot be located results in a *LIBRARY NOT FOUND* error message.

Errors: *WS FULL*

Caution: ⎕*LIBS* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples:

```
      ⎕LIBS          (Empty result means directory mode.)
      ρ⎕LIBS
0 0

      ⎕LIBS          (Non-empty means library mode.)
 1 [APL.REL1]
11 [APL.WS]
```

Load a Workspace ⎕*LOAD*

Purpose: Replace the active workspace by loading the designated workspace (under program control).

Syntax: ⎕*LOAD wsid*

Argument: *wsid* workspace identification (see section 2-2)

wsid is a character scalar or vector that specifies the workspace to be loaded. If the directory name or library number is omitted, your current default library is assumed.

Effect: The specified workspace becomes the new active workspace, ⎕*WSID* changes, and ⎕*LX* is executed. ⎕*QLOAD* provides a similar capability and does not display the *SAVED* message.

Errors:

```
DISK ERROR
DOMAIN ERROR
LENGTH ERROR
LIBRARY NOT FOUND
RANK ERROR
WS ARGUMENT ERROR
WS NOT COMPATIBLE
WS NOT FOUND
WS TOO LARGE
```

Examples:

```
        ⎕LOAD 'TESTWS'
TESTWS SAVED 12:27:39 07/22/87
```

(Switch to path mode.)

```
        ⎕LIBD '1234 [APL.REL1]'
        ⎕LOAD '1234 TESTWS'
1234 TESTWS SAVED...
```

Purpose: Lock functions under program control.

Syntax: *result* ← ⎕*LOCK fnlist*

Argument: *fnlist* list of function names

fnlist contains a list of the function names that can be represented as a character matrix, with one function name in each row or a character vector containing function names separated by blanks.

Result: *result* is an alphabetized character matrix of requested function names whose definitions cannot be locked. If all requested names are locked, *result* is an empty matrix with shape 0 0.

Effect: Only the most local definition of a function is locked. Functions shadowed by more local use of the same name are not locked.

Locking a function also removes any stop or trace settings it may have (see descriptions of ⎕*STOP* and ⎕*TRACE* in this manual).

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
      ρ⎕VR 'TRI'
72

      ρ⎕CR 'TRI'
3 32

      ⎕LOCK 'TRI'
      ρ⎕VR 'TRI'
0

      ρ⎕CR 'TRI'
0 0
      ⎕LOCK ⎕NL 3          (Lock all functions in
                            the workspace.)
```

Purpose: Store an APL expression to be executed when the workspace is loaded. This provides a convenient way to start an application automatically once it has been loaded.

Syntax:
expr ← ⎕*LX*
⎕*LX* ← *expr*

Domain: *expr* is a character vector containing a valid APL expression. In a clear workspace, the default value for ⎕*LX* is an empty vector (' ').

Effect: Stores a statement that is executed whenever the workspace is loaded (except by ⎕*XLOAD* or)*XLOAD*). If ⎕*LX* represents an invalid APL statement, an error is reported and execution is suspended as if the statement were a line entered in immediate execution mode.

Errors:
DOMAIN ERROR
RANK ERROR

Example: The following example illustrates a typical latent expression:

```
      ∇  AUTOSTART
[1]      'WELCOME TO THIS WORKSPACE'
[2]      MAIN
      ∇

         ⎕LX←'AUTOSTART'
         )SAVE STARTWS
```

The *AUTOSTART* function is executed as soon as the workspace is loaded.

```
         )LOAD STARTWS
STARTWS SAVED...
WELCOME TO THIS WORKSPACE
```


Purpose: Set and unset monitoring of function execution and read monitor data.

Syntax:

```
result ←      ⎕MF fnname
result ← flag ⎕MF fnlist
```

Arguments:

flag	monitoring switch setting
fnlist	list of function names
fnname	function name

flag is a Boolean scalar or one-element vector that controls the monitoring setting. A 1 sets monitoring on, and a 0 turns it off.

fnname is a character scalar or vector containing the name of one function.

fnlist contains a list of function names. It can be represented as a character matrix with one function name in each row or a character vector containing function names separated by blanks.

Monitoring cannot be set or unset on functions that are locked, suspended, pendent, or executing.

Result: The result depends on the arguments supplied. If *flag* and *fnlist* are supplied, *result* is a Boolean vector with one element for each function name in *fnlist*. A 1 indicates that monitoring was successfully set or unset for the corresponding function. A 0 indicates that ⎕*MF* was unable to set or unset monitoring for the corresponding function.

If only *fnname* is supplied, *result* is a three-column integer matrix with one row per function line and one row for the function header. The first row of the result contains information about the execution of the entire function. The second and subsequent rows of the result contain information about the corresponding function line.

[1;1]	Total CPU time for entire function
[1;2]	0
[1;3]	Number of times the function was called
[2…n;1]	Accumulated CPU time for the line
[2…n;2]	CPU time for the line minus that used while subfunctions called on that line were executing
[2…n;3]	Number of times the line was executed

Effect: Sets monitoring on a function and causes it to expand internally to include space for accumulated monitor data. When monitoring is unset, the function contracts to its normal size.

If a function is already being monitored, using `1 ⎕MF` *fnlist* resets monitor data to zero.

A monitored function which is subsequently locked continues to accumulate monitor data while executing. However, the data cannot be read. `0 ⎕MF` *fnlist* can be applied to unset monitoring.

Example: Monitor all functions in the workspace whose name starts with C:

```
      ⍴F←'C' ⎕IDLIST 3
24 15

      ⍴A←1 ⎕MF F
24
      ∧/A
0

      ⎕MF 'COMPLEX'      (Display execution time.)
15 0 3                   (For entire function.)
  8 8 3                  (For line 1.)
  4 4 3                  (For line 2.)
  3 3 3                  (For line 3.)
```

Purpose: Allow APL to call an external machine language routine by associating it with a name in the APL workspace.

Syntax:
result ← ⎕*NA fnname*
result ← *class* ⎕*NA* '*module:fname routine* (*arg*, *arg*...) *res*'

Arguments:

class — syntax class of the external routine. The only possible value of *class* is 3 0 in this release.

fnname — name of a function

module — name of a file with extension `.exe` containing the routine to be called from APL. module must have been defined as a logical name prior to invoking APL with a `DEFINE` command. For example, `$DEFINE VTOM $DUA0:[APL.REL1].EXE`.

fname — name of the APL function created in the workspace by ⎕*NA*. *fnname* is optional; if omitted, *routine* will be used as the function name

routine — name of the entry point in the module to be associated with the APL function created by ⎕*NA*

arg — describes the form of the argument expected by the external routine. The list of argument specifications appears in parentheses, separated by commas. If the external routine requires no parameters, an empty list within parentheses is required. *arg* describes the datatype of each argument, how the argument is passed, and whether it will be modified by the external routine. Any value marked as modifiable will be returned as an item of the explicit result of the external function, whether or not it has actually been modified. Datatypes recognized by the current release of the APL⋆PLUS System are:

arg	**Datatype**
B1	Boolean (1 bit per element)
C1	Character (1 byte per element)
I4	Integer (4 bytes per element)
D4	VAX D - format float (4 bytes per element)
D8	VAX F - format float (8 bytes per element)
G0	General object; a variable in the form used internally by APL (always passed by reference)

The presence of an asterisk '*' before the datatype descriptor indicates that the argument is to be passed by reference; APL will pass the address of the beginning of the data in the array. Otherwise, the argument is passed by value and APL passes the value of the first item of the array. An array of more than one item can only be passed by reference. The presence of an arrow '←' after the datatype descriptor indicates that the value may be modified and will be included in the explicit result returned by the external routine.

res describes the form of the result, if any, returned by the routine. If specified, the routine's result will be returned as the first item of the explicit result returned by the associated APL function. If omitted, the routine's explicit result is discarded

When ⎕NA is used dyadically, the right argument is a character vector containing the specifications for an external routine.

Result: *result* is 1 if dyadic ⎕NA is successful, 0 if it is not. If used monadically, *result* is 3 if *fnname* is the name of a function that has been associated with an external routine. Otherwise, *result* is 0 indicating that *fnname* is not associated with an external routine.

Effect: Creates a locked function in the APL workspace that is associated with the external routine. Using this locked function causes APL to call the routine specified by *fnspec*, passing the pointers (or actual value in the case of scalars) of the arguments supplied to *fnname*. *fnname* is always assumed to be monadic and the number of items in its right argument must match the number of *args* specified in the right argument.

Used monadically, ⎕*NA* simply reports on whether *fnname* is an external routine.

Note: See Chapter 9 of the *APL ⋆PLUS System User's Manual* for more information on using ⎕*NA*.

Warning: ⎕*NA* is experimental in this release of this APL⋆PLUS System. This feature may change or be removed in a future release.

Example:

```
      )CLEAR
CLEAR WS

      3 0 ⎕NA 'VAXCRTL:∆T TIMES(*I4←) I4'
1

      T←∆T ,⊂⍳4
      1⊃T
0                          (Return code.)

      2⊃T
1662 0 0 0                 (CPU time for APL process.)
```

Purpose: Append data to the end of a designated native file.

Syntax: *value* ⎕*NAPPEND tieno*

Arguments:

value any simple, homogeneous APL array
tieno native file tie number

Effect: Appends new data to a native file. Each item of data in the array is written to the native file using the current internal representation of the APL data.

The system function ⎕*DR* should be used to determine the datatype since the display form of the data does not indicate the internal representation. For example, the vector `1 0 1` displays the same whether it is stored internally as Boolean, integer, or floating-point data. Explicit conversion of numeric data may be needed.

The following expressions will convert data to the desired internal representation (note that datatype conversions are not considered part of the APL language and are therefore subject to change in future releases).

Datatype Conversions

Conversion	Expression
Boolean (signal domain error if not Boolean-valued)	*DATA*←1∧*DATA*
Integer	*DATA*←⌊*DATA*+0.5
Integer (from Boolean)	*DATA*←0+*BOOLEAN*
Floating Point	*DATA*←*DATA*÷1

When an APL array is written to a native file, only the data values in the array are stored. Rank, shape, and datatype information are not written to the file.

Caution: ⎕*NAPPEND* is intended for use with the sequential Stream_LF files created with ⎕*NCREATE*. Other types of files may be damaged if ⎕*NAPPEND* is used to write to them.

⎕*NAPPEND* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

DISK ERROR
DISK FULL
DOMAIN ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR

Examples:

```
(⎕VR 'TRI') ⎕NAPPEND ¯27

TEXT ⎕NAPPEND ¯33
```

Purpose: Return classification of a list of identifiers (object names).

Syntax: *result* ← ⎕*NC objlist*

Argument: *objlist* list of object identifiers

objlist contains a list of zero or more workspace identifiers (function, variable, or label names). The argument can be a character vector with one or more names separated by blanks or a character matrix with one name per row.

Result: *result* is a numeric vector of classification codes, one for each name in the argument. Values that can be returned are:

Value	Classification
0	not defined
1	label
2	variable
3	defined function
4	other

A value of 4 indicates that the object identifier is invalid or that it is the name of a system function or variable (that is, it begins with a ⎕).

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
      ⎕NC 'A TRI'
2 3

      ⎕NC 2 3 ρ'A  TRI'
2 3

      ⎕NC '⎕WA'
4
```

Purpose: Create a new native file with specified name and tie the file.

Syntax: *file* ⎕NCREATE *tieno*

Arguments:

file file name
tieno file tie number

file is a character vector containing the name of a valid operating system file. You may prefix the file name with any directory and disk information desired. Native files are created as unblocked Stream_LF files.

tieno must be a negative, integer-valued singleton designating an available file tie number. You cannot have another file currently tied with this number.

Native files are created as unblocked sequential Stream_LF VMS files.

Effect: A new file is created with file name as specified by *file*. The new file is then tied to *tieno*.

Caution: File names ending in `.VF` and `.WS` designate APL component files and workspaces to APL, respectively. We recommend against using `.VF` and `.WS` for any other purpose.

⎕NCREATE as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE NAME ERROR
FILE NAME TABLE FULL
FILE TIE QUOTA EXCEEDED
RANK ERROR
WS FULL
```

Examples:

```
'SAMPLE.C' ⎕NCREATE ¯27
'PRINT' ⎕NCREATE ¯33
'[RIK]EXAMPLE.TXT' ⎕NCREATE ¯25
```

Purpose: Erase a native file.

Syntax: *file* ⎕*NERASE tieno*

Arguments:

file	file name (see ⎕*NTIE*)
tieno	native file tie number

The file described by name (*file*) and by tie number (*tieno*) must be the same file.

Effect: Unties a file and erases it from the disk and directory. All of the data in the file is destroyed.

Caution: ⎕*NERASE* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE NAME ERROR
FILE TIE ERROR
HOST ACCESS ERROR
RANK ERROR
WS FULL
```

Examples:

```
      'MEMO.TXT' ⎕NTIE ¯27
      'MEMO.TXT' ⎕NERASE ¯27

      'SCRATCH' ⎕NTIE ¯33
      'SCRATCH' ⎕NERASE ¯33
```

Name List of Identifiers ⎕*NL*

Purpose: Return a character matrix of function, variable, and/or label identifiers (names).

Syntax:

```
result ←         ⎕NL class
result ← letters ⎕NL class
```

Arguments:

letters beginning letters of identifiers
class classification of identifiers

letters is an optional character vector of letters (blanks are not permitted) that restricts *result* to names whose first letter is in *letters*.

class is an integer vector that determines the class of names produced; the acceptable values are

Value	Identifiers
1	labels
2	variables
3	functions

If more than one value is designated, identifiers defined as belonging to any of those classes are returned. For example, ⎕*NL* 2 3 produces a matrix of names of all variables and functions. The most local definitions of the identifiers are used.

Result: *result* is a character matrix of identifiers with the rows alphabetized.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
        )FNS
TRI      UPDATE   VOID      WITH
WITHOUT  XMIT

        'TX' ⎕NL 3
TRI
XMIT
```

```
      )VARS
ARC TERM XRAY

      'TX' ⎕NL 3 2
TERM
TRI
XMIT
XRAY
```

File Identifications of All Tied Native Files ⎕NNAMES

Purpose: Return the file identifications of all files currently tied with *⎕NTIE*.

Syntax: *result* ← *⎕NNAMES*

Result: *result* is a character matrix that contains one file identification per row and as many columns as are necessary to hold the longest name. The rows of *result* have the same ordering as the result of *⎕NNUMS*.

Directory information is included in the result of *⎕NNAMES* in the same form as it was used when the file tie was established (using *⎕NCREATE* or *⎕NTIE*).

Errors: *WS FULL*

Caution: *⎕NNAMES* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Example:

```
      ⎕NNAMES
[APL.REL1]CHAPTER1
SCRATCH
```

Purpose: Return the file tie numbers of all files currently tied as native files.

Syntax: *result* ← ⎕NNUMS

Result: *result* is a numeric vector of file tie numbers.

result has the same ordering as the rows of the result of ⎕NNAMES.

Errors: WS FULL

Caution: ⎕NNUMS as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples:

```
      ⎕NNUMS
¯27 ¯52 ¯3 ¯37 ¯4

      ⎕NUNTIE ⎕NNUMS

      ⍴⎕NNUMS
0
```

Purpose: Read the current file mode (access permissions) for a native file.

Syntax:
```
result ← ⎕NRDAC file
result ← ⎕NRDAC tieno
```

Arguments:

file native file
tieno native file tie number

The argument identifies the file by file tie number (*tieno*) or by name (*file*). If identified by tie number, the argument must be a negative integer singleton representing a tied native file. If identified by name, a character vector or singleton must be a valid file name.

Result: *result* is an integer scalar representing the current file permissions as the sum of the following values:

Value	Explanation
256	Read permission for owner
128	Write permission for owner
64	Execute permission for owner
32	Read permission for group
16	Write permission for group
8	Execute permission for group
4	Read permission for all others
2	Write permission for all others
1	Execute permission for all others

For a discussion of file permissions, see the documentation supplied with your operating system. Other bits may be set; their effect is presently undefined.

Caution: *⎕NRDAC* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:
```
DOMAIN ERROR
FILE NAME ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR
```

Example:

```
            'FILE' ⎕NCREATE ¯1
            T←3 3⍴¯9↑(32⍴2) ⊤ ⎕NRDAC ¯1
            ' RWX',[1]'OWN' 'GRP' 'ALL',T
     R W X
OWN  1 1 0
GRP  1 0 0
ALL  0 0 0
```

Purpose: Read data from a native file.

Syntax: *result* ← ⎕*NREAD tieno conv count startbyte*

Arguments:

tieno	native file tie number
conv	data conversion to be used
count	number of element of type *conv* to be read
startbyte	starting byte at which to begin reading

The argument is an integer vector of three or four elements (*startbyte* is optional and assumed to be the next byte following the last byte that has been read with ⎕*NREAD*). Tying the file with ⎕*NREAD* sets *startbyte* to 0 (the first byte in the file). *tieno* must be a valid native file tie number (see ⎕*NTIE*) and *conv* must be one of the following conversion types:

⎕*NREAD* Data Conversions

Conv.	Conversion Type
11	Read one bit per element, result is Boolean data
82	Read one byte per element, result is character data
163	Read two bytes per element, result is integer data
323	Read four bytes per element, result is integer data
644	Read eight bytes per element, result is VAX floating-point data

Result: *result* is the data in the file in the datatype specified by *conv*. *result* will be an APL vector with length *count*.

Effect: Copies the data in the file into the workspace and converts it to the specified datatype.

Caution: ⎕*NREAD* is capable of reading on sequential Stream_LF files. Other types of VMS files may not be readable.

Not all eight-byte sequences represent valid floating-point numbers. If arbitrary data is read in with a floating-point conversion, the effect of APL primitives on this data is undefined.

⎕*NREAD* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE INDEX ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR
WS FULL
```

Example:

```
      ⎕NREAD ¯12 82 57 0
THIS FILE CONTAINS SALES DATA FOR
1987.  CREATED 1/26/87.
```

Change the Name of a Native File ⎕*NRENAME*

Purpose: Change the name of a native file or move it to another directory.

Syntax: *file* ⎕*NRENAME tieno*

Arguments:

file	native file name (including directory, if needed)
tieno	tie number

The right argument describes the existing file by tie number (*tieno*). The left argument (*file*) provides the new file name and, optionally, directory information.

Effect: Renames a native file tied to *tieno*. You become the file owner. ⎕*NRENAME* provides the same facility as the DCL command `rename` and you must have the same access permission required to use `rename` in order to use ⎕*NRENAME*.

⎕*NRENAME* cannot replace an existing file and produces a *FILE NAME ERROR* if the target file already exists.

Errors:

DOMAIN ERROR
FILE ARGUMENT ERROR
FILE NAME ERROR
FILE TIE ERROR
HOST ACCESS ERROR
LIBRARY ACCESS ERROR

Caution: ⎕*NRENAME* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Example:

```
'TEST.C' ⎕NTIE ¯1
'[MRVN]WORKING.C' ⎕NRENAME ¯1
```

Purpose: Stores a new value in an existing native file storage space, replacing the data already there.

Syntax: *value* ⎕*NREPLACE tieno startbyte*

Arguments:

value single, homogeneous array
tieno negative file tie number
startbyte starting byte where the new data is to be placed

The right argument designates the file by tie number (*tieno*). It must be an integer two-element vector with the second element positive (*startbyte*).

Effect: Replaces the value of the designated storage space in the file. If the storage from the specified *startbyte* to the end of the file is insufficient for the specified value, the file is extended to accommodate it.

Caution: ⎕*NREPLACE* is intended for use only with sequential Stream_LF files of the kind that are created with ⎕*NCREATE*. Other types of files may be damaged if ⎕*NREPLACE* is used to write to them.

Numeric data is written to file in its present internal representation. Explicit coercion of numeric data to the desired datatype is recommended (see "⎕*NAPPEND* -- Native File Append"). Boolean data is written in whole bytes (writing *n* Boolean values will cause ⌊(*n*+7)÷8 bytes to be replaced in the file). The value of trailing bits in the last byte is undefined.

⎕*NREPLACE* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE INDEX ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR
WS FULL
```

Example:

```
      BLOCK←⎕NREAD ¯33 323 10 1048520

      BLOCK
23 7 1984 ¯22 79 22 48 41 68 82

      BLOCK[3]←1982

      BLOCK ⎕NREPLACE ¯33 1048520
```

Purpose: Report the amount of disk storage occupied by a file.

Syntax:
result ← ⎕NSIZE *file*
result ← ⎕NSIZE *tieno*

Arguments:

file	name of the native file
tieno	native file tie number

The right argument can either be a character vector containing a file name (*file*) or an integer singleton containing a tie number (*tieno*).

Result: *result* is a numeric scalar indicating the total disk storage (in bytes) used by the file.

Caution: ⎕NSIZE as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

```
DOMAIN ERROR
FILE NAME ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR
WS FULL
```

Example:

```
      'PRIMES' ⎕NTIE ¯37

      ⎕NSIZE ¯37
233472
```

Purpose: Set the file mode (access permissions) for a native file.

Syntax: *access* ⎕*NSTAC tieno*

Arguments:

access	access permissions
tieno	native file tie number

access is an integer singleton containing the sum of the file permissions that are to be set for the native file.

Access Permission Value	Explanation
256	Read permission for owner
128	Write permission for owner
64	Execute permission for owner
32	Read permission for group
16	Write permission for group
8	Execute permission for group
4	Read permission for all others
2	Write permission for all others
1	Execute permission for all others

tieno is the tie number of the native file. It must be a negative integer.

Effect: The new permissions are established for the file and take effect immediately.

Caution: ⎕*NSTAC* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

DOMAIN ERROR
FILE ACCESS ERROR
FILE TIE ERROR
LENGTH ERROR
RANK ERROR

Example:

```
           'DEMO' ⎕NTIE ¯1
(+/256 128 32 4) ⎕NSTAC ¯1
```

Purpose: Establish a file tie for a native file.

Syntax: *file* ⎕*NTIE* *tieno*

Arguments: *file* native file name (see section 2-2)
tieno file tie number

file must be a character vector or singleton containing a valid file name. It may optionally be preceded by a directory designation.

tieno is the file tie number to be used and must be a negative, integer-valued singleton not currently in use as a tie number.

Effect: The native file is tied (opened) for reading and writing if the user has both permissions; read-only if the user lacks write permission.

A file that is already tied with ⎕*NTIE* can be re-tied using ⎕*NTIE* without first being untied. The tie number can be the same number or a different number. The only restrictions are that no other file can already be tied with the new tie number and the file cannot be tied to a positive number. This "slippery" tie can be used to verify that a file is tied (without looking up its name in ⎕*NNAMES* and ⎕*NNUMS*).

Caution: ⎕*NTIE* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

```
DISK ERROR
DOMAIN ERROR
FILE ACCESS ERROR
FILE ARGUMENT ERROR
FILE NAME TABLE FULL
FILE NOT FOUND
FILE TIE ERROR
FILE TIE QUOTA EXCEEDED
LENGTH ERROR
RANK ERROR
```

Examples:

```
      'SAMPLE.C' ⎕NTIE ¯1

      [APL.TEST]SAMPLE.C ⎕NTIE ¯1
```

Purpose: Untie native files currently tied.

Syntax: ⎕*NUNTIE tieno1 tieno2 tieno3 ... tienon*

Argument: *tieno* tie numbers

The argument designates the files by tie number. It must be a numeric singleton or vector of zero or more tie numbers. The numbers do not have to be distinct, nor do they need to designate actual tied files.

Effect: Has no response if the argument is empty. If the argument includes tie numbers of tied files, they are closed and associated entries are removed from ⎕*NNAMES* and ⎕*NNUMS*.

Errors:
DISK ERROR
DOMAIN ERROR
RANK ERROR

Caution: ⎕*NUNTIE* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples:

```
      ⎕NUNTIE ¯3 3

      ⎕NUNTIE ⎕NNUMS
```

Protected Copy From Saved Workspace ⎕*PCOPY*

Purpose: Copy APL functions and variables from a saved workspace into the active workspace provided the object does not already exist.

Syntax:

result ← ⎕*PCOPY wsid*
result ← *objlist* ⎕*PCOPY wsid*

Arguments:

wsid	workspace name (see section 2-2)
objlist	list of functions and variables to copy

objlist can be either a character matrix of object names, one name per row, or a character vector with each name separated by one or more blanks.

Result: *result* is an integer vector representing the success or failure of ⎕*PCOPY*. If *objlist* is specified, *result* contains a response code for each object in *objlist*:

Code	Explanation
2	A variable was copied successfully
1	A function was copied successfully
0	No objects copied; none found with the supplied name
¯1	An object with this name already exists in the workspace
¯2	The object was too large to copy given the available free workspace
¯3	The name is defined as a label and cannot be changed
¯4	There is insufficient space in the symbol table to copy this object
¯6	The amount of workspace available is too small to perform the copy

If ⎕*PCOPY* is used without specifying *objlist*, then *result* is empty if all objects of *wsid* were copied successfully.

Effect: Copies objects from the specified workspace (*wsid*) into the local environment of the active workspace unless they would replace any objects by the same name. See the description of *⎕COPY* for a way to copy while replacing any existing objects.

If an unanticipated error occurs, no result is returned.

Copying a function copies only its source form; all compiled code is discarded and *⎕STOP* and *⎕TRACE* settings are cleared in the active workspace.

Errors:

```
DOMAIN ERROR
INSUFFICIENT MEMORY
LENGTH ERROR
RANK ERROR
WS ARGUMENT
WS DAMAGED
WS FULL
WS NOT FOUND
```

Examples:

```
      )VARS
MTRX

      MTRX
1 2
3 4

      )SI
SPND[3]*

      'MTRX XXX DAT SPND' ⎕PCOPY 'WS3'
¯1 1 0 2 ¯3

      )VARS
DAT MTRX

      MTRX
1 2
3 4
```

(Compare to *⎕COPY* which changes the value of *MATRIX*.)

Purpose: Report the current settings of the logical programmable function keys or, optionally, redefines the function key settings.

Syntax: *string* ⎕*PFKEK key*
string ← ⎕*PFKEY key*

Arguments:

string	character sequence associated with a programmable function key
key	character or integer identifying the key

The right argument identifies the keystroke whose programmable value is being queried or set. It is an integer singleton in the range from `0` to `127` or a character singleton from `128↑⎕AV`. For example, the character sequence associated with the *D* key can be referred to either as the character value `'D'` or the integer value `36 ((⎕AV⍳'D')-⎕IO)`.

The optional left argument is used to redefine the character sequence associated with the keystroke. It can be any character scalar or vector. It can also be an integer scalar or vector containing the origin-`0` (`⎕IO←0`) indices of those characters in `⎕AV`.

The total space available for function keys is sufficient to hold `512` characters. The longest possible character sequence is `64` characters.

Result: The explicit result of monadic ⎕*PFKEY* is a character vector containing the current character sequence defined for the key indicated in the right argument. Dyadic ⎕*PFKEY* does not return an explicit result.

Effect: Defines logical programmable function keys that are independent of any physical function keys on a terminal keyboard. The logical function keys are invoked by typing the PF-key keystroke followed by another character. The effect is to substitute the stored character sequence for that key, just as if it had been typed at the keyboard.

If the character sequence contains a newline character (⎕*TCNL*), the effect is equivalent to pressing Return to enter a line of input. A single function key can contain multiple input lines separated by newline characters. If the Escape character ⎕*TCESC* occurs in the sequence, it is sent through to APL as an Escape. One function key cannot invoke another function key.

Default values are defined for each of the ASCII characters. These are listed in Section 5-3 of the *APL**PLUS System User's Manual*.

Caution: ⎕*PFKEY* as described here is specific to this APL*PLUS System. It may be different or absent in other APL*PLUS Systems.

Errors:

```
DOMAIN ERROR
LENGTH ERROR
LIMIT ERROR
RANK ERROR
WS FULL
```

Examples:

```
      ⎕PFKEY 'V'      (Previous definition.)
V
      (')VARS',⎕TCNL) ⎕PFKEY 'V'
      ⎕PFKEY 'V'
)VARS
```

After executing the above example, the sequence ')*VARS*' can be entered as input by pressing PF-key followed by a shift V. Note that *v* and *V* are distinct and can be given different function key definitions.

Printing Precision ⎕*PP*

Purpose: Specify the maximum number of significant digits, or print precision, provided by the system when it displays numeric data.

Syntax:
result ← ⎕*PP*
⎕*PP* ← *number*

Domain: ⎕*PP* can be assigned an integer value between 1 and 18 inclusive. The default value is 10 in a clear workspace.

Effect: The value of ⎕*PP* is used when computing the result of monadic format (⍕) or any system-generated numbers. The system uses up to ⎕*PP* significant digits in the representation of numbers. If a value cannot be represented exactly with ⎕*PP* digits, the result is rounded to ⎕*PP* digits.

Note: ⎕*PP*←18 permits display of full internal precision, with every internal floating-point value distinguishable from its nearest neighbors. The final digit may not be otherwise significant.

Errors:
DOMAIN ERROR
LENGTH ERROR

Examples:

```
      ⎕PP
10
      ÷3
0.3333333333

      2÷3
0.6666666667

      ÷8
0.125               (Requires fewer than ten significant digits.)

      ÷64
0.015625

      ⎕PP←3
      ÷64
0.0156              (Only three significant digits are displayed.)
```

Purpose: The workspace-related system variable ⎕*PR* controls how ⍞ input is affected by the input prompt.

Syntax: *prompt* ← ⎕*PR*
⎕*PR* ← *prompt*

Domain: ⎕*PR* can be assigned a character singleton or empty vector. The default value is ' ' in a clear workspace.

Effect: The value of ⎕*PR* determines how an input prompt, if any, is merged with the result of ⍞ input. If ⎕*PR* is an empty vector, the result of ⍞ input contains the original input prompt, including any changes the terminal user might have made to the prompt. This provides a mechanism for supplying a prompt that the user is expected to modify into an input line.

If ⎕*PR* is a one-element vector, the result of ⍞ input contains the value of ⎕*PR* in every position of the prompt, except those positions that have been modified by the user backspacing into the prompt and performing actions. For more information, see Section 5-1 of the *APL*PLUS System User's Manual.*

⎕*PR* has no effect when ⎕*ARBOUT* ⍳0 is used to prevent the prompt from appearing in ⍞ input. If ⎕*ARBOUT* ⍳0 is used, as is common practice with APL*PLUS Systems, the value of ⎕*PR* is immaterial.

Caution: ⎕*PR*, as described here, is specific to this APL*PLUS System. It may be different or absent in other APL*PLUS Systems.

Errors: *DOMAIN ERROR*

Examples:

```
          ⎕PR←'?'
          ⍞←'PROMPT: ' ⋄ ⎕ARBOUT ⍳0 ⋄ Z←⍞
PROMPT: ANSWER

          Z
ANSWER
```

(Prompt not included.)

(Without ⎕*ARBOUT*.)

```
          ⍞←'PROMPT: ' ⋄ Z←⍞
PROMPT: ANSWER

          Z
????????ANSWER
```

(Prompt replaced with "?".)

```
          ⎕PR←'*' ⋄ ⍞←'PROMPT: ' ⋄ Z←⍞
PROMPT:
```

(User then modifies line before pressing RETURN.)

```
PROMPTLY ANSWER

          Z
******LY ANSWER
```

Purpose: Save the active workspace under program control without halting execution and check that saving the workspace will not replace an existing workspace with the same name.

Syntax:

```
'RESET' ⎕PSAVE wsid
        ⎕PSAVE wsid
```

Arguments: *wsid* workspace identification for the saved workspace (see section 2-2)

The optional left argument, if present, is the character vector containing the value `'RESET'`, indicating that the workspace is to be saved with a clear state indicator.

wsid is a character singleton, vector, or one-row matrix specifying the name of the saved workspace.

Effect: Saves the active workspace without halting execution of the APL statement in which it appears. Monadic `⎕PSAVE` produces a saved workspace with execution suspended at the start of the function line at the top of the state indicator at the time it is called. Dyadic `⎕PSAVE` saves the workspace with a clear state indicator. The system variables `⎕WSID`, `⎕WSTS`, and `⎕WSOWNER`, for both the newly saved and the current workspace, are all changed as a side-effect of `⎕SAVE`.

If a workspace already exists with the supplied name (*wsid*), a `WS ARGUMENT ERROR` is produced. Contrast this to `⎕SAVE` which performs the save by replacing the existing workspace with the new version.

Errors:

```
DISK ERROR
DOMAIN ERROR
LENGTH ERROR
LIBRARY NOT FOUND
RANK ERROR
WS ACCESS ERROR
WS ARGUMENT ERROR
```

Caution: ⎕*PSAVE* as described here is specific to this APL*PLUS System. It may be different or absent in other APL*PLUS Systems.

Examples: The first example shows the use of dyadic ⎕*PSAVE* to save a workspace with a clear state indicator. Note the local ⎕*WSID*.

```
      ∇ INSTALL WSID;⎕WSID
[1]     'RESET' ⎕PSAVE WSID
      ∇
```

The next example uses monadic ⎕*PSAVE* to checkpoint a running application (note the local ⎕*LX*):

```
      ∇ CHECKPOINT WSID;⎕LX
[1]     ⎕LX←'→0' ◊ ⎕PSAVE WSID
      ∇
```

Purpose: Set the maximum number of character positions or columns available for output.

Syntax:
```
result ← ⎕PW
⎕PW ← number
```

Domain: ⎕*PW* can be assigned an integer value between 30 and 255, inclusive. The default value at the start of an APL session is 80.

Effect: The system uses no more than the first ⎕*PW* print positions on each line during output. Output that would extend beyond this number of positions is "folded" onto subsequent lines that are indented six spaces. The display of numeric data is folded between numbers.

The value of ⎕*PW* is used during output from monadic format (⍕), ⎕*FMT*, default output from executing a statement creating an explicit result, and requested output (⎕← or ⍞←). It does not affect the creation of variables.

Errors: *DOMAIN ERROR*
LENGTH ERROR

Examples:
```
      ⎕PW     (Display the value of ⎕PW at session startup.)
80
      ⎕PW←30
      60⍴'⎕'
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
      ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
      ⎕⎕⎕⎕⎕⎕
```

Purpose: Load a workspace under program control without displaying the saved message.

Syntax: *⎕QLOAD wsid*

Argument: *wsid* workspace identifier (see section 2-2)

wsid is a character scalar or vector that specifies the workspace to be loaded.

Effect: Replaces the active workspace with a copy of the contents of the designated workspace. No *SAVED...* message is displayed.

When *⎕QLOAD* is used, the new active workspace begins execution automatically if *⎕LX* is set appropriately in it, giving the effect of continuing a multistep program through two or more workspaces. You can exchange information between the two workspaces by storing data in a file while in one workspace and then reading the data back while in another workspace.

Example:

```
      )CLEAR
CLEAR WS...
      ⎕QLOAD 'STAGE2'        (Note the absence of
                              the SAVED message.)

      ⎕WSID                   (Shows the new
STAGE2                        workspace id.)
```

Purpose: Set the seed value (or random link) used by the pseudo-random number generator.

Syntax:
```
result ← ⎕RL
⎕RL ← number
```

Domain: Any integer from 1 to 2147483646 (¯2+2*31). In a clear workspace, the default value is 16807 (7*5).

Effect: The value of ⎕*RL* is used in computing the result of the roll (monadic ?) and deal (dyadic ?) primitive functions.

⎕*RL* can be assigned a specified value in order to reproduce test results (by resetting ⎕*RL* to the same value each time) or to "randomize" results (by setting ⎕*RL* to an arbitrary value, such as the time of day).

As each pseudo-random number is generated, the seed (⎕*RL*) is used in the computation and is also changed.

Errors:
```
DOMAIN ERROR
LENGTH ERROR
RANK ERROR
```

Examples:
```
      )CLEAR
CLEAR WS
      ⎕RL
16807

      ?3⍴100   (Generate 3 random numbers from 1 to 100.)
50 74 59

      ⎕RL
984943658

      ⎕RL←16807
      ?3⍴100
50 74 59

      ⎕RL
984943658
```

Stop Action ⎕*SA*

Purpose: Specify the action to be taken whenever execution stops for immediate execution input.

Syntax:

```
result ← ⎕SA
⎕SA ← action
```

Domain: The domain for assignment to ⎕*SA* is limited to one of the following character vectors:

```
''
'CLEAR'
'EXIT'
'OFF'
```

Superfluous leading and trailing blanks are ignored; an all-blank vector is treated as empty.

In a clear workspace, the default value of ⎕*SA* is an empty character vector (`''`).

Effect: Specifies the stop action to be taken whenever execution stops for immediate execution input. The effect of each possible value of ⎕*SA* is explained below:

`''`	No special stop action is taken. Execution suspends in the local environment and the system accepts immediate execution input.
`'CLEAR'`	The active workspace is cleared.
`'EXIT'`	The state indicator is stripped back to an environment where ⎕*SA* is not `'EXIT'`. If the value of ⎕*SA* in the resulting environment is `'CLEAR'`, the workspace is cleared.
`'OFF'`	The APL session is terminated with normal untying of any tied files; you are returned to the operating system.

After the stop action has been taken (except for '*OFF*'), the system accepts immediate execution input.

If execution is interrupted at a point where ⎕*SA* has been localized but not assigned, the state indicator is stripped back to an environment where ⎕*SA* is defined.

Errors:

DOMAIN ERROR
RANK ERROR

Examples:

These examples show the effect of each of the settings of ⎕*SA* in the global environment. For illustration, ⎕*SA* is not localized in any of the functions called and no other exception handlers are used.

```
      )WSID
IS PROCESS

      ⎕SI

      ⎕SA←''

      PROCESS 'PAYROLL'
INDEX ERROR                         (An error occurs with
LOOKUP[4] ∧                         ⎕SA set to its default
                                    value.)
      ⎕SI
LOOKUP[4]*                          (Execution is suspended at
DSEARCH[14]                         the point of error.)
XQT[8]
PAYUPDATE[38]
PROCESS[12]

      )RESET
      ⎕SI

      ⎕SA←'EXIT'                    (⎕SA is set to 'EXIT'
                                    in the global environment
      PROCESS 'PAYROLL'             and the function is
                                    executed again.)
INDEX ERROR
LOOKUP[4] ∧                         (The error occurs again and
      PROCESS 'PAYROLL'             the state indicator is
      ⍴⎕SI                          cleared.)
0 0
```

```
      ⎕SA←'CLEAR' ◇ PROCESS 'PAYROLL'
INDEX ERROR
LOOKUP[4]  ⍒
CLEAR WS

      )WSID
IS CLEAR WS
```

(⎕SA is set to 'CLEAR' and the function is executed again. The error occurs once more, but the entire active workspace is cleared.)

Purpose: Saves the active workspace under program control without halting execution.

Syntax:

```
'RESET' ⎕SAVE wsid
        ⎕SAVE wsid
```

Arguments: *wsid* workspace identification for the saved workspace (see section 2-2)

The optional left argument, if present, is the character vector containing the value *'RESET'*, indicating that the workspace is to be saved with a clear state indicator.

wsid is a character singleton, vector, or one-row matrix specifying the name of the saved workspace.

Effect: Saves the active workspace without halting execution of the APL statement in which it appears. Monadic ⎕*SAVE* produces a saved workspace with execution suspended at the start of the function line at the top of the state indicator at the time it is called. Dyadic ⎕*SAVE* saves the workspace with a clear state indicator. The system variables ⎕*WSID*, ⎕*WSTS*, and ⎕*WSOWNER*, for both the newly saved and the current workspace, are all changed as a side-effect of ⎕*SAVE*.

See ⎕*PSAVE* for a way to prevent the save from overwriting an existing workspace.

Errors:

```
DISK ERROR
DOMAIN ERROR
LENGTH ERROR
LIBRARY NOT FOUND
RANK ERROR
WS ACCESS ERROR
WS ARGUMENT ERROR
```

Caution: ⎕*SAVE* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples: The first example shows the use of dyadic ⎕*SAVE* to save a workspace with a clear state indicator. Note the local ⎕*WSID*.

```
    ∇ INSTALL WSID;⎕WSID
[1]   'RESET' ⎕SAVE WSID
    ∇
```

The next example uses monadic ⎕*SAVE* to checkpoint a running application (note the local ⎕*LX*):

```
    ∇ CHECKPOINT WSID;⎕LX
[1]   ⎕LX←'→0' ◊ ⎕SAVE WSID
    ∇
```

Purpose: Return a character matrix representation of the state indicator.

Syntax: *result* ← ⎕SI

Result: *result* is a character matrix containing essentially the same information as displayed by the)SI system command. The names of pendent or suspended functions, quad symbols, and execute symbols may appear in the result. Each row can contain one of the following:

- a quad symbol (⎕), indicating a pending evaluated input request
- an execute symbol (⍎), indicating a pending statement invoked by the execute primitive function
- a function name followed by a bracketed line number, indicating a pendent function
- a function name followed by a bracketed line number and a star, indicating a suspended function

If the state indicator is empty, the result of ⎕SI is an empty matrix of shape 0 0.

Errors: WS FULL

Example:

```
      1 ⎕STOP 'TRI'

      ⍎'TRI 5'

TRI[1]

      ⎕SI
TRI[1]*
⍎

      ⍴⎕SI
2 7
```

Purpose: Return the amount of space used by a list of object identifiers (names).

Syntax: *result* ← ⎕*SIZE idlist*

Argument: *idlist* list of identifiers (functions, variables, or labels)

idlist contains a list of zero or more names that can be represented as a character matrix with one function name in each row or a character vector containing names separated by blanks.

Result: *result* is a numeric vector. Each element of *result* is the amount of space (in bytes) required for the internal representation of the object named in the corresponding position of the argument (Note: Symbol table space is included). Zeros are returned for undefined identifiers, ill-formed names, and system functions and variables. ⎕*SIZE* references the most local definition of each name.

Caution: The value of ⎕*SIZE* cannot be used to reliably estimate the increase in workspace from erasing an object in the workspace. It is possible that multiple variable names refer to the same variable in the workspace (see **Examples:** below). A nested array can also contain multiple items that have the same value and occupy the same storage in the workspace.

Note also that functions can change in size. In particular, a function grows larger when a line in the funciton is executed for the first time and compiled code is generated for that line. Function monitoring (⎕*MF*) also changes the size of a function.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
      ⎕SIZE 'A TRI'
0 144
      A←B←C←2 400 ◊ ⎕SIZE 'A B C'
52 52 52

      ⎕WA
35916

⎕ERASE 'A C' ◊ ⎕SIZE 'A B C'
0 52 0

      ⎕WA          (Workspace available did not increase.)
35916
```

Purpose: Perform a string search, locating all occurrences of a character scalar or vector within another character vector.

Syntax: *result* ← *data* ⎕SS *pattern*

Arguments:

data character vector to be searched

pattern character vector or scalar to be located in *data*

The left argument (*data*) must be a character vector. The right argument (*pattern*) may be a character vector or scalar.

Result: *result* is a Boolean vector of the same length as the left argument, showing the location of all occurrences of *pattern* within *data*. A 1 in the result signifies a match beginning at that position within *data*. All matches are shown, including those that overlap. If *pattern* is empty (' ') *result* is all 1's.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
      'MISSISSIPPI' ⎕SS 'ISSI'
0 1 0 0 1 0 0 0 0 0 0

      'EMPTY MATCHES ALL' ⎕SS ''
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

      CV←'THIS    IS    TOO    SPACED.'
      (~CV ⎕SS '  ')/CV
THIS IS TOO SPACED.
```

Purpose: Set, remove, or report flags for a function.

Syntax: *result* ← ⎕*STOP fnname*
result ← *linenums* ⎕*STOP fnname*

Arguments: *linenums* line numbers to set a stop flag
fnname function name

The optional left argument (*linenums*) is an integer vector or singleton containing the lines of the function *fnname* for which stop flags should be set. Zero and integers that are not line numbers in the specified function are ignored.

fnname is a character vector or singleton containing the name of an unlocked function in the workspace.

Result: *result* is an integer vector of the lines of *fnname* for which prior stops were set.

Effect: Executing ⎕*STOP* has no effect immediately. However, it does affect the executing of other functions in the workspace. If ⎕*STOP* is used to set a stop flag on a function line, it removes all existing stop flags for other lines in the function. Once the stop flag is set, all subsequent executions of the function (*fnname*) are halted prior to executing the flagged lines (*linenums*).

Each time function execution reaches a line that has been set to stop, execution is halted, and the system enters immediate execution mode, preserving the state indicator and all local values and definitions. You can then explore and even alter the local environment before branching (→) back into or out of the suspended function. The resulting ability to observe and alter the local environment at those chosen points in function execution is a valuable aid for debugging a program.

Stop settings are saved and reloaded with a workspace, but they are not copied along with the particular function to which they apply (by ⎕*COPY*,)*COPY*, ⎕*PCOPY*, or)*PCOPY*). Redefining a function with either ⎕*DEF* or ⎕*FX* removes all stop settings from that function. Editing a function line with either ∇ or ⎕*DEFL* removes any setting associated with that line of code. If other lines are inserted or deleted in the function, the setting moves with the line of code thereby changing the line number. Locking a function either by ⍫ or ⎕*LOCK* removes all stop settings in the function.

All stop flags for a function can be cleared with:

```
(⍳0) ⎕STOP fnname
```

Errors:

```
DOMIAN ERROR
RANK ERROR
WS FULL
```

Examples: Given a function:

```
    ∇  R←FIBONA N
[1]    R←1 1
[2]  BACK: R←R,+/¯2↑R
[3]    →BACK×N>ρR
    ∇

          (⍳3) ⎕STOP 'FIBONA'
```

(Empty explicit result means no lines were previously set.)

```
    FIBONA 1
    FIBONA[1]
    R
VALUE ERROR
    R
    ∧

    →1
    FIBONA[2]
    R
1 1

    →2
    FIBONA[3]
    R
1 1 2
```

```
      ⎕SI
      FIBONA[3] *
      ⎕LC
3

      →⎕LC
      FIBONA[2]
      R
1 1 2
```

Workspace Symbols ⎕*SYMB*

Purpose: Return the current number of symbol table entries in the active workspace.

Syntax: *result* ← ⎕*SYMB*

Result: *result* is a two-element numeric vector. The first element is the number of entries reserved in the symbol table of the active workspace. The second element is the number of entries already used in the symbol table of the active workspace.

Returns the same information as)*SYMBOLS*, but without the text.

Note: The symbol table contains entries for all functions, variables, and labels referenced in defined functions and executing statements. The symbol table size increases automatically as needed and can be changed by using the system command)*SYMBOLS*.

Errors: *WS FULL*

Examples:

```
      )CLEAR
CLEAR WS

      ⎕SYMB
500 0

      A←1
      ⎕SYMB
500 1

      )ERASE A
      ⎕SYMB
500 1
```

System Identifier ⎕*SYSID*

Purpose: Return the identification of the APL⋆PLUS System being used.

Syntax: *result* ← ⎕*SYSID*

Result: *result* is a character vector containing the identification of the APL⋆PLUS System being used. All characters are used to identify a system.

Errors: *WS FULL*

Caution: ⎕*SYSID* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Example:

```
      ⎕SYSID
APLPLUSD
```

Purpose: Return identification of the current version of this APL⋆PLUS System.

Syntax: *result* ← ⎕*SYSVER*

Result: The explicit result of ⎕*SYSVER* is a character vector. Its exact form changes from one version of the system to another.

Errors: *WS FULL*

Caution: ⎕*SYSVER* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Example:

```
      ⎕SYSVER
1.0.0 31AUG87 VAX/VMS
```

Purpose: Contain terminal, non-printable characters for easy addition to code. None of the following constants actually produce characters on the screen; rather, they store the terminal control characters often used to affect output.

There are eight terminal control constants:

Terminal Control Constants

Name	Value	⎕*AV*[*n* + ⎕*IO*]
⎕*TCBEL*	bell character	7
⎕*TCBS*	backspace character	8
⎕*TCDEL*	delete character	127
⎕*TCESC*	ASCII escape character	27
⎕*TCFF*	form feed character	12
⎕*TCLF*	linefeed character	10
⎕*TCNL*	new-line character	13
⎕*TCNUL*	null character	0

Effect: Produces the following effects when displayed at a terminal:

⎕*TCBEL* is treated differently depending upon the `atermcap` definition for the terminal in use. The effect is either to produce a beep sound or to "flash" the terminal screen by briefly switching to reverse video and back again.

Note that on some terminals the sound produced by the "BEL" control code will last only one character-time (1/30th of a second at 30 CPS). Thus, several bell characters may need to be separated by one or more null characters (⎕*TCNUL*) to be heard as distinct sounds.

⎕*TCBS* moves the cursor one position to the left so that the next character to be displayed will overstrike the preceding character.

⎕*TCDEL* is transmitted to the terminal as an ASCII DEL character (decimal 127). On the APL⋆PLUS system for the VAX, ⎕*TCDEL* is usually displayed as a blot.

⎕*TCESC* is transmitted to the terminal as the ASCII ESC (decimal 27). Many devices recognize the ESC character as the start of a special control sequence.

⎕*TCFF* clears the current window (see ⎕*WINDOW*) when transmitted to the terminal and places the cursor in the upper left corner.

When ⎕*TCFF* is transmitted to some hardcopy printers or terminals, the paper is ejected to the start of the next page (form feed).

⎕*TCLF* varies with the device to which it is transmitted. When displayed on some terminals and printers, it causes the screen or paper to advance one line while keeping the cursor in the same column position as on the previous line. On other terminals and printers, however, it may be treated as a ⎕*TCNL* or ignored completely.

⎕*TCNL* moves the cursor to the first position of the next line.

⎕*TCNUL* does not move the cursor, but causes the terminal to pause in output for one character-time (1/30th of a second on a 30 CPS terminal).

Example:

```
            B←'DOWN',⎕TCLF,'WE'
            C←'⎕TCBS,'__',⎕TCLF,'GO'
            A←B,C
            ⍴A
      13

            A
      DOWN
          WE
            GO
```

Trace Function Execution ⎕*TRACE*

Purpose: To aid you in debugging a program by allowing lines of functions to be flagged for diagnostic output when next executed.

Syntax:

result ← ⎕*TRACE fnname*
result ← *linenums* ⎕*TRACE fnname*

Arguments:

linenums integer line numbers to trace
fnname function name

The optional left argument (*linenums*) is an integer vector or singleton indicating which lines of the function named in the right argument are to be traced. They will continue to be traced until a later execution of ⎕*TRACE* on the function name in this workspace resets the lines. Zero and integers that are not line numbers in the specified function are ignored.

fnname is a character vector or singleton containing the name of an unlocked function in the workspace.

Result: *result* is an integer vector of the lines of *fnname* for which tracing was in effect until this execution of ⎕*TRACE*.

Effect: ⎕*TRACE* does not trace output as its direct result. Instead, it flags lines in a function so that, in future execution, diagnostic output is produced.

During execution of a function that is being traced, the system displays the final value calculated in each statement on each traced line. The value appears after the function's name and the bracketed line number or after a ◊. This is true even for values that would not display in normal (untraced) execution. In the case of a branch in the function, a → is displayed before the value to which the function branched.

The resulting ability to observe the sequence in which the lines are executed and the internal values of statements (those not normally displayed) is a valuable aid in debugging a program.

Trace settings are saved and reloaded with a workspace, but they are not copied along with the particular function to which they apply (by ⎕*COPY*,)*COPY*, ⎕*PCOPY*, or)*PCOPY*). Redefining a function with either ⎕*DEF* or ⎕*FX* removes all trace setting from that function. Editing a function line with either ∇ or ⎕*DEFL* removes any setting associated with that line of code. If other lines are inserted or deleted, the setting moves with the line of code, thereby changing the line number.

Locking a function with either ⍫ or ⎕*LOCK* removes all trace settings in that function. Execution of ⎕*TRACE* removes any existing trace flags previously set, so

(⍳0) ⎕*TRACE* *fnname*

can be used to remove all trace settings for a function.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
      ∇  RESULT ← GO
[1]      ⍞←'NEW LINE DURING TRACE DESPITE ⍞
         OUTPUT!'
[2]      RESULT ← 1
[3]   LABEL: RESULT ← (0,RESULT)+RESULT,0
[4]      → LABEL × 4 > ⍴RESULT

         (⍳5) ⎕TRACE 'GO'
```

(An empty explicit result means no lines were set.)

```
         GO
NEW LINE DURING TRACE DESPITE ⍞ OUTPUT!
GO[2] 1
GO[3] 1 1
⋄→3
GO[3] 1 2 1
⋄→3
GO[3] 1 3 3 1
⋄→0
1 3 3 1
```

(The explicit result of *GO*.)

Purpose: Return the current date and time of day as represented by the system clock.

Syntax: *result* ← ⎕*TS*

Result: *result* is a seven-element numeric vector containing the following information:

[1] year
[2] month
[3] day
[4] hour
[5] minute
[6] second
[7] millisecond

⎕*TS* relies on the system clock maintained by the operating system for its time measurement. The seventh element of the result is included for consistency with other APL*PLUS Systems. However, the computer system's clock precision determines if this element provides useful information.

The first three elements in the result of ⎕*TS* always indicate a date, and the last four elements always indicate a time of less than 24 hours.

Errors: *WS FULL*

Example:

```
      ⎕TS
1986 9 8 19 12 7 0
```


Purpose: Return the number of users.

Syntax: *result* ← ⎕*UL*

Result: *result* is a numeric scalar containing the number of users currently signed on to the system.

Note: You can use)*CMD* or ⎕*CMD* to execute the DCL command `show users` to obtain detailed information about users signed on to the system.

Errors: *WS FULL*

Example:

```
      ⎕UL
5
```

User Identification ⎕USERID

Purpose: Return your VMS logon identification.

Syntax: *result* ← ⎕USERID

Result: *result* is an eight-element character vector containing your logon identification. The name is left justified and padded with blanks.

Caution: ⎕USERID may return a different number of elements on other APL★PLUS Systems.

Errors: *WS FULL*

Example:

```
      ⎕USERID
MYERS
      ⍴⎕USERID
8
```

Purpose: Provide a validity check on an input character vector (often used in conjunction with ⎕*FI*).

Syntax: *result* ← ⎕*VI data*

Argument: *data* character data

data is a character singleton or vector of data.

Result: *result* is a Boolean vector with 1's in the positions where groups of characters represent well-formed numbers, and 0's where they do not.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Examples:

```
      A←'666 ¯1.20    .1 314159E¯5'
      ⎕FI A
666 ¯1.2 0.1 3.14159

      ⎕VI A
1 1 1 1

      ⎕FI 'ANSWER: 666'
0 666

      B←'ANSWER IS 666 LBS.'
      ⎕FI B
0 0 666 0

      ⎕VI B
0 0 1 0

      (⎕VI B)/⎕FI B
666
```

Visual Representation of a Function ⎕*VR*

Purpose: Return the visual representation of a function as a character vector.

Syntax: *result* ← ⎕*VR fnname*

Argument: *fnname* function name

fnname is a character scalar or vector containing one function name.

Result: *result* is a character vector. It is a visual representation of the function with bracketed line numbers and embedded newline characters separating the character representations of the successive lines of the function. The explicit result is not affected by ⎕*PW*.

If *fnname* is a character singleton or vector but does not contain the name of an unlocked function, *result* is an empty vector.

Errors:

```
DOMAIN ERROR
RANK ERROR
WS FULL
```

Example:

```
      ρQ←⎕VR 'TRI'
81

      Q
      ∇ TRI N;A
[1]   ⎕←A←,1
[2]   →(N<ρA)/0 ◊ ⎕←A←(0,A)+A,0 ◊ →⎕LC
      ∇
```

Work Area Available ⎕WA

Purpose: Return the current amount of work area available in the active workspace (in bytes).

Syntax: *result* ← ⎕WA

Result: *result* is a numeric scalar whose value is the current number of unused bytes in the active workspace.

Errors: *WS FULL*

Example:

```
      )WSID
IS OFFICE

      ⎕WA
14372
```

Get Window Data ⎕*WGET*

Purpose: Read the characters or attributes or both from a specified (or the current) screen window.

Syntax:
result ← ⎕*WGET rtype*
result ← *wspec* ⎕*WGET rtype*

Arguments:

wspec	window specification
rtype	type of result desired

The optional left argument (*wspec*) is a specification of a window to be used during this one operation. If *wspec* is not specified, the entire window is used.

rtype is an integer singleton with a value of 1, 2, or 3. It affects the type of result produced.

Value	**Result**
1	A character matrix containing the characters visible in the window (without their display attributes).
2	An integer matrix containing the attribute values associated with each character position in the window (the attribute values are given in the table below).
3	A rank 3 character array where *result*[; ; 1] contains the characters displayed on the screen (the same as the result if *rtype*=1). *result*[; ; 2] contains the attributes coded as characters by ⎕*AV*[*att*+⎕*IO*] where *att* is the same as the integer result when *rtype*=2.

Result: *result* is the data requested by the specified *rtype* from the specified window as a matrix (or for *rtype* type 3, a three-dimensional array with last coordinate of length 2).

Attribute Values:

The conventional values used for display attributes in this APL⋆PLUS System are:

Attr.	Description
0	default display form for the terminal
1	reverse video
2	alternate intensity (brighter or dimmer than usual)
4	blinking
8	underlined (unrelated to APL's underscored alphabet)

A combination of atttributes is represented by the sum of their values. For more details on the logical nature of these attributes, see Chapter 1 of the *APL⋆PLUS System User's Manual.*

Effect: Retrieves the data specified by *rtype* and *wspec* from the display buffer and returns it as a result.

Caution: ⎕*WGET* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Errors:

```
DOMAIN ERROR
LENGTH ERROR
RANK ERROR
WS FULL
```

Examples:

Obtain the characters on the top row of the screen.

```
      TOP ← 0 0 1 80 ⎕WGET 1
```

Save the entire screen including its current attributes.

```
      SCREEN ← ⎕WINDOW ⎕WGET 3
```

Purpose: Report the dimensions of the terminal screen or window. Its value is a vector containing the first row and first column of the window followed by the window size (number of rows and columns).

Syntax: *value* ← ⎕*WINDOW*

Domain: *value* is limited by the physical device. It is a numeric vector containg the first row and first column of the window followed by the window size (number of rows and columns).

Effect: The value of ⎕*WINDOW* is used in connection with ⎕*CURSOR*, which is relative to the upper-left corner of the current window, to determine the absolute screen location for output.

When normal screen input or output is displayed, it is limited to the rectangle on the screen described by ⎕*WINDOW*. The first two elements of the current value are taken as the row and column numbers of the upper-left corner of the window (in origin 0). The last two elements are taken as the window size -- the number of rows and columns contained within the window.

The number of rows and columns of the terminal screen is derived from the specifications in the atermcap file.

Caution: ⎕*WINDOW* as described here is specific to this APL*PLUS System. It may be different or absent in other APL*PLUS Systems. In particular, some APL*PLUS Systems allow ⎕*WINDOW* to be set by the users. This system produces a *NONCE ERROR* instead.

Errors: *NONCE ERROR*

Example:

```
      ⎕WINDOW
0 0 23 80
```

Purpose: Replace the characters or attributes on the screen or window.

Syntax:

```
      ⎕WPUT data
wspec ⎕WPUT data
```

Arguments:

wspec	window specification
data	characters or attributes to be placed on the screen

The optional left argument (*wspec*) specifies the region on the screen to display the data. If *wspec* is not supplied, the entire window (⎕*WINDOW*) is used.

The right argument (*data*) is the data to be placed on the screen. It must be a character array with rank 3 or less or a numeric array of rank 2 or less (matrix). Its shape should be either a singleton or match the window size (¯2↑⎕*WINDOW* or ¯2↑*wspec*) to prevent it from being reshaped to fit the specified window.

Effect: ⎕*WPUT* changes the screen display. The actual effect depends greatly on the shape of *data*.

Value	**Effect**
Character singleton supplied.	Fill region of screen with character
Numeric singleton specified	Change attribute of region with attribute (see below).
Character matrix	Fill region of screen with text supplied.
Numeric matrix position	Change attributes of each character with attribute specified in *data*.
3-dimensional character array	Fill the region of the screen with *data* [: : 1] and then change the attributes with those specified by *data* [: : 2].

Attribute Values:

The conventional values used for display attributes in this APL⋆PLUS System are:

Attr.	Description
0	default display form for the terminal
1	reverse video
2	alternate intensity (brighter or dimmer than usual)
4	blinking
8	underlined (unrelated to APL's underscored alphabet)

A combination of atttributes is represented by the sum of their values. For more details on the logical nature of these attributes, see Chapter 5 of the *APL⋆PLUS System User's Manual.*

Errors:

```
DOMAIN ERROR
LENGTH ERROR
RANK ERROR
```

Caution: ⎕*WPUT* as described here is specific to this APL⋆PLUS System. It may be different or absent in other APL⋆PLUS Systems.

Examples:

```
      ⎕WPUT WI
```

Fill top row of screen with "⋆".

```
      0 0 1 80 ⎕WPUT '⋆'
```

Put the contents of the current window into reverse video.

```
      ⎕WPUT 1
```

Clear screen except for small portion that is preserved.

```
      SCR←5 10 6 20 ⎕WGET 3

      ⎕TCFF
      5 10 6 20 ⎕WPUT SCR
```

Purpose: Store the active workspace identification.

Syntax: *wsid* ← ⎕*WSID*
⎕*WSID* ← *wsid* (see section 2-2)

Domain: ⎕*WSID* contains any well-formed workspace name optionally preceded by directory or library designation. In a workspace, ⎕*WSID* is a character vector containing the workspace identification (see section 2-2 for a description of a valid workspace identification). In a clear workspace, ⎕*WSID* is an empty vector.

Result: When referenced, ⎕*WSID* returns the workspace identification or an empty vector if the workspace name is *CLEAR WS*. The actual format depends upon whether the system is in library mode or directory mode.

If the system is in directory mode, ⎕*WSID* is a character vector containing the name left justified. If the system is in library mode, ⎕*WSID* is a 22-element character vector containing the workspace identification. The 22-element vector has the following format:

Elements 1-10	Library number, right justified
Element 11	Blank
Elements 12-22	Workspace name, left justified

Errors: *WS FULL*

Examples:

```
      )WSID
IS ANSWER

      ⎕WSID ⋄ ⍴⎕WSID
ANSWER
6

                              (Switch to library mode.)
      ⎕LIBD '11 [APL.WS]
      ⎕WSID←'11 ANSWER'
      ⎕WSID ⋄ ⍴⎕WSID
        11 ANSWER
22
```

Purpose: Return a character matrix listing all the workspaces in the designated library, even if the user has no access to them.

Syntax:

result ← ⎕*WSLIB dir*
result ← ⎕*WSLIB lib*

Arguments:

dir	directory to be searched
lib	library to be searched

The argument designates the directory or library whose workspaces are to be listed. It is either a character singleton or vector containing the directory name (*dir*) or a positive integer associated with a directory in ⎕*LIBS*. An empty vector specifies the current working directory.

Result: The form of the explicit result of ⎕*WSLIB* depends upon the form of the argument. If a path name is supplied, the result is a matrix of workspace names, left-justified. The number of columns is the length of the longest workspace name in the list.

If the argument is a numeric library number, the result is a 22-column character matrix that contains one workspace identification in each row. The columns in the result are defined as follows:

Column 1-10	Library number, right-justified
Column 11	Space
Columns 12-22	Workspace name, left-justified

In either form, the ordering of the rows (workspace identifications) is alphabetic.

Note:)*WSLIB* produces the same list of workspaces, but they are listed in multiple columns to save lines on the screen and are listed without library numbers.

Errors:

```
DISK ERROR
DOMAIN ERROR
LENGTH ERROR
LIBRARY NOT FOUND
WS FULL
```

Examples:

```
      ⎕WSLIB '[APL.REL1]'   (In directory mode.)
FEBRUARY
JANUARY
MARCH

                            (Switch to library mode.)
      ⎕LIBD '1 [APL.WSS]'
      ⎕WSLIB 1              (In library mode.)
   1 PERSONS
   1 SALES

      ⍴⎕WSLIB 1
2 22

      ⍴⎕WSLIB '[APL.WSS]'
3 8
```

Workspace Owner ⎕*WSOWNER*

Purpose: Return the user number of the user who last saved the current workspace.

Syntax: *result* ← ⎕*WSOWNER*

Result: *result* is an integer scalar representing the user number (1↑⎕*AI*) of the user who last saved the workspace.

In a clear workspace, *result* is 0.

Errors: *WS FULL*

Example:

```
      ⎕WSOWNER
3720
```

Workspace Size ⎕*WSSIZE*

Purpose: Return the size of the active workspace in bytes.

Syntax: *result* ← ⎕*WSSIZE*

Result: *result* is a numeric scalar containing the total size of the active workspace, including the space used by APL objects, the symbol table, and unused storage (⎕*WA*). In this APL★PLUS System, ⎕*WSSIZE* is determined by the initial workspace size specified in the command line when APL is invoked from the operating system or by the size specified with)*CLEAR*. For more information, see Chapter 1 of the *APL★PLUS System User's Guide*.

Errors: *WS FULL*

Examples:

```
      ⎕WSSIZE
102483

      ⎕WA
26782

      ⎕WSSIZE-⎕WA
67701                 (The approximate number of bytes
                      needed to store this workspace on disk.)
```

Workspace Timestamp ⎕*WSTS*

Purpose: Return the save time of a loaded workspace or the time of the most recent *)SAVE* or *)CLEAR* performed on the active workspace.

Syntax: *result* ← ⎕*WSTS*

Result: *result* is a numeric scalar containing the time of the most recent *)SAVE* or *)CLEAR* performed on the active workspace. The time code is in microseconds since 00:00 on 1 January 1900.

Errors: *WS FULL*

Examples:

```
      )LOAD MYWS
MYWS SAVED 15:14:00 07/14/87

      ⎕WSTS
2.76226284E15

      )COPY DATES FTIMEFMT
SAVED 15:17:21 08/07/87

      FTIMEFMT ⎕WSTS
15:14:00.000 07/14/87
```

Load a Workspace, Bypassing the Latent Expression ⎕XLOAD

Purpose: Replace the active workspace by loading the designated workspace (under program control), but without executing the latent expression (⎕*LX*).

Syntax: ⎕*XLOAD wsid*

Argument: *wsid* workspace identification (see section 2-2)

The argument is a character scalar or vector that specifies the workspace to be loaded. If the directory name or library number is omitted, your default library is assumed.

Effect: Loads the specified workspace, making it the new active workspace. ⎕*WSID* changes and ⎕*LX* is not executed.

Errors:
```
DISK ERROR
DOMAIN ERROR
LENGTH ERROR
LIBRARY NOT FOUND
RANK ERROR
WS ARGUMENT ERROR
WS NOT COMPATIBLE
WS NOT FOUND
WS TOO LARGE
```

Examples:
```
      ⎕XLOAD 'STAGE2'
STAGE2 SAVED 19:41:55 10/19/87

      ⎕XLOAD 'TESTWS'
TESTWS SAVED 19:42:07 03/19/87
```

(Switch to library mode.)
```
      ⎕LIBD '1234 [APL.REL1]'

      ⎕XLOAD '1234 TESTWS'
TESTWS SAVED 23:24:25 01/20/87
```

Purpose: Initiate, communicate with, send interrupts to, or shut down a concurrent VMS process. Identical facilities are provided by ⎕*XP*2, ⎕*XP*3, ⎕*XP*4, and ⎕*XP*5, permitting as many as five independent concurrent processes. See Chapter 7 in the *APL*PLUS System User's Manual* for more information.

Syntax:

result ← ⎕*XP*1 *process*
result ← ⎕*XP*1 *intnum*
result ← *array* ⎕*XP*1 *array*

Arguments:

process	name of a VMS `.exe` file containing the program to be run as a concurrent process
intnum	integer to be signaled to the concurrent process
array	any simple homogeneous APL array

The left and right arguments, when both are present, can be any simple homogeneous APL array to be passed to the external process associated with ⎕*XP*1. Only the dyadic use of ⎕*XP*1 passes input to the external process, which must previously have been initiated by a monadic use of ⎕*XP*1.

The right argument to ⎕*XP*1 when there is no left argument (a monadic use of ⎕*XP*1) must be:

- a character vector representing the name of the executable module to be activated as a subprocess child of the APL process and associated with ⎕*XP*1 for further communications
- an empty character vector (`''`) to inquire what process is currently associated with ⎕*XP*1
- an integer-valued singleton representing an interrupt to be signaled to the external process using the "kill" system call. Note that in Release 1 of the APL*PLUS System, 9 is the only interrupt supported, and it teminates the external process.

Result: The explicit result of a dyadic use of ⎕*XP*1 can be any simple homogeneous APL array created and returned by the external process.

The explicit result of a monadic use of ⎕XP1 varies according to the nature of the argument that produced it:

⎕XP1 Arguments	Results
'' (empty character vector)	the character argument previously used to associate an external process with XP1
a character vector containing the process name	a positive integer representing the VMS process ID of the process started up, if successful; a two-element vector consisting of a 0 as the first element and the VMS System Service Condition Value as the second element, if unsuccessful; or ¯2 if a process is already running for this ⎕XPn
an interrupt	an integer showing that number the specified interrupt was judged valid (=0) or invalid (=¯1)

Effect: Varies with the nature of the argument or arguments used with it.

Monadic ⎕XP1 used with a character vector naming a `.exe` file containing a program:

- sets up a VMS subprocess running that program
- sets up a VMS mailbox to communicate with that process
- associates that process with ⎕XP1 so that ⎕XP1 can be used as a means of communicating with that process
- returns the process ID number as *result*; indicating that the program has been successfully started, or returns a zero if it has not been successfully started

Used with an empty character vector, monadic ⎕XP1 returns the process name used to initiate the external process currently associated with ⎕XP1. If no process is currently associated with ⎕XP1, *result* is an empty character vector.

Used with an integer-valued singleton (*intnum*), monadic ⎕XP1 sends that value as an interrupt to the child process using the VAX 'C' "kill" system call (see `kill(2)` and `signal(2)` in your VAX C reference manual) and returns a zero if the interrupt is valid or a ¯1 is the interrupt is not valid. Interrupt 9 is the only valid VMS interrupt supported in Release 1 of the APL⋆PLUS System.

Used with two arguments, dyadic ⎕XP1 transmits first the left then the right argument (complete with their internal headers) through the mailbox to the external process. The output of the external process is then read from the mailbox, checked to assure that it is well formed, and returned as the explicit result.

Warning: ⎕XP*n* is experimental in Release 1 of the APL⋆PLUS System. This feature may change or be removed in a future release.

Errors:

DOMAIN ERROR
FILE ARGUMENT ERROR
FILE NOT FOUND
FILE TIE QUOTA EXCEEDED
HOST ACCESS ERROR
NO PROCESS RUNNING
RANK ERROR
WS FULL
⎕XP1 ERROR n
⎕XP1 INTERRUPT

The external process can also return error codes that are interpreted through the list in `ERRMACRO.H` distributed with the APL⋆PLUS system. These error messages are presented as if the errors were signaled by APL itself, using the spelled out message rather than the error code number. The messages are not part of the APL session, however, and will disappear when you press the Refresh key.

In addition, the external process can cause arbitrary error reports to appear on the screen by using `fprintf` with `stderr`. The file must be created in the external process before it can be used for debug information. See Chapter 7 of the *APL*PLUS System User's Manual* for details and solutions.

Example:

```
        ⎕XP1 ''                  (No process associated
                                 with ⎕XP1.)
        ⎕XP1 'VTOM.EXE'          (Initiate a process.)
204                              (Process ID number.)

        ⎕XP1 ''
VTOM.EXE

        Z←'' ⎕XP1 'ONE TWO THREE'
                                 (Pass data to external process.)
        Z
ONE                              (Result returned by VTOM
TWO                              process.)
THREE

        ⍴Z
3  5

        ⎕XP1 9                   (Terminate process.)
0

        0 = ⍴⎕XP1 ''             (⎕XP1 now available to start
1                                another process.)
```

UTILITY FUNCTIONS

Chapter 4
Workspace Functions

4-1 Introduction

This chapter describes in detail some of the functions in the workspaces supplied with your APL★PLUS System. They are listed alphabetically. Each description contains:

- the function name
- the workspace containing it
- the syntax of the function
- a description of the arguments, result, and effect of the function.

Most of the descriptions also show at least one example of the function.

The following conventions are used in the detailed function descriptions for the *DATES* workspace:

date an integer array whose last dimension is 3 (3 = ¯1↑⍴*date*)
ts an integer array whose last dimension is 7 (7 = ¯1↑⍴*ts*).

Typically, *date* is a vector in 3↑⎕TS form:

date[1] two- or four-digit year (1900s are assumed for two-digit representations)
date[2] an integer (1 to 12) representing the month
date[3] an integer (1 to 31) representing the day of the month.

Typically, *ts* is a vector in 7↑⎕TS form:

ts[1] two- or four-digit year (1900s are assumed for two-digit representations)
ts[2] an integer (1 to 12) representing the month
ts[3] an integer (1 to 31) representing the day of the month
ts[4] an integer (0 to 23) representing the hour
ts[5] an integer (0 to 59) representing the minute

ts[6] an integer (0 to 59) representing the second
ts[7] an integer (0 to 999) representing the millisecond.

ts can also be a matrix with one date or time per row.

4-2 Detailed Descriptions

CALEN *DEMOAPL*

Syntax: *CALEN year*

Displays the 12 monthly calendars for the specified year.

```
      CALEN  1987

This function will now print out a
calendar for 1987.  You can turn the
printer on and align the paper before
pressing Enter.

       CALENDAR FOR 1987

|---------------------------------------|
|             JANUARY 1987              |
| SUN   MON  TUES   WED  THUR   FRI   SAT |
|                           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 |
|---------------------------------------|
|            FEBRUARY 1987              |
| SUN   MON  TUES   WED  THUR   FRI   SAT |
|   1     2     3     4     5     6     7 |
|   8     9    10    11    12    13    14 |
|  15    16    17    18    19    20    21 |
|                                         |
|       (This table has been abbreviated.) |
```

CALENDAR *DEMOAPL*

Syntax: *CALENDAR month year*

Displays a calendar for the *month* and *year* requested.

```
        CALENDAR  7  1987

              JULY 1987
SUN    MON  TUES    WED  THUR    FRI    SAT
                      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
```

CENTER *FORMAT*

Syntax: *result* ← *formatstring* *CENTER* *title*

result is a one-row matrix with appropriate blanks added to the title to center it in the width specified by *formatstring*, a character vector. Usually, it is in the same format string that was used to produce a report with ⎕*FMT*, but it can be any format string with an appropriate width, or it can be the result of *RWTD*. The title is centered within the width of the format string when it is displayed, and it is truncated on the right if it is too long. *title*, a character vector, is the desired title.

In the following example, a report is set up with ⎕*FMT* and then titled with *CENTER*.

```
        F ← '6A1,T10,I5,T17,P<$> CF11.2'
        NAMES ← 3 6ρ'JAMES ROGAN TAYLOR'
        SALES ← 36.5 30 67.13
        VALUES ← 981.24×SALES

        REP1←F ⎕FMT NAMES SALES VALUES
        REP1
JAMES         37    $35,815.26
ROGAN         30    $29,437.20
TAYLOR        67    $65,870.64
```

```
        T←'ANNOUNCEMENT OF NEW DATA'
        CTITLE←F CENTER T
        '' ◊ CTITLE ◊ '' ◊ REP1

  ANNOUNCEMENT OF NEW DATA

JAMES        37    $35,815.26
ROGAN        30    $29,437.20
TAYLOR       67    $65,870.64
```

COLNAMES — *FORMAT*

Syntax: *result* ← *formatstring* *COLNAMES* *columnnames*

result is a one-row character matrix with the column names from the right argument lined up appropriately to be used as column headers for a report. *formatstring* is usually the format string that was used to produce the report with ⎕*FMT*. *columnnames* is a character vector containing column names separated by a delimeter character. The first character in *columnnames* becomes a separator character for each new column heading. Each time the function reaches a separator, it skips to the next field produced by an editing format phrase to display the next string of text. In the following example, | is the separator and *FIRST*, *SECOND*, and *THIRD* are column names.

```
'|FIRST|SECOND|THIRD'
```

Column names for numeric fields are right-justified, while column names for character fields are left-justified. The width of the column name for a numeric field is limited by the width of the corresponding format phrase. A column name for character data may extend into a *text* phrase immediately to the right.

```
        T← '∘NAME∘SALES∘VALUE'
        CNAME ← FSTR1 COLNAMES T

        CNAME1 ◊ REP1

NAME        SALES         VALUE
JAMES          37    $35,815.26
ROGAN          30    $29,437.20
TAYLOR         67    $65,870.64
```

```
      T← '∘----∘-----∘-----'
      CNAME2 ← FSTR1 COLNAMES T

      CNAME1 ⋄ CNAME2 ⋄ REP1

NAME        SALES          VALUE
----        -----          -----

JAMES          37     $35,815.26
ROGAN          30     $29,437.20
TAYLOR         67     $65,870.64
```

COMB   *DEMOAPL*

Syntax: *result* ← *n* *COMB* *m*

result is a table containing all the possible sets of *n* items chosen from a set of *m* items. There are (*n* ! *m*) such possible sets.

```
      3!5
10
      3 COMB 5
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

DATEBASE   *DATES*

Syntax: *result* ← *DATEBASE* *date*

Returns an integer array of shape ¯1↓ρ*date* representing the number of days elapsed since January 1, 1900. Elements of *result* may be negative. In the example, we find the number of days between February 28, 1972, and March 2, 1972. (The year 1972 was a leap year.)

```
      DATEBASE 2 3ρ72 3 2 72 2 28
26358 26355

      26358-26355
3
```

DATECHECK *DATES*

Syntax: *result* ← *DATECHECK* *date*

Returns a Boolean vector of shape ¯1↓ρ*date*, in which 1s indicate valid dates. In the example, February 29, 1976, is a valid date (since 1976 is a leap year), but February 29, 1977, is not.

```
      DATECHECK 2 3ρ76 2 29 77 2 29
1 0
```

DATEOFFSET *DATES*

Syntax: *result* ← *days* *DATEOFFSET* *date*

Adds the number of days in *days* to each date in *date* and returns the new dates. The *result* is the same format and shape as *date*. The *days* argument is a vector or scalar with one element for each row in *date*. In the example, 30, 60, and 90 days are added to November 15, 1986. The resulting dates are December 15, 1986; January 14, 1987; and February 13, 1987.

```
      30 60 90 DATEOFFSET 86 11 15
1986    12     15
1987     1     14
1987     2     13
```

DATEREP *DATES*

Syntax: *date* ← *DATEREP* *elapsed*

The *elapsed* argument is the number of days since January 1, 1900. *DATEREP* returns a date in ⎕*TS* format.

```
      DATEBASE 87 5 27
31922
      DATEREP 31922
1987 5 27
```

DATESPELL *DATES*

Syntax: *result* ← *code* DATESPELL *ts*

Returns *ts* formatted according to *code*. The *ts* argument need not include hour, minute, second, or millisecond although hour is required if you use the hour offset. *code* is a one- or two-element vector in which the first element is the display style and the second (optional) element is an hour offset. If omitted, it is assumed to be 0. The following table shows the available styles.

Code	Result
0	1 MAR 1987
1	MAR 1, 1987
2	1 MARCH 1987
3	MARCH 1, 1987
4	TUE 1 MAR 1987
5	TUE, MAR 1, 1987
6	TUESDAY 1 MARCH 1987
7	TUESDAY, MARCH 1, 1987

The preceding codes display time in AM/PM style; add 8 to each code to display time in 24-hour style (military time). For example, code 15 is the same as code 7, but time will be displayed in 24-hour style.

```
      0 DATESPELL 1987 12 31 12
31 DEC 1987  12 N

      5 DATESPELL TS←78 1 1 2 10
SUN, JAN 1, 78  2:10 AM

      5 ¯3 DATESPELL TS (Change to Pacific time.)
SAT, DEC 31, 87  11:10 PM
```

DAYOFWK *DATES*

Syntax: *result* ← *DAYOFWK date*

Returns the the day of the week (1 through 7). The *result* will have one element for each date in *date*. In the example, we find that January 1, 1975, was a Wednesday; January 1, 1976, was a Thursday; and January 1, 1977, was a Saturday.

```
        DAYOFWK 3 3ρ75 1 1 76 1 1 77 1 1
4 5 7
```

DAYOFYR *DATES*

Syntax: *result* ← *DAYOFYR date*

Returns the day of the year (1 through 366). *result* will have one element for each date in *date*.

```
        T←76 12 31 77 1 3 77 12 31
        DAYOFYR 3 3ρT
366 3 365
```

DAYSDIFF *DATES*

Syntax: *result* ← *date1 DAYSDIFF date2*

Returns an integer array containing the difference in days between the corresponding dates supplied in the arguments.

```
        L←2 3ρ72 3 2 73 3 2
        R←2 3ρ72 2 28 73 2 28
        L DAYSDIFF R
3 2
```

DEB *INPUT*

Syntax: *result* ← *DEB text*

Removes all extra blanks (leading, trailing, and multiple) from the character vector *text*.

```
      DEB '  The  car cost   $10,960 '
The car cost $10,960
```

DISPLAY *UTILITY*

Syntax: *result* ← *DISPLAY array*

result is the pictorial representation of an array. This is particularly useful in illustrating the structure of a nested array.

```
            DISPLAY ⍳¨⍳3

.→------------------------------.
| .-.   .---.   .-----.   .-------. |
| |1|   |1 2|   |1 2 3|   |1 2 3 4| |
| '~'   '~--'   '~----'   '~------' |
'∊------------------------------'
```

DLB *INPUT*

Syntax: *result* ← *DLB text*

Deletes leading blanks from the specified character vector.

```
      DLB '   THE QUICK BROWN FOX.'
THE QUICK BROWN FOX.
```

DLTB *INPUT*

Syntax: *result* ← *DLTB text*

Deletes the leading and trailing blanks from *text*, a character vector.

```
      (DLTB '   Some   text    '), '!'
Some   text!
```

DSPELL *DATES*

Syntax: *text* ← *DSPELL* *ts*

Displays the date and time in the argument in the form:

DD MMM YY HH:MM:SS:NNN

The time precision of the result depends on the length of the last dimension of the argument. Time is displayed in 24-hour style.

```
      DSPELL 87 10 9 14
9 OCT 87  14:00
```

DTB *INPUT*

Syntax: *result* ← *DTB* *text*

Deletes trailing blanks from the specified character vector.

```
     (DTB '   SOME    TEXT      '),'!'
  SOME    TEXT!
```

DTF *SERHOST*

Syntax: *objectlist* *DTF* *tieno*

Relates to: *DTFALL*, *LFF*, *REP*, *DEREP*

Creates the representation of the objects specified in the left argument and appends them to the APL file tied to the tie number in the right argument. If the left argument is empty, the values of ⎕*IO*, ⎕*PW*, ⎕*CT*, ⎕*RL*, ⎕*SA*, ⎕*LX*, ⎕*ALX*, and ⎕*ELX* are represented and filed.

The left argument is either a matrix of object names to be filed or a vector of names separated by spaces. If the workspace parameters are to be filed, the left argument is an empty vector. The right argument is the tie number of the file to which *DTF* appends the representation of the objects.

```
      'FN1 FN2' DTF 13
Starting size is 1 1 2048 0
FN1 filed
FN2 filed
Ending size is 1 3 3050 0
```

DTFALL — *SERHOST*

Syntax: *DTFALL tieno*

Requires: *DTF*

Relates to: *DTF*, *SENDTFILE*, *LFF*, *REP*, *DEREP*

Writes all of the workspace environment parameters, the variables, and the functions to a "transfer" file in the standard representation format.

The argument is the tie number of the APL file into which the function writes the objects.

```
      DTFALL 21
Starting size is 1 1 2084 0
⎕IO filed
⎕PP filed
    :
    :
(Display continues.)
    :
Ending size is 1 1025 12560 0
```

DTFN — *TRANSFER*

Syntax: *object DTFN tieno*

Appends the source code of the functions supplied in *object* to the native file specified by *tieno*.

```
      'FN1 FN2' DTFN ¯13
Starting file size is 0
FN1 filed
FN2 filed
Ending size is 3050
```

DTFNALL *TRANSFER*

Syntax: *DTFNALL tieno*

Appends the source code of all the functions in the current workspace to the native file specified by *tieno*.

```
      DTFNALL ¯21
Starting size is 0
⎕IO filed
⎕PP filed
   :
   :
```

(Display continues.)

```
   :
Ending size is 21065
```

DUMPFILE *SLT*

Syntax: *fileid DUMPFILE sltid*

Appends a component file to a source level native file. The file is stored as though it was a workspace with variables comp1, comp2, ..., comp*n* representing each component of the file. This allows you to retrieve the data later from the native file into a component file with *LOADFILE*, or into a workspace with *LOADWS*. The component file is specified by tie number or name (*fileid*). The native file is specified by tie number or name (*sltid*).

```
      23 DUMPFILE ¯1
NATIVE FILE SIZE:  1629
.............  (One dot displayed for each component.)
NATIVE FILE SIZE:  8537
```

DUMPWS *SLT*

Syntax: *DUMPWS sltid*

Appends the current workspace (functions, variables, and workspace-dependent system variables) to the file. The file is a native file and is specified by name or tie number (*sltid*).

```
      )LOAD MYWORK
      )COPY [APL.REL1]SLT
      DUMPWS 'STORE.WRK'
NATIVE FILE SIZE:  3218
⎕PP
⎕IO
⎕CT
 :
 :
```

(Display continues.)

```
 :
NATIVE FILE SIZE:  8943
```

EXPLAIN

Syntax: *result* ← *EXPLAIN fnname*

Returns all the initial public comments from the function specified by *fnname*.

```
      EXPLAIN 'CXACOSH'
CXARRZ←CXACOSH CXARR -- COMPUTE THE
```

FTIMEBASE *DATES*

Syntax: *result* ← *FTIMEBASE ts*

Converts the dates and time in *ts* to single numbers representing elapsed microseconds since 00:00, January 1, 1900.

```
      FTIMEBASE ⎕TS
2736769242000000
```

FTIMEFMT *DATES*

Syntax: *text* ← *FTIMEFMT elapsed*

Converts scalars representing elapsed microseconds since 00:00, January 1, 1900, and formats the result in the form:

DD MMM YY HH:MM:SS:NNN

Time is displayed in 24-hour style.

```
      FTIMEFMT ⎕WSTS
10/13/86  19:56:15.0000
```

FTIMEREP *DATES*

Syntax: *result* ← *FTIMEREP elapsed*

Converts scalars representing elapsed microseconds since 00:00, January 1, 1900, to dates in ⎕*TS* timestamp form. *result* is an integer array of dates corresponding to the elements of *elapsed.*

```
      ⎕WSTS
2730387878000000

      FTIMEREP ⎕WSTS
1986 1 4 12 56 43 685
```

HOURBASE *DATES*

Syntax: *result* ← *HOURBASE dateshours*

Converts dates and hours in the argument to single numbers representing the elapsed hours since 00:00, January 1, 1900. *dateshours* is an integer array whose last dimension is 4; typically, a vector in the form 4↑⎕*TS*.

```
      HOURBASE 77 10 25 14
682118
```

HOURREP *DATES*

Syntax: *result* ← *HOURREP elapsed*

Converts scalars representing the elapsed hours since 00:00, January 1, 1900, to dates and times in 4↑⎕*TS* format.

```
      HOURREP 682118
1977 10 25 14
```

LEAPYR — *DATES*

Syntax: *result* ← *LEAPYR year*

Returns a Boolean value representing whether the year specified in the argument is a leap year. The argument *year* is the year in two- or four-digit form; the 1900s are assumed when two digits are used. The *result* is 1 if the year is a leap year.

```
      LEAPYR 1970+⍳10
0 1 0 0 0 1 0 0 0 1
```

LFF — *SERHOST*

Syntax: *LFF tieno*

Relates to: *DTF*, *DTFALL*, *REP*, *DEREP*

Takes the objects stored in transfer format in the APL file referenced by the tie number (*tieno*) and creates those objects in the active workspace.

The example recreates a workspace that had previously been stored in the file named *DTFFILE*. This is the reverse of *DTF*.

```
      )CLEAR
      )COPY [APL.REL1]SERHOST LFF
      'DTFFILE' ⎕FTIE 10
      LFF 10
⎕IO←
⎕PP←
 :
 :
```

LFFN — *TRANSFER*

Syntax: *LFFN tieno*

Recreates the objects stored in the native file specified by *tieno*. This is the reverse of *DTFN*.

```
        )CLEAR
        )COPY [APL.REL1]UTILITY LFFN
        'DTFN FILE' ⎕NTIE ¯10
        LFFN ¯10
⎕IO←
⎕PP←
 :
 :
```

LJUST *FORMAT*

Syntax: *result* ← *formatstring* *LJUST* *title*

formatstring is usually the same format string that was used to produce the report, but it is can be any format string with an appropriate width, or it can be the result *RWTD*. *title* is a character vector containing a title. The text in *title* is left-justified within the width of the format string and returned as a one-row matrix.

```
      LT←F1 LJUST 'THIRD UPDATE'
      CT ◊ '' ◊ LT ◊ '' ◊ REP1
  ANNOUNCEMENT OF NEW DATA

THIRD UPDATE

JAMES           37     $35,815.26
ROGAN           30     $29,437.20
TAYLOR          67     $65,870.64
```

LOADFILE *SLT*

Syntax: *fileid* *LOADFILE* *sltid loc*

Recreates a component file from a source level native file. The source level native file should have been created with *DUMPFILE*. The right argument is a two-element vector specifying the native file and the location in the file to find the requested source code. *sltid* can be specified either as a tie number or a file name. *loc* can be specified as the offset from the beginning of the file or as a workspace name.

Since *sltid* and *loc* can either be a character string or a numeric value, the right argument may either be a simple numeric vector or a nested array.

```
       'NEWFILE' ⎕FCREATE 13
       13 LOADFILE 'XFILE.SLT' 'FILE'
OFFSET:  1652         WSID:  FILE TEST
FROM:  APL*PLUSD      VERSION 1.0 06 AUG
87 VMS
     :
     :
OFFSET 50866 END OF FILE
```

LOADWS SLT

Syntax: *wsid LOADWS sltid loc*

Retrieves a workspace from a file. The file is a native file containing APL source code. It is specified by name or tie number (*sltid*). *loc* specifies the location in the file to retrieve the workspace as an offset from the beginning of the file, or the name of the workspace.

The right argument to LOADWS is a two-element vector. Since *sltid* and *loc* can either be a character string or a numeric value, the right argument may either be a simple numeric vector or a nested array.

wsid is the name of the resulting workspace (⎕WSID) and is optional. If specified, it must be a character vector valid for assignment to ⎕WSID.

```
       'WICTEST' LOADWS ¯1 961
OFFSET:  961          WSID:  WS TRANSFER
⎕PP
⎕IO
  :
  :
(Display continues.)
  :
OFFSET:  14014     WSID:  FILE  XFILE
SAVING WICTEST
WICTEST SAVED 17:59:31 08/07/87
```

MDYTOYMD *DATES*

Syntax: *result* ← *MDYTOYMD mdy*

Converts dates in the form month-day-year to dates in the form year-month-day. The argument *mdy* is an array of dates represented as MMDDYY or MMDDYYYY.

```
      T←2 2⍴20577 42577 102077 61077
      MDYTOYMD T
770205 770425
771020 770610
```

MINBASE *DATES*

Syntax: *result* ← *MINBASE datestimes*

Converts dates and times to single numbers representing the elapsed minutes since 00:00, January 1, 1900. *datestimes* is an integer array of dates whose last dimension is 5. Typically, it is a vector in 5↑⎕TS form.

```
      MINBASE 77 10 25 14 10
40927090
```

MINREP *DATES*

Syntax: *result* ← *MINREP elapsed*

Converts scalars representing the elapsed minutes since 00:00, January 1, 1900, to dates and times in 5↑⎕TS format.

```
      MINREP 40927090
1977 10 25 14 10
```

PERMX *DEMOAPL*

Syntax: *result* ← *PERMX n*

result is a table of the permutations of numbers from ⎕IO to *n*. The number of rows in the table is equal to ! *n* .

```
      PERMX 3
1 2 3
2 3 1
3 1 2
2 1 3
1 3 2
3 2 1
```

PRIMES *DEMOAPL*

Syntax: *result* ← *PRIMES n*

result is a numeric vector containing all the prime numbers from 1 to *n*.

```
      PRIMES 30
2 3 5 7 11 13 17 19 23 29
```

RJUST *FORMAT*

Syntax: *result* ← *formatstring RJUST title*

formatstring is a character vector usually containing the same format string that was used to produce the report, but it can be any format string with an appropriate width, or it can be the result of *RWTD*. The title is right-justified within the width of the format string and returned as a one-row matrix.

```
      RTITLE ← F1 RJUST 'JULY 27, 1987'

      '' ◊ RTITLE ◊ '' ◊ REP1

                 JULY 27, 1987

JAMES          37    $35,815.26
ROGAN          30    $29,437.20
TAYLOR         67    $65,870.64
```

ROWNAMES *FORMAT*

Syntax: *result* ← *shape ROWNAMES rownames*

shape is a numeric vector or singleton containing up to two integers which specify the dimensions of the matrix of row names.

rownames contains the row names as a character vector. The first character in *rownames* is a separator character for each new row name. Each time the function reaches a separator, it skips to the next row. *result* is a character matrix containing *rownames* arranged in a column format.

If *shape* contains two elements, the absolute value of the first element is the number of rows in *result*. If the absolute value of the first element specifies more rows than separator characters in *rownames*, extra rows are padded with blanks at the bottom if the first element is positive and at the top if the first element is negative.

The absolute value of the second element in *shape* is the number of columns in *result*, unless the second element is zero. When the second element is zero, *result* has as many columns as the maximum number of text characters between separators. If the second element is positive, the row names are left-justified; if it is negative or zero, the row names are right-justified. If the number of columns specified is insufficient, the row name field is filled with stars.

```
      3 ¯6 ROWNAMES '=SUNNY=SIDE=UP'
 SUNNY
  SIDE
    UP
```

If *shape* contains one element, that element controls the number of columns in the character matrix. If the element is positive, the row names are left-justified; if it is negative or zero, the row names are right-justified. The number of rows in *result* is determined by the number of separator characters in the right argument.

```
      S←'⊤SMITH⊤VASSAR'
      T←'⊤BRYN MAWR⊤RADCLIFFE'
      9 ROWNAMES S,T
SMITH
VASSAR
BRYN MAWR
RADCLIFFE
```

If both elements of *shape* are missing, *result* has as many rows as there are separator characters and as many columns as the maximum number of text characters between separators. The row names are left-justified.

```
      T←'?NEVER?SOMETIMES?ALWAYS'
      (⍳0) ROWNAMES T
NEVER
SOMETIMES
ALWAYS
```

The first format phrase in the format string should provide formatting instructions for the character matrix of row names.

```
      F1 ← '12A1,X1,6A1,T28,I5,'
      F2 ← 'P< $>CF11.2'
      T ← '*AREA*NAME*SALES*VALUE'
      CNAME ← (F1,F2) COLNAMES T
      T ← '↑TERRITORY 1↑TERRITORY 2'
      T ← T,'↑TERRITORY 3'
      RNAME ← 3 12 ROWNAMES T
      DATA ← RNAME NAMES SALES VALUES
      REPORT2 ← (F1,F2) ⎕FMT DATA

      CNAME ◊ REPORT2
AREA          NAME    SALES        VALUE
TERRITORY 1   JAMES      37   $35,815.26
TERRITORY 2   ROGAN      30   $29,437.20
TERRITORY 3   TAYLOR     67   $65,870.64
```

RWTD *FORMAT*

Syntax: *result* ← *RWTD formatstring*

formatstring, a character vector, is any valid left argument to ⎕*FMT*. *resu* a numeric matrix with four columns and as many rows as there are format phrases in *formatstring*. The columns have the following interpretation:

Column 1	Number of repetitions
Column 2	Width of field, or relative tab if *X*, or the equivalent relative tab if *T*
Column 3	Type of field, as follows:

0	*G*	pattern
1	*F*	fixed point
2	*I*	integer
3	*E*	exponential or floating-point
4	*A*	character
5	*X*	relative tab
6	*<text>*	character text
7	*T*	absolute tab

Column 4	Number of decimal positions for fixed-point format, number of significant digits for exponential format, zero otherwise.

SECBASE — *DATES*

Syntax: *result* ← *SECBASE datestimes*

Converts dates and times to single numbers representing the elapsed seconds since 00:00, January 1, 1900. The argument *datestimes* is an integer array of dates whose last dimension is 6. Typically, it is a vector in 6↑⎕*TS* form.

```
      SECBASE 77 10 25 14 10 56
2455625456
```

SECREP — *DATES*

Syntax: *result* ← *SECREP seconds*

Converts scalars representing the elapsed seconds since 00:00, January 1, 1900, to dates and times in 6↑⎕*TS* format.

```
      SECREP 2455625456
1977 10 25 14 10 56
```

TIMEBASE *DATES*

Syntax: *result* ← *TIMEBASE ts*

Converts the date specified by the argument to the number of elapsed milliseconds since 00:00, January 1, 1900.

```
      TIMEBASE 77 10 25 14 10 56 0
2455625456000
```

TIMEFMT *DATES*

Syntax: *result* ← *TIMEFMT ts*

Formats dates and times specified in the argument in the form:

MM/DD/YY HH:MM:SS:NNN

The precision of the time depends on whether the last four elements of *ts* are present.

```
      TIMEFMT 77 12 31 12
12/31/77  12:00
      TIMEFMT ⎕TS
8/15/87 09:31:25.000
```

TIMEREP *DATES*

Syntax: *result* ← *TIMEREP elapsed*

Converts scalars representing elapsed milliseconds since 00:00, January 1, 1900, to dates and times in ⎕*TS* form.

```
      TIMEREP 2455625456000
1977 10 25 14 10 56 0
```

UNBLOCKS *SERHOST*

Syntax: *oldtieno UNBLOCKS newtieno*

Converts the native file specified as a tie number by *oldtieno* to an unblocked Stream_LF file tied to *newtieno*. *oldtieno* may optionally be a 2-element numeric vector in which the second element is the oringinal data size. It is intended for use in converting files created by Kermit.

```
      'OLDFILE' ⎕NTIE ¯1
      'NEWFILE' ⎕NCREATE ¯2
      ¯1 627 UNBLOCKS ¯2
```

WKDAYSDIFF   *DATES*

Syntax: *result* ← *date1* *WKDAYSDIFF* *date2*

Calculates the number of weekdays between the corresponding dates in the arguments.

```
      86 10 15 WKDAYSDIFF 86 10 1
10
```

WSLIB   *SLT*

Syntax: *WSLIB sltid*

Displays a listing of the workspaces stored in the source level transfer file. The file is a native file and is identified by name or tie number (*sltid*).

```
      'MYFILE.SLT' ⎕NTIE ¯1
      WSLIB ¯1
OFFSET: 961    WSID: WS TRANSFERWS
OFFSET: 14014  WSID: FILE TRANSFERFILE
OFFSET: 50868  END OF FILE.
```

YMDTOMDY   *DATES*

Syntax: *result* ← *YMDTOMDY* *ymd*

Converts dates in the form year-month-day to dates in the form month-day-year. In the example, the dates are put in the correct form and then formatted with ⎕*FMT*.

```
      FSTR←'G<ZZ/ZZ/ZZ>'
      T←870527 870303 870424 871216
      FSTR ⎕FMT YMDTOMDY 2 2⍴T
5/27/87  3/03/87
4/24/87 12/16/87
```

INDEX

Index

U = *APL ⋆PLUS System User's Manual*
R = *APL ⋆PLUS System Reference Manual*